Towards Realistic Interactive Sand: A GPU-based Framework

by

Juan-Pierre Longmore

supervised by

Patrick Marais and Michelle Kuttel

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

Master of Science

The Department of Computer Science
University of Cape Town

November 25, 2009

Abstract

Many real-time computer games contain virtual worlds built upon terrestrial landscapes, in particular, "sandy" terrains, such as deserts and beaches. These terrains often contain large quantities of granular material, including sand, soil, rubble, and gravel. Allowing other environmental elements, such as trees or bodies of water, as well as players, to interact naturally and realistically with sand, is an important milestone for achieving realism in games.

In the past, game developers have resorted to approximating sand with flat, textured surfaces that are static, nongranular, and do not behave like the physical material they model. A reasonable expectation is that sand be granular in its composition and governed by the laws of physics in its behaviour. However, for a single PC user, physics-based models are too computationally expensive to simulate and animate in real-time. An alternative is to use computer clusters to handle numerically intensive simulation, but at the loss of single-user affordability and real-time interactivity.

Instead, we propose a GPU-based simulation framework that exploits the massive computational parallelism of a modern GPU to achieve interactive frame rates, on a single PC. We base our method on a discrete elements approach that represents each sand granule as a rigid arrangement of particles. Our model shows highly dynamic phenomena, such as splashing and avalanching, as well as static dune formation. Moreover, by utilising standard metrics taken from granular material science, we show that the simulated sand behaves in accordance with previous numerical and experimental research. We also support general rigid bodies in the simulation by automated particle-based sampling of their surfaces. This allows sand to interact naturally with its environment, without extensive modification to underlying physics engine. The generality of our physics framework also allows for real-time physically-based rigid body simulation *sans* sand, as demonstrated in our testing. Finally, we describe an accelerated real-time method for lighting sand that supports both self-shadowing and environmental shadowing effects.

Acknowledgments

I have the privilege here to brag of having received guidance from some of the very best.

The direction of my research owes a great deal to my advisor, Patrick Marais, who introduced me to programmable GPUs. His enthusiasm for my unusual ideas, as well as his sobering counsel, have guided me throughout my postgraduate research. I thank him for his support, the hours spent proofing numerous drafts, and abiding my "perfectionism".

Whatever the weaknesses of this thesis, they would have been far greater without the help of my co-advisor, Michelle Kuttel, whose suggestions continually proved sagacious and discerning. I offer my sincerest apologies to her and Neil Ravenscroft for the demise of their 8800 GTX. Sebastian Wyngaard and Patrick Marais deserve commendation for courageously lending me theirs.

I would also like to thank all other members of the Geometry Interest Group (GIG) for their support and generous advice. In particular, I am grateful to James Gain, Ashley Reid, Rudy Neeser, and Ian Tunbridge, with whom I participated in a successful joint application to NVIDIA for research equipment.

The University of Cape Town Funding Office provided funding for my research through the University Committee, Jock Beattie, KW Johnstone, and Harray Crossley scholarships.

Above all, it is to my friends, who put up with my hermetic lifestyle; my parents, who have given me constant support; and Keri, who remains a ray of sunshine through difficult times, that I dedicate this work.

Contents

Part I

Preliminary

Chapter 1

Introduction

Modern computer games often use complex terrains as navigable environments in which players can travel about and interact. Sandy terrain, in particular, features extensively in games depicting natural outdoor settings. Often these terrains contain highly detailed grasses and trees, which may sway in simulated wind or bend aside from passing vehicles or players. Where such vegetation is sparse, however, the underlying surface is visible. Here game engine designers have resorted to modelling sand as textured geometry, which, under close inspection, lacks the granular behaviour and physical interactivity we expect to see. For deforming sandy surfaces, where geometry changes in response to an applied force, the result is still unattractively continuous, if not physically implausible. Thus, while low level-of-detail models may be enough for simple terrains that are hidden from view, there is a sore lack of high level-of-detail models for visible sandy terrain.

Producing high level-of-detail *virtual* sand entails modelling both its appearance and behaviour of its real-world counterpart. In reality, sand may be characterised physically as a *granular material*: a collection of solid macroscopic particles called granules [JN92]. Granules, as the sole constituent of sand, therefore determine the appearance of sandy terrain, while the forces produced both during collision among granules and between each granule and the surfaces in the environment, determine the global behaviour of the material [JN92]. This mechanical simplicity, however, belies the many ways in which granular materials, like sand, differ from *other* substances in both appearance and behaviour. We are accustomed to dividing matter into gases, liquids, and solids. Granular materials span these categories.

Consider a sand dune. Exerting a force vertically into its surface produces *solid-like* resistance, while its surface gives way to a relatively small force when pulled, comparable to scooping out a volume of *thick fluid* [HL98]. Unlike a fluid, however, sand must expand before it can deform, since granules need to first disentangle themselves from their tight packing. This makes sand prone to clogging a conduit, such as a metal pipe, when forced through. Yet sand, like many materials, can *transition* from a solid-like packed state to a fluid-like flowing state, as in an avalanche. Importantly, this may occur without a significant change to thermal energy [JN92],

in contrast to the melting of most solids [HRW93].

Granular materials also have many distinct behaviours. Granules of the same size, for example, will separate into layers when an inhomogeneous granular mixture is subjected to vibration. Such size segregation is commonly referred to as *Brazil-nut effect* [HQL01]. Specifically, if we shake a container of variously sized granules (or nuts), we notice that larger granules float to the top of the mixture, while smaller ones move downward. This self-sorting of granules occurs in part from the downward migration of smaller grains through open channels between larger grains [RSP+87].

Another distinctive behaviour of sand is due to the geometric packing of granules. *Localized stress tracts* [JNB96] result from the *arching* of granules in heaped sand, as demonstrated in Figure 1.1. This results in individual granules having a large effect on the properties of the heap. For example, if we fill a large empty grain silo to the brim with water, we expect the pressure at the bottom of the silo to be proportional to its height. With a sand-filled silo, however, no such dependency exists. Both the arcing and the friction along the walls of the container are sufficient to withstand the extra weight lying above [JN92].

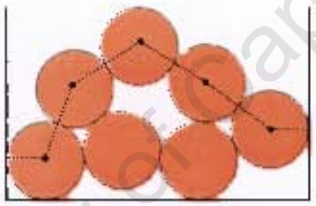


Figure 1.1: Stress tract in a sand pile. Like a more ordered arrangement of bricks that make up an archway, a random arrangement of granules may form arcs (dotted black line) with empty spaces below. Such arcs can serve as networks for conducting force through the sand to the sides of the container. Subject to enough extrinsic compression, however, these arcs may collapse and consume the empty space beneath.

A natural way of producing highly detailed granular behaviours, is to simulate sand mechanically at the level of individual grains using Newton's laws of motion. Unfortunately, both maintaining the physical properties of millions of grains and calculating the forces that occur among them, makes physically-based approaches prohibitively expensive for *a single PC*.

Research has tried to address this problem through *cluster-based* methods that spread simulation workload across multiple processing units (CPUs) in a computer network [WII89, KK95, CHP+96, Fer01]. In the time taken by a single processor to simulate a small volume of sand, a cluster simulates the granular behaviour of a much larger volume, as needed for realistic landform modelling. Computer generated scenes in films are one area where this has been applied. For example, in the recent film *Spider-Man 3*, a central character in the story, *Sandman*, is composed entirely of sand. A key scene shows Sandman emerging from a pile of sand and taking on human form [ABC+07]. This computer-generated set-piece includes close ups of flowing sand grains. A later scene, shows a larger version of Sandman arising from a pile of sand *and* debris at a construction site [PT07]. Thus, virtual sand is seen interacting with itself and other environmental materials. Such highly-detailed behaviour is important to the believability of the final animation.

yet requires many hours of computation using a large cluster [PT07]. Moreover, maintaining and cooling a cluster is simply too expensive for a *single user* to manage.

As an alternative to cluster-based methods, much recent research has explored the use of commodity graphics hardware to run physics calculations in parallel on a single PC. In particular, modern graphics cards have programmable *graphics processors* (GPUs) [NVI08a] supporting general purpose computation. The massively-threaded GPU found on most commodity graphics cards, supports a high degree of thread parallelism. Importantly, each thread is able to run numerical computations written in one of the many high-level programming languages available for this purpose [FK03, KBR08, Hal08].

Previous work has successfully implemented fluid [Har05], smoke [Lla07], fire [ZWF+03], clouds [HBS+03], hair [KHS04], and cloth [HK06] simulation on a GPU rather than a cluster, producing results of comparable quality at a significantly faster rate. However, apart from hair, most of these simulations model the underlying material as a *continuum*, or fluid, which fails to capture many granular behaviours, as previously mentioned.

In contrast, *distinct element methods* (DEM) take a constitutive approach that models *individual* sand granules, thus remaining faithful to the granular composition of real sand [JN92, Faz07]. In particular, a DEM represents each granule as a single particle or *soft sphere*, which accepts some interpenetration with other spherical particles. Using suitable physical laws to oppose this interpenetration, plausible responses to collisions and resting contacts are modeled [SDW96, BD98]. A resulting frame of animation might look like the illustration in Figure 1.1 for a two-dimensional DEM.

While some DEM models exist for the offline simulation of sand on a cluster [BYM05, Fer01], only recent research has simulated granular material on a GPU [YHK08]. However, real-time scene lighting effects, such as granule self-shadowing and environmental shadows, were not explored for the three-dimensional material. Instead, two-dimensional DEM without GPU acceleration [MSW+08] and three-dimensional non-DEM models [LM93, Nor06, DBM07] have received attention. In addition, previous GPU-based DEM models for sand use simple force models [YHK08], which are known to produce unrealistic behaviour [SDW96]. Finally, previous GPU-based DEM simulations fail to address the need for an easy method to add arbitrary objects into a sandy environment and allow these objects to interact with sand. Cluster-based sand simulation, however, provides such methods [BYM05].

Thus, an unexplored question is whether granular sand, in both appearance and behaviour, can be reproduced in real-time using a physically-based high level-of-detail DEM approach, accelerated by the massive parallelism available in a modern GPU.

1.1 Our approach

To recapitulate, there are three major problems with current sand representations in virtual environments:

Problem 1 The sand modeled in games and other real-time applications lacks the granular behaviour we expect to see, including piling, rolling, and shifting.

Problem 2 Similarly, in-game sand often lacks real-time physical responses to other materials in its environment. That is, sand cannot flow around fixed obstacles, react to impact from a projectile, or be transported in a container.

Problem 3 In addition, without explicit granular representation, producing subtle surface texturing due to grain self-shadowing and the shadows cast by sand on its environment (including characteristically uneven dune silhouettes) is nontrivial.

This thesis addresses these problems with a unified architecture that produces real-time granular sand, in both appearance and behaviour, for arbitrary environments. The aforementioned problems are dealt with naturally in three parts, as sketched below. The ultimate view is that this high level-of-detail granular simulation will form part of future research into a complete level-of-detail system for sandy terrain simulation.

1.1.1 Problem 1: GPU-based simulation of granular material

The first component is a shader-based physics framework for simulating the dynamical behaviour of rigid arrangements of one or more soft-sphere particles on the GPU. In particular, a modified distinct element method is employed, which models each sand granule as a fixed arrangement of soft-spheres, allowing six degrees of freedom (three rotation, three translational). This leads to nontrivial behaviours, such as complex dune formation, that are computationally difficult to achieve with less sophisticated one-to-one particle-granule models (that is, with only three degrees of translational freedom). Moreover, the simulation produces these complex behaviours in real-time, since granule physics is handled entirely in parallel on the GPU.

For a discussion of the physics involved, see Chapter 3. For GPU-specific implementation details see Chapter 4.

1.1.2 Problem 2: GPU-based mesh-to-particle conversion

The second component simplifies inclusion of arbitrary objects into the simulation by implementing a GPU-based method for transforming the surface of an arbitrary closed triangular mesh into a particle-based representation of itself. This effectively makes the mesh into a large sand granule,

albeit with a different mass and particle count, which is included into the first component with only trivial extensions required to the implementation.

See Chapter 6 for details on a GPU-based conversion implementation.

1.1.3 Problem 3: Lighting the sand

Shadows are important for the feeling of depth they provide to otherwise flat objects. The third component, therefore, implements real-time GPU-based rendering and shadowing of sand granules. Both self-shadowing among grains and environmental shadowing by grains is supported. This implementation forms the visualisation part of the first component mentioned above.

See Chapter 5 for more details on a well-known shadowing technique that has been adapted to accelerate the rendering of large sand granule systems.

1.2 Contributions

Previous work has simulated the granular behaviour of sand exploiting the parallelism available on clusters to accelerate expensive physics. Recent work has looked at GPU based particle simulation as an alternative to cluster techniques. Our work improves on the latter in the following ways:

- We have implemented a GPU-accelerated real-time particle simulation of sand that performs *all physics calculation* and rendering on the GPU. In particular, the framework demonstrates simulation performance that is competitive with similar GPU based DEM simulations, while exceeding previous CPU-based simulations.

- We have accounted for inter-granule forces specific to sand. In particular, the simulation framework demonstrates *physically valid* sand behaviour, as verified by standard metrics used in the field of granular material science.

- We have developed and implemented *a novel lighting method* to accelerate complex *self shadowing* and *surface shadowing* for granules.

- We have developed and implemented *a novel GPU-based mesh-to-particle conversion method* that rapidly produces a particle-based representation of a mesh for direct inclusion into our simulation. This permits construction of an arbitrary environment able to interact with sand.

- The *generality* of our GPU-accelerated physics engine allows for real-time *physically-based rigid body simulation*, irrespective of the presence of granular material.

1.3 Thesis organisation

The remaining chapters of this thesis are organised as follows: Chapter 2 reviews previous work on sand simulation techniques relevant to the whole thesis, while subsequent chapters begin with their own specifically relevant background. Chapter 3 presents a distinct element model able to produce complex granular behaviour. In Chapter 4 we introduce our GPU-based method for accelerating the sand model. Chapter 5 extends the GPU-based approach to add real-time lighting. Chapter 6 introduces a GPU-based method for converting arbitrary meshes into objects that may be included into the simulation and interact with sand. Chapter 7 presents physical validation of the claimed granular behaviour. It also looks at performance results for simulation and rendering. Conclusions and future work are presented in Chapter 8, which ends the main body of the thesis. An Appendix presents additional technical details, proofs, and source code, which supports the main text.

Chapter 2

Background

Sand is a term that may have different meanings to different people. If one lives near a beach, for example, perhaps white dunes of coral sand come to mind. Still others may imagine hundreds of square kilometers of desert or the alluvium deposited by rivers. This thesis, however, addresses only a subsection of the great variety of sands, namely the *dry* granular type, made of small but visible grains.

The exclusive interest of this thesis is GPU-based simulation of this dry granular sand using a high-fidelity DEM model. This chapter, therefore, describes previous work in DEM modelling and GPU based acceleration of granular material simulation. In addition, methods unrelated to DEM and methods that may easily extend to modelling granular material are discussed for comparative purposes.

This chapter begins with an overview of previous work in granular material simulation, followed by separate sections discussing individual techniques in greater detail.

2.1 Overview

Previous work has attempted to model dry granular sand with various representations for the underlying material. As mentioned in Chapter 1, sand's behaviour arises from fine-scale granule responses to forces acting on the material as whole. This results in a dichotomous relationship between the individual granule responses and the collective behaviour of thousands of granules. Indeed, this relationship is used in this chapter to classify the level-of-detail of the underlying material representation, as illustrated in Figure 2.1.

At one extreme, *continuum methods* do not consider granules at all, but model sand as a uniform volume of viscous fluid. The advantage of this simpler representation, however, must be balanced against the loss of certain granular effects, since no separate material pieces are being simulated.

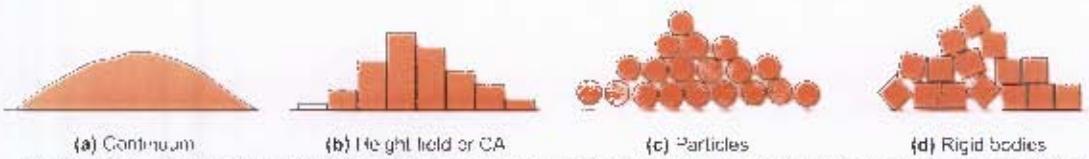


Figure 2.1: Different level-of-detail in sand representation. Moving from left to right, the model of sand moves from a global mode of behaviour to a localised model. *Continuum* refers to a viscous fluid model, while *height fields* are a special case of cellular automata (CA).

One step beyond a purely continuous representation is to subdivide both the material and the simulation space housing that material into discrete subvolumes. The methods modelling sand in this way are called *discrete methods*, which include *cellular automata* (CA) and *height-fields*. These two methods differ in that a cellular automaton moves its indivisible subvolumes around through the discrete simulation space, while a height-field shifts sand volume between its *fixed-position* subvolumes.

Distinct element methods (DEM)[1] are a more detailed representation that subdivides a material into many small distinguishable pieces or particles. Particles are modeled as separate indivisible spheres. The difference between a DEM and a CA, is that DEM particles move *freely* through a continuous space, while CA volume elements move from one discrete subvolume position to another, instantaneously. In addition, a DEM usually incorporates the laws of motion, while a CA does not.

Finally, rigid polyhedrons, such as tetrahedrons or cubes could be used, instead of spherical DEM particles. However, these *rigid-body* representations need expensive numerical methods for calculating forces that resolve complex collision patterns among their angular geometry.

We now explore each of these methods in greater detail, with reference to prior work.

2.2 Continuum dynamics

Continuum, or fluid, dynamics ignores the diverse parts that make up a material and does away with handling physical quantities separately for each part. Instead, a continuum deals with physical quantities in the limit, estimating these values for a section of material at a time [Bat00]. This simplification allows continuum approaches to describe a material's mechanical behaviour with differential equations, for which there is a large body of supporting theory and software in the area of computer graphics [BMF07]. Fluid dynamics has been implemented on the GPU as well [GWL+03, Har05].

Applications in engineering and computer graphics have modelled sand as a viscoelastic continuum [III98] because it displays both elastic and viscous properties. Specifically, sand

[1] This thesis distinguishes between the terms *discrete* and *distinct*, though these terms are often used interchangeably when referring to distinct element methods. To avoid confusion, this thesis uses discrete to mean a noncontinuous domain *and* material. In contrast, distinct implies the material alone is noncontinuous.

responds to compression by closing spaces between grains, subsequently returning to its original shape, like an elastic, when the stressor ends. On the other hand, sand resists shear stresses or forces acting parallel to its surface, which is typical of many viscous fluids.

One of the first applications of the continuum approach to granular material simulation was by Savage, who modeled granular mechanics on the continuum equations of motion and kinetic theories [Sav79]. Jenkins and Savage later adapted the thermodynamics equations for granular materials, accounting for the absence of a temperature effect on granule behaviour [JS83].

A recent technique in computer graphics by Zhu and Bridson, uses a fluid-based model for granular material simulation [ZB05]. In this method, a cloud of particles tracks the motion of a surface, while an auxiliary grid preserves incompressibility and boundary conditions. This technique uses an existing fluid solver adapted to include both inter-grain and boundary friction.

Although fluid-based models such as these reproduce many properties of sand, they lack important aspects of granular behaviour. In particular, they often lack solid-like behaviour, such as dune formation, resistance to pushing, and expansion before deformation. Conversely, fluid behaviours may persist in these simulations. Many fluids experience zero divergence, for example, where the same number of particles will enter a region as leave it. But this does not happen to a significant extent in a granular material. In dry sand, both aeration and supporting granular arcs may adapt to compression by collapsing a little first [JN92].

For these reasons, much research has focused on discrete methods, such as the cellular automaton.

2.3 Cellular automata

Canonically, a cellular automaton (CA) [CD98] is a collection of cells, either occupied or empty, forming a grid. It evolves at each time step, according to a set of rules based on each cell's state. In general, a rule set applies a single update scheme, adjusting cells either individually or in groups, as seen in Figure 2.2.

(a) Border occupancy (b) Pattern matching

Figure 2.2: Cellular automata update schema. Cells are either empty (white) or occupied (black). (a) One scheme adjusts cell state (red dot) according to border cell occupancy (blue dots). In this example, having two out of eight occupied border cells leads to cell death. (b) Another scheme matches the occupancy pattern (yellow dots), for each disjoint subsection of the grid, to a specific output pattern.

To model sand, we can form a ground layer from a collection of unmodifiable cells, while marking various modifiable cells in upper layers as occupied by granules. A simulation then adjusts modifiable granule cells at each time step, producing downward flow and eventual sand pile

formation. In this way, a CA equipped with a reasonable rule set can create the illusion of gravity and granular behaviour, as seen in Figure 2.3. This technique easily extends to three dimensions, including consideration for a cell's mass, velocity, and displacement from the ground [LCG+07].

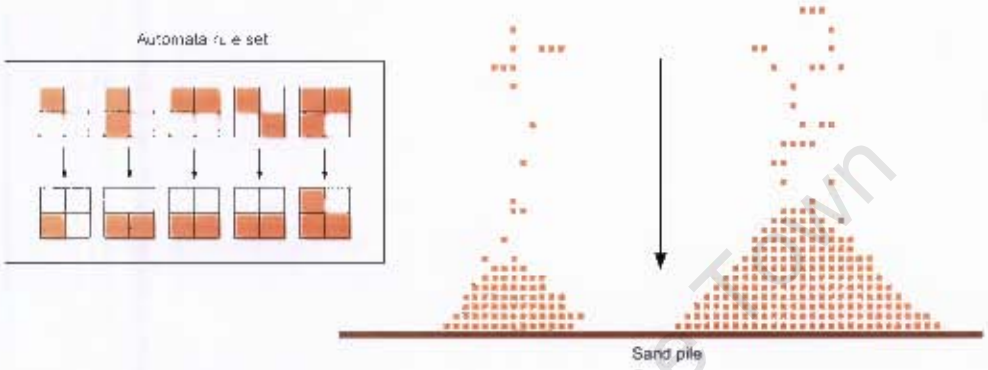


Figure 2.3: Simulation of sand pile with cellular automata. For each neighbourhood of cells, the rule set specifies a unique next arrangement of granules (modifiable cells). For nondeterministic behaviour, rules can have likelihoods associated with their activation. The rules given above mimic the effect of gravity and granule collisions to produce dune formation.

While simple in design, CA models have surprisingly wide application to modelling complex behaviours, such as bacterial and tumour growth, fluid turbulence, smoke evolution, and fire. For granular behaviours, cellular automata have found application in modelling avalanches [BTW87, NG01], sand pile formation [CCH+98], and particle segregation [BB90].

As stated in [LCG+07], CA models are faster than many alternatives and have the ability to model non-smooth geometry. In contrast, continuous models may run slower and are limited to simple surface geometry or spherical particles. Distinct element methods, which model individual particulate elements, face the runtime cost of having to track many particles. Compounding this are the expensive force calculations they must perform for each collision between particles, which may preclude real-time simulation.

CA models, however, are limited in their ability to predict the behaviour of the material they model [LCG+07]. One reason is that CA simulation is discrete, while space is continuous in reality. Another is that material pieces interact physically, while the CA elements behave according to fixed rule set, instead of physical laws. In both cases, unrealistic large-scale behaviour may result.

The lack of physically-based interaction between cells, however, can be rectified by fixing the *position* of the lowest layer of cells and varying their *volume* instead. In computer graphics, this technique is referred to as a height-field.

2.4 Height-fields

A height field is a Computer Graphics approach to volume representation. The technique divides a volume of material into vertical columns lying on a rectilinear grid. As with the cells in a CA, the behaviour of a column is governed by the properties of the columns that border it. In this sense, a height field is a special case of a 2D cellular automaton, with each cell fixed in position and storing at least a height property. The latter allows us to visualise the height field as a 3D surface, as seen in Figure 2.4.

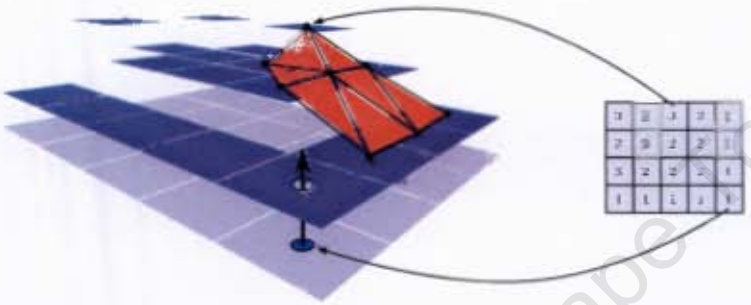


Figure 2.4: Example of a height-field. A 2D rectilinear plane divides 3D space into a grid of rectangular cells. Each cell stores the elevation of a discrete point or a column lying above the plane. We can visualise this grid either as a collection of columns or as a triangulated point set (mesh).

Height-field methods have produced attractive results, including realistic behaviours absent from fluid models. Sumner et al. described an appearance-based method for producing footprints, tyre tracks, and skid marks in a material [SOH99]. The production of these effects depended on various parameters, including a "liquidity" parameter that attempted to model sand, mud, or snow, from low to high water concentration, respectively. However, these parameters do not resemble quantitative physical properties of the material being modeled. Thus, qualitative evaluation is needed to produce the desired result.

Li and Moshell developed a height field simulation to model real-time soil slippage [LM93]. The sand composing the height-field evolves until it reaches a static equilibrium. When this equilibrium is disturbed, such as when the sides of a dune exceed the angle of repose for sand, then volume is shifted to restore the equilibrium. Unlike the method of Sumner et al., this technique estimates volume shifts using a *force model* based on physical properties of sand, such as mass. Their technique also allows users to shift cell volume horizontally, as well as dig and pile sand. Sand interaction, however, is not supported for arbitrary objects, thus precluding effects such as tracks or footprints.

Onoue and Nishita [ON03] use a multi-valued height field to achieve separate volume transport. Each column of sand may have multiple separate sand volumes within that column, where the lowest section rests on the ground plane, while the next rests on an object. In this way, a subvolume can be transported in a bucket across the surface. Later improvements by Zeng et al. added momentum-based object interaction to the height-field, thus enabling more physically-

realistic volume changes [ZTY+07].

Another physically-based model, by Chanclou et al., achieved soil compression, piling, and track formation [CLH96]. They represented the influence of soil on an external object with a particle-based model at the contact surface.

However, these height-field techniques fail to model high-speed events, such as sand spattering away from projectile impact, where granules must rapidly exchange energy [BYM05]. Instead, a particulate model is needed where granules involved in such energy exchange are represented explicitly with their own physical properties such as velocity. Techniques that model individual interacting elements in this way are known as distinct element methods.

2.5 Distinct element methods

The distinct element method (DEM) represents a class of numerical techniques that model the motion of large quantities of particles, such as grains of sand [WHM87]. A seminal application by Cundall and Strack applied a DEM to the modelling of rock mechanics [CS79]. They introduced the use of distinct spherical particles to model rocks, applying ideas from *molecular dynamics* (MD) to granular systems.

While the terms MD and DEM are often used interchangeably, they have important differences [Rap04]. In contrast to DEM, MD particles are restricted to groups of atoms, so that distances are atomic and the time-scale of interactions is in nanoseconds. Further, particles follow a global potential gradient that relies on relative particle positioning. Finally, MD simulations typically use periodic boundary conditions, which reduces the number of particles in the simulation, while removing boundary effects that do not conform to the assumption of a uniform environment.

While DEM and MD particles are both spherical, DEM particles are usually *macroscopic* (rocks and boulders) or *mesoscopic* (granules and powders) in size [CS79]. In addition, interactions between particles occur at perceivable distances and on a millisecond time-scale. Also, these interactions are generally free of thermodynamic effects, which is true of sand as well [JS83]. Finally, the simulation boundaries are modeled explicitly with geometric surfaces.

The application of DEM to modelling sand entails representing mesoscopic granules as particles driven by forces produced during collision. These forces are important, since they determine the overall behaviour of the material [CS79]. Therefore, the next section examines the forces involved in modelling sand granules as spherical particles.

2.5.1 Forces

We may separate the forces experienced by granules into two types [JN92]. First, a strictly repulsive force occurs with head-on collisions, akin to two billiard balls elastically rebounding

off one another. However, since sand comes to rest, a second dissipative force must be at work. One source of this force is *inelastic collision*, which refers to colliding bodies that do not separate after impact. A second source is *tangential friction*, which acts against the relative motion of two colliding bodies.

When granules achieve rest collectively a third less high-energy force takes effect. This force, called *static friction*, acts between bodies that remain in persistent contact. In granular material, it is responsible for holding the granular formation together, producing dunes.

Finally, long-range forces occur between granules. Most of these forces, including *Coloumb forces*, cause insignificant attraction between granules and are safely ignored [JN92]. For mesoscopic and macroscopic particles, however, only the long-range effect of gravity is significant [JNB96].

Those forces important to mesoscopic particles are illustrated in Figure 2.5.

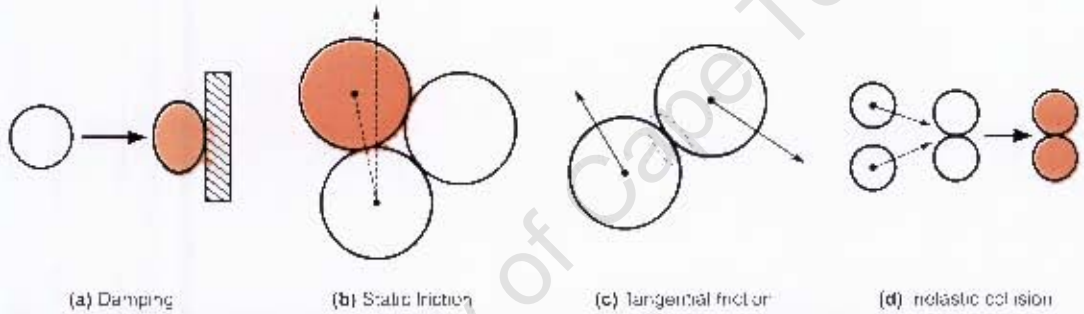


Figure 2.5: Forces due to particle collisions. (a) A slight amount of deformation produces damping, which acts synchronously and oppositely to velocity. (b) Slight deformation at a contact point and static friction keeps a sphere from slipping down the pile. (c) Colliding spheres act tangentially on each other producing friction opposing relative motion. (d) Inelastic collision results in spheres remaining in contact after collision, instead of rebounding off one another.

Given an appropriate force model for the these interactions [SDW96], sand granules can be represented directly as individual particles with associated physical properties such as mass, position and velocity. Particle-based simulation begins by placing all particles in their starting positions and setting their initial velocities. For each discrete time interval, or *time step*, the simulation will need to calculate all forces acting on the granular material. These forces are used to produce the motion of particles, which ultimately results in global material behaviour.

Unfortunately, finding all the forces needed in the last step is a computationally expensive task, since it entails finding which particles have collided. This problem and various solutions are discussed in the following section.

2.5.2 Collision detection

Space-occupying spherical particles, when used to simulate granular assemblies or other materials, must respond to inter-particle collisions. However, finding collisions or possibly colliding particle pairs among thousands of particles is nontrivial.

The problem is easier than for rigid bodies, as discussed in Section 2.6, since we deal only with spheres. A positive collision is when the distance between two particles is less than the sum of their radii. A simulation is further simplified by having all particles of the same radii, since no explicit radius value needs to be stored in memory.

Many force models in particle simulation work on resolving binary contacts as an estimate for the multiple simultaneous contacts that occur in reality [CS79, SDW96, BYM05]. Thus, the problem is to find positive contacts among all the possible contact pairs, which naïvely means checking intersection of each particle against all others, as illustrated in Figure 2.6a.

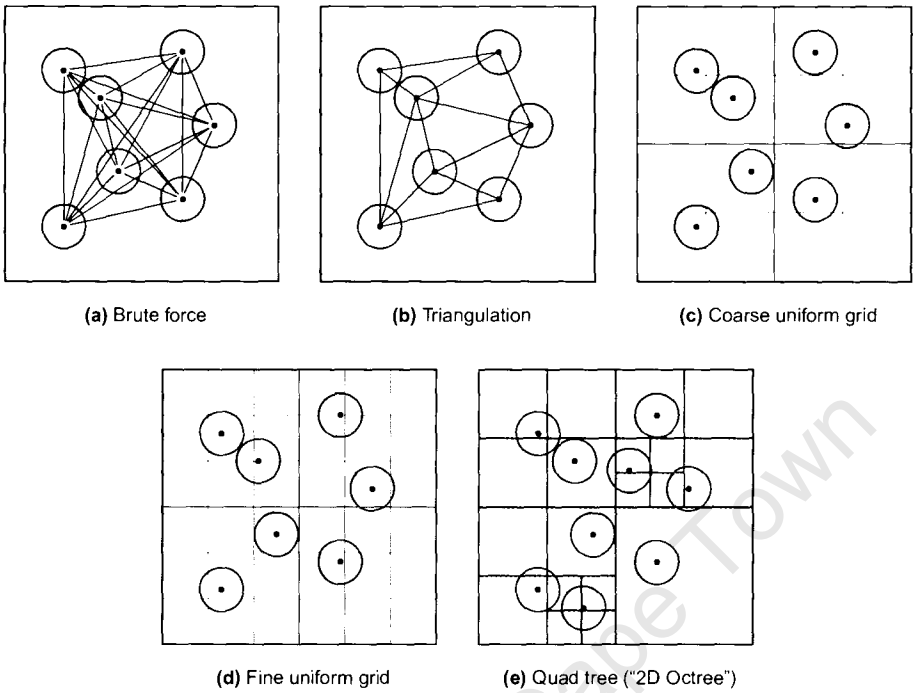
A further optimisation is to consider only particles that lie within a small *neighbourhood* of one another. This simplification is valid, since we have established that forces in a granular material arise either from gravity (a uniform force) or from local contact between particles. Limiting searches to local neighbourhoods lowers the collision search cost for each particle, since the simulation now checks only local particle pairs, rather than all possible pairs.

Various implementations of this idea exist. For sand simulation, one method is to triangulate the network of possible collisions among particle pairs [Fer01], as illustrated in Figure 2.6b. This markedly reduces pair count, but also discards some potential collision pairs. However, the triangulation is used to resolve these lost pairs during simulation.

Another approach is to subdivide the simulation into a rectilinear grid, as shown in Figures 2.6c and 2.6d. The subparts of this grid, called cells, store the identities of particles that occupy the same space as the cell. Such grid-based methods have been used extensively for GPU-based particle system simulation [KLRS04, KKK+05, KvdDP03], but these methods do not necessarily apply to DEM particles. General particles, for example, are often represented as a collection of point sprites, thus avoiding inter-particle collisions calculation, but preventing pile formation. More applicable are the GPU-based methods that treat spherical particles as the space-occupying material that evolves through time based on physical laws [HBS+03, LLW04, Har05, HKK07].

There are many cases when either large storage demands or the lack of uniform grain size become issues in DEM simulation. In these instances, instead of using uniformly divided grids, an adaptive subdivision approach can be taken. In the two-dimensional case, a well-known alternative is the *quadtree* [FB74, Sam84]. For a set of points on a square two-dimensional surface, tree construction consists of recursively subdividing space into four quadrants until the final squares (leaf nodes of the tree) contain only one point, as seen in Figure 2.6e. *Octrees* follow an analogous three-dimensional process that recursively subdivides a cube into eight parts until points are isolated. Importantly, both generic [LSK+06] and specific [KLS+05] GPU implementations exist for this approach.

With collision detection in place, the simulation can now accumulate forces to produce motion of particles. Particles represent approximations to near spherical grains. However, many sand types have grains with faces and edges [Mah02]. In addition, calculating static friction for spherical particles can be computationally expensive, since particle contact has to be tracked across the



(a) Brute force **(b)** Triangulation **(c)** Coarse uniform grid

(d) Fine uniform grid **(e)** Quad tree ("2D Octree")

Figure 2.6: Spatial subdivision for collision detection. (a) The naïve approach is to test each particle for intersection against all others. However, for n particles, this algorithm uses on the order of n^2 pair-wise tests. (b) Triangulating the particles reduces the number of pair-wise comparisons, but may be offset by the cost of constructing the triangulation itself. (c) A faster approach is to consider only particles that lie within a small *neighbourhood* of one another, since only local forces are needed. A grid can be used to divide the domain into equal-sized sections called cells (blue squares) storing granule occupancy based on their centre positions. This limits searches to local neighbourhoods, thus lowering the collision search cost per particle. (d) However, a fine balance exists between grid resolution and cell-size, since *every* cell, whether occupied or not, is allocated in memory. (e) The quad tree and octree try to mitigate excessive space use by storing only occupied cells. A quadtree structure is used to record cells centres by progressively quartering the domain to locate particle centres. However, a construction cost is involved in producing these trees.

evolution of the simulation [LH93].

Bell et al. [BYM05] circumvent the expensive static friction calculation by modelling grains of sand as non-spherical bodies made from an arrangement of spherical particles constrained to move together. Modelling sand grains with an irregular arrangement of geometry has the advantage of automatically handling static friction. In sand, these static forces occur between motionless, packed grains and contribute significantly to dune formation. Since the shape of the granules leads to their interlocking, there is no need to estimate inter-grain static friction with expensive and empirical inter-particle force calculations. Instead, the tangential force used in collision processing will naturally impede slippage.

Another approach is to forgo spherical particles and use polyhedrons with faces approximately matching a desired grain geometry. The simulation of polyhedrons requires rigid body interactions, which is discussed in the next section.

2.6 Rigid body dynamics

Rigid body dynamics deals with objects that occupy finite space and have various geometric properties, including a centre of mass, moments of inertia, and so on [Wit77]. While real-world materials may undergo deformation during collision or resting contact, the geometry of a rigid object is unchanged by an applied force: the distance between any two points inside a rigid body remains constant across time, regardless of any external influences. This avoids the difficulties of having to handle a changing centre of mass; detecting and resolving self-collisions; and calculating internal tension and various other properties related to deformation behaviour.

Fortunately, as sand granules and many other materials experience negligible deformation, we can model them naturally and accurately as rigid bodies. Matuttis et al. have modeled sand in the two-dimensional case as polygons and compared these results to a similar particle-based simulation using two-dimensional spheres (disks) [MLH00]. In both cases, they observed arching and stress-chains in the presence of a rough ground surface, but without intergranule friction. However, spherical granules produced piles with smaller angles of repose. Similar behaviour is observed with regular many-sided polygons, which are close to spheres in shape. This suggests that regular many-sided geometry offers little advantage over a DEM approach.

In addition, three-dimensional particle-based representations allow movement in three mutually exclusive directions (three degrees of freedom). However, a rigid body representation includes rotation around these axes, giving six degrees of freedom in total, as seen in Figure 2.7. This leads to more calculations per moving component in the simulation when using rigid-body dynamics.

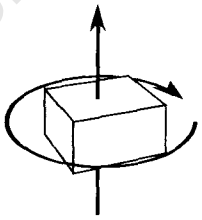


Figure 2.7: One mutually exclusive axis adds two degrees of freedom to rigid body motion.

Nevertheless, rigid-body methods may still apply to simulating certain granule shapes. While little work has been done in computer graphics relating specifically to the simulation of granular material, various rigid body simulation methods may apply to simulating granule behaviour.

Mirtich described an impulse-based approach, improving on previous constraint-based methods that continually calculate separation forces between contact points [Mir96]. This method models all interactions between rigid bodies as collisions. It simulates all types of mechanical behaviour, such as rolling, sliding and resting, in this way. While simple and efficient for many purposes, this model does not handle prolonged contact between bodies correctly and results in oscillation or slow shifting of supposedly stationary stacked objects. Similarly, the method cannot model static fiction. The result is *static friction creep* (Figure 2.8) where objects supposedly at rest slide slowly down an incline. This is a significant drawback for granular material simulation, where

prolonged contact and static friction are integral to proper behaviour.

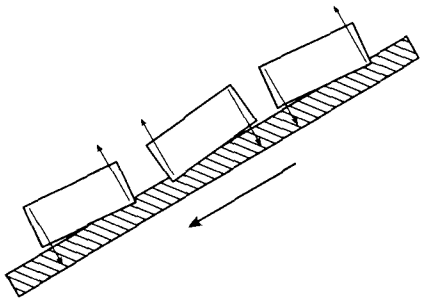


Figure 2.8: An example of static friction creep. Impulse-based approaches can lead to rocking behaviour because of small and sequentially reactive forces produced from seemingly negligible interpenetration. This results in a rigid body progressively sliding down an incline.

Guendelman et al. [GBF03] produced rigid body simulation using a dual representation for geometry as both a triangulated surface and signed-distance function (see Section 6.1) defined on a grid. A time integration algorithm that clearly separates contact resolution from collision was used to prevent artifacts, such as objects which jitter when seemingly at rest. This avoids the common empirical approach of using improvised threshold velocities to remove these special cases from the collision model. They also described a shock propagation algorithm, applied after contact propagation, which prevents stacked objects from slowly interpenetrating. Finally, they show stacking of one thousand nonconvex rigid bodies with friction, thus demonstrating applicability to granular material modelling.

However, thousands of granules are needed to model real landforms. This method and many of those mentioned in other sections are limited by the computational cost of the physics calculations they employ. In Chapter 1, we alluded to the importance of graphics hardware for accelerating these physics calculations. The next section briefly discusses previous work applicable to sand simulation that has taken advantage of graphics hardware to produce real-time results.

2.7 GPU-based simulation

The graphics processor (GPU) has evolved into a powerful data-parallel computational device, specifically designed for arithmetically intensive processing. Much of the driving force behind these developments is an increasing demand for larger rendering performance in games. The bulk of GPU rendering concerns processing batches of vertices and pixels. In particular, the GPU processing is parallel, working separately on transforming vertices or colouring pixels. Owens et al. discusses the many GPU-related algorithms that have exploited this parallelism [OLG+07]. In Section 4, we provide a detailed look at how this parallelism is exploited in this thesis.

Kipfer et al. [KSW04] animated large particle sets by taking advantage of the GPU. A special geometry container, called a pixel buffer object, stores position and motion data for particles, which are sent to the GPU to be updated. The results are then read back to the CPU. Their

method accounts for inter-particle collisions by sorting particles into collision pairs directly on the GPU.

Kolb et al. [KLRS04] produced a GPU-based particle system simulator able to render a dynamically growing particle system of up to one million particles in real-time. Their main contribution is a collision detection algorithm based on surface normal data and depth maps, which capture the distance values of visible surfaces in the scene. The first step is to store this information in textures for fast access by rendering operations. The collision detection algorithm then compares arbitrary particle positions to local objects by reconstituting object surface information from the static texture data. Hardware-acceleration is due in part to rendering operations driving storage, updates, and queries of the texture data. They also present a GPU-based parallel algorithm for depth sorting that orders particles for transparency effects.

Advancement in GPU programmability has lead to more sophisticated simulations being carried out on the GPU. Harada et al. [HKK07] and Venetillo et al. [VC07] have demonstrated particle simulations on the order of millions of particles running in real-time. They perform collision detection, force calculation, and motion updates entirely on the GPU. The results of the simulation are visualised in real-time as well.

Similar to the work presented in this thesis, Yasuda et al. [YHK08] have recently demonstrated granular material simulation on the GPU. However, static friction behaviour is not addressed in their method, which must forgo large angles of repose and complicated pile formation.

2.8 Summary

Unsurprisingly, no single simulation model has accounted for the great variety of granular behaviour exhibited by sand and related materials. In particular, the strongest division among previous methods is the level of detail they use in simulating granules, and the behaviour resulting from this decision, both successfully produced or not. In addition, while many relevant techniques in the real-time simulation of physical materials do not deal with sand or other granular material specifically, they trivially extend to such simulation. Nevertheless, we summarise the capabilities of previous methods in Table 2.1.

Model	Real-time	GPU*	Cluster*	Weaknesses
Discrete	Yes	Yes	No	Lacks physical basis
Continuous	Yes	Yes	Yes	No granular effects
Rigid bodies	Yes	Yes	No	Limited granule count
DEM	Yes	Yes	Yes	Static friction is expensive

Table 2.1: Comparison of methods available for simulating sand. The above comparison assumes a large quantity of material is simulated.

* Implementation platform

The simulation presented in this thesis exploits the granular behaviour and speed advantage of

DEM simulations [HKK07], as well as the modelling accuracy of a non-GPU approach [BYM05] to produce real-time granular material simulation. An underlying DEM force model is developed first to handle complex static friction behaviours that are needed for pile formation. The latter development is the topic of the next chapter.

Part II

Design and Algorithms

Chapter 3

A DEM model for sand

In the previous chapter, we discussed various techniques for simulating the behaviour of sand. We found the DEM approach well suited to modelling sand behaviour at a high level-of-detail, that is, using *grain-grain* interaction. However, we also found the DEM approach has two challenging drawbacks. The first is the overhead and special handling required to produce static friction for *spherical* DEM particles. The second is the expensive physics calculations needed for large particle counts, which model realistic landforms. Given the importance and attractiveness of the DEM approach, however, the latter two problems need to be addressed.

This chapter describes a DEM-based model able to simulate the granular behaviour of sand. The model presented here addresses the first problem of static friction production by modelling grains as multiparticle bodies. Thus, the advantages of DEM are maintained, while mitigating expensive calculation and storage costs through implicit geometric "roughness" of the grains. Intergrain force models are discussed, which are relevant to the behaviour of the material as a whole. The approach presented in this chapter is calibrated to real-world experimental results, where possible. Finally, we discuss the physics involved in producing the resulting grain motion. The remaining problem of accelerating the DEM physics calculations, as presented in this chapter, is left to Chapter 4.

3.1 Overview

The DEM simulation described in this thesis follows the basic procedure outlined in Section 2.5. Collisions between granules are detected; the force produced by granule contact and collision is calculated; and the motions of the granules are updated. The difference is that we use a multiparticle model for individual granules, as suggested by Bell et al. [BYM05], instead of a single particle for each granule. In this model, as shown in Figure 3.1, each multiparticle granule consists of four rigidly positioned particles that move together as a single body through space. Thus, while granules are modeled as a "rigid body", the physically-based handling of the granule

is handled by a DEM, rather than by rigid body dynamics[1]. The advantages of this is the absence of expensive collision tests for vertex, edge, and face geometry, and handling of multiple penetration points for force calculation [Mir96, Dru08].

Furthermore, the *regular tetrahedral* arrangement of *four* particles enjoys two advantages over alternate configurations. Firstly, the centres of any arrangement of three particles (or less) will always lie on a plane, producing a granule with at most two-dimensional rotational symmetry. In contrast, the centres of four particles can form a nonplanar arrangement called a *general* tetrahedron. In the latter case, if we space particles evenly, we have a *regular* tetrahedron with three-dimensional rotational symmetry. Unlike the two-dimensional case, rotational inertia for the regular tetrahedron is the same irrespective of the axis of rotation. This fact both simplifies and accelerates calculation of rotational momenta. We also speculate that concave regions between the spheres promote interlocking among granules, thus mimicking static friction. The second advantage of a tetrahedral arrangement is faster simulation, as the next regular structure with rotationally invariant inertia is a cube, which would require the simulation to manipulate and store twice the number of particles per granule.

Figure 3.1: Particle-granule models. The left model represents a granule in a standard DEM model, which uses a one-to-one granule-particle mapping. The DEM presented in this chapter, however, maps each granule to four particles positioned at the corners of a tetrahedron.

Particle and granule properties, such as their position and velocity, evolve through time, based on the forces they experience. Since we cannot sample these properties with infinite precision, finite observations must be made at particular time intervals. These intervals, called *time-steps*, may vary in length. However, this thesis employs a fixed-size time-step Δt (see Section 3.3). The actions taken by the simulation to produce the observation at time $t + \Delta t$ from the observation at time t are listed in Figure 3.2.

Forces acting on particles over the interval Δt are summed to give the total force acting on their parent granules. Numerical integration of the resulting force allows the simulation to calculate new granule positions and velocities at time $t + \Delta t$. Finally, the resulting granule properties at time $t + \Delta t$ are used to derive the new properties of their particle constituents at that time.

This design assumes that forces have acted for a duration Δt on granules whose properties are *fixed* at time t. Notice that this is not strictly valid, since forces acting on granules change granule properties instantaneously. Nevertheless, with sufficiently small Δt, simulations are able to reliably estimate continuous behaviour [Bar97].

[1]The use of multiparticle granules entails handling particle and granule processing separately. For this reason, both in this chapter and the next, the term *rigid body* is used when referring to a tetrahedral granule, as distinguished from its particle constituents.

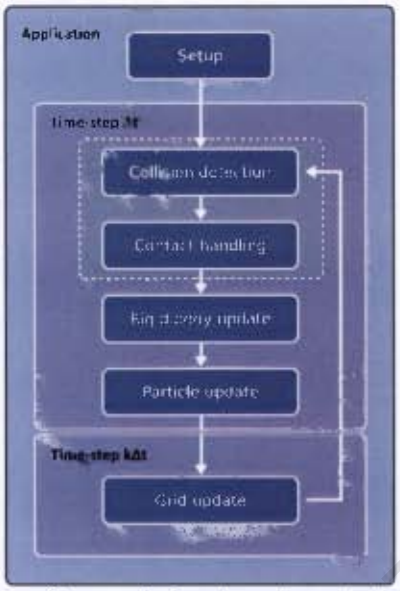


Figure 3.2: Simulation behaviour. Once collisions have been detected, a contact model is applied to colliding granules to find forces acting on each particle. The total force acting on each particle is accumulated and used to calculate the total force acting on each granule. The latter force is integrated to produce new values for granule (rigid body) position and velocity. The updated properties of particles can be derived from the new properties of granules they compose. The collision detection structure (grid) is then rebuilt using particle positions and simulation processing returns to the first step.

Another consequence of the discrete time interval is that particle geometry may interpenetrate, as illustrated in Figure 3.3. In the next section, we shall see the *soft sphere* model gives physical meaning to this overlap, for small Δt.

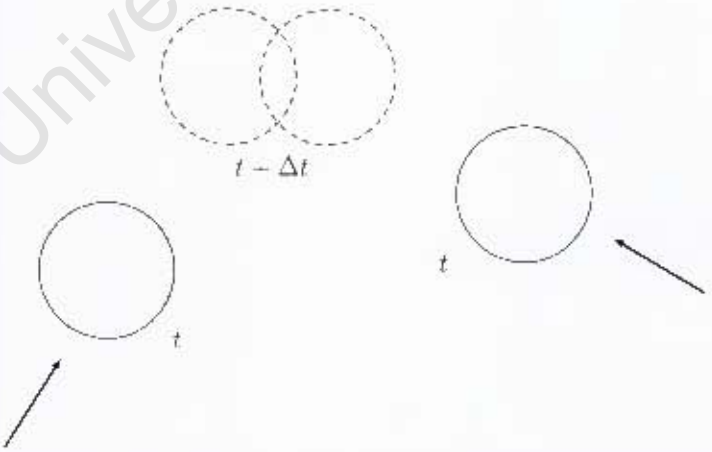


Figure 3.3: Interpenetration resulting from discrete time stepping. Both particles (full circles) are separate at time t, while they interpenetrate (two dashed circles) at time $t + \Delta t$. For a *hard sphere* model, however, the simulation would have to run backwards to time $t' < t + \Delta t$ where contact first occurs, since interpenetration is not allowed.

A related problem occurs when particles reach sufficiently high velocity, allowing them to "pass" through one another during an interval. If this occurs, the simulation observes no interpenetration

at time $t + \Delta t$ and thus finds no collision event. For this reason, the simulation must ensure velocities are within a certain limit, as discussed later in Section 3.3.1.

We now turn to the most important part of a DEM, the modelling of forces between granules. However, as described above, the model is applied between *particles*, before deriving the granule's response based on its particle constituents. Thus, the remainder of this chapter deals with the forces and motion of particles, unless otherwise indicated.

3.2 Intergrain force models

The DEM presented here is sometimes referred to as a soft particle method, as mentioned in the previous section. This means that particles may interpenetrate one another during the course of the simulation. Indeed, this penetration is necessary, since it represents the virtual deformation behaviour of the particles. That is, intergrain contact handling *parameterises* and *estimates* a body's deformation and its consequent response to deformation, by the *depth* of particle overlap. Much like pressing a balloon into the floor, the more force you apply, the more compression and deformation the balloon experiences, and the higher it will rise when released. For colliding soft spheres, the same analogy applies, except the spheres do not change shape, as illustrated in Figure 3.4.

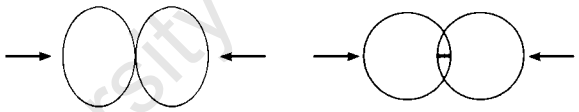


Figure 3.4: Estimating deformation by penetration depth. (a) In reality, colliding spheres experience deformation, the extent of which is determined by their material composition. (b) However, the soft-sphere model does not modify rigid spherical geometry during collision. Instead it estimates the change in spherical shape by the penetration depth (blue line).

More sophisticated force models, however, use other collision parameters as well. These parameters are examined next.

3.2.1 Parametrising collision between two particles

We consider general contact between two soft-sphere particles, each on *separate* granules $\mathbf{G}_i$ for $i \in \{1, 2\}$. These particles have radii R_i, positions $\mathbf{r}_i$, velocities $\mathbf{v}_i$, and angular velocities ω_i for $i \in \{1, 2\}$, as illustrated in Figure 3.5.

As mentioned above, particle deformation is parameterised according to the depth of particle overlap ξ_n, which is given by

$$\xi_n = \max\{0, R_1 + R_2 - \|\mathbf{r}_1 - \mathbf{r}_2\|\}. \tag{3.1}$$

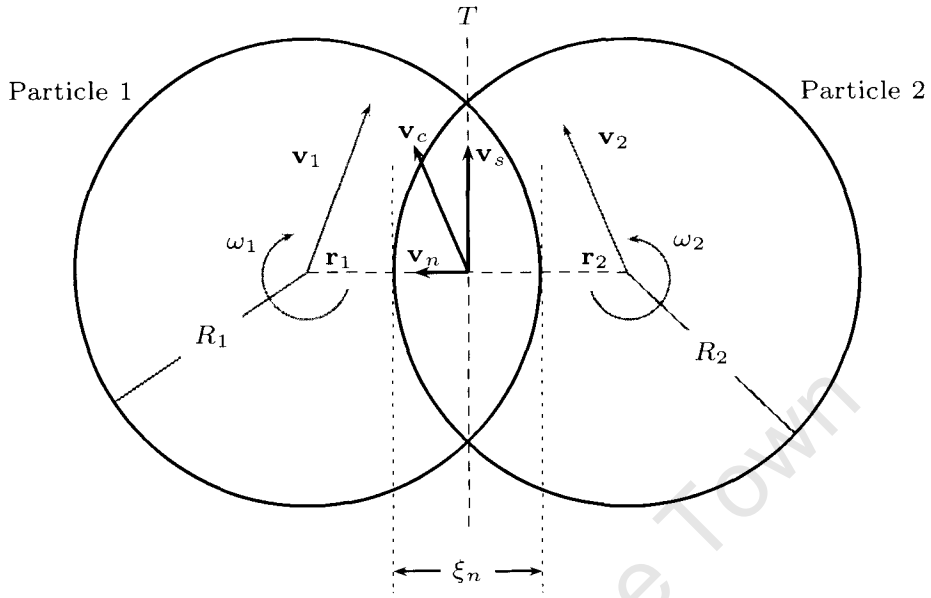


Figure 3.5: Parameterised collision between two soft-sphere particles in a DEM simulation.

When considering the collision from the perspective of particle P_1, we are concerned with forces produced by the action of particle P_2 on P_1. From the remainder of this chapter, when $\mathbf{F}$ is used, we mean the force $\mathbf{F}_{12}$ that P_2 applies to P_1. This comes without a loss of generality, since *Newton's third law of reciprocal action* gives that P_1 exerts an equal and opposite force $\mathbf{F}_{21}$ on P_2, such that $\mathbf{F}_{21} = -\mathbf{F}_{12}$.

It is convenient for contact modelling to decompose forces and velocities with respect to their components in the coordinate frame having one axis in the *normal* direction, that is, from P_2 to P_1 and the other axis in the tangent plane T produced by the intersection of the particle surfaces. The unit vector $\mathbf{n}$ identifies the normal axis and is given by

$$\mathbf{n} = \frac{\mathbf{r}_2 - \mathbf{r}_1}{\|\mathbf{r}_2 - \mathbf{r}_1\|}. \tag{3.2}$$

The relative velocity for the contact point is given by

$$\mathbf{v}_c = (\mathbf{v}_2 - \mathbf{v}_1). \tag{3.3}$$

Thus, we can calculate the relative velocity acting in the normal direction by projecting it onto $\mathbf{n}$ to obtain

$$\mathbf{v}_n = (\mathbf{v}_c \cdot \mathbf{n})\,\mathbf{n}. \tag{3.4}$$

The tangential components are similarly obtained, but with the projection onto the tangential

plane, which is equivalent to removing the normal component of the relative velocity:

$$\mathbf{v}_s = \mathbf{v}_c - \mathbf{v}_n = \mathbf{v}_c - (\mathbf{v}_c \cdot \mathbf{n})\,\mathbf{n}. \tag{3.5}$$

A more convenient notation, used later on, is to let $\dot{\xi}_n = \mathbf{v}_c \cdot \mathbf{n}$, which represents the rate of change of the distance between the particles in the normal direction. Rewriting relative tangential velocity using this notation gives

$$\mathbf{v}_s = \mathbf{v}_c - \dot{\xi}_n \mathbf{n}. \tag{3.6}$$

If the force $\mathbf{F}$ experienced by each particle is not collinear with $\mathbf{n}$, then there is a tangential force component and $\mathbf{v}_s \neq \mathbf{0}$. For this case, we may derive the unit tangential vector $\mathbf{s}$ by

$$\mathbf{s} = \frac{\mathbf{v}_s}{\|\mathbf{v}_s\|}. \tag{3.7}$$

Importantly, we have assumed that all variables, whether vector or scalar, are functions of the simulation time t. We do not show this explicitly to avoid notational clutter. Thus, any differentiation with respect to time, such as for $\dot{\xi}_n$, is well-defined.

The preceding definitions give us enough parameterisation of the collision event to begin discussing DEM contact models. These are methods the simulation may use to calculate the force produced by each intergrain particle-particle collision.

3.2.2 Contact model I: The linear spring-dashpot

The linear spring-dashpot is a widely used and simplistic model for simulating granular material [CS79]. The idea is to insert a normal and a tangential spring at the point of contact between two colliding grains (particles in this work). These "virtual springs" exist throughout the contact phase, producing force until the particles have separated. Each virtual spring is modeled as a linear spring combined with a dashpot, as illustrated in Figure 3.6.

The linear-dashpot model can be formulated as

$$F_n = -k_n \xi_n - \gamma_n \dot{\xi}_n, \tag{3.8}$$

$$\mathbf{F}_s = -k_s \xi_s - \gamma_s \dot{\xi}_s, \tag{3.9}$$

where F_n is the force measured along the normal direction. This value depends on the relative size, shape, and positions of the particles participating in the binary collision. The values k_n and γ_n are the normal stiffness and normal damping coefficients, respectively. They determine the respective amount of force contributed from normal displacement ξ_n and relative normal velocity $\dot{\xi}_n$ at the contact point. ξ_s is the tangential displacement of the contact point and $\dot{\xi}_s$ is the speed of this displacement.

The shear force $\mathbf{F}_s$, tangential to the normal, differs from the normal force in that it is a vector

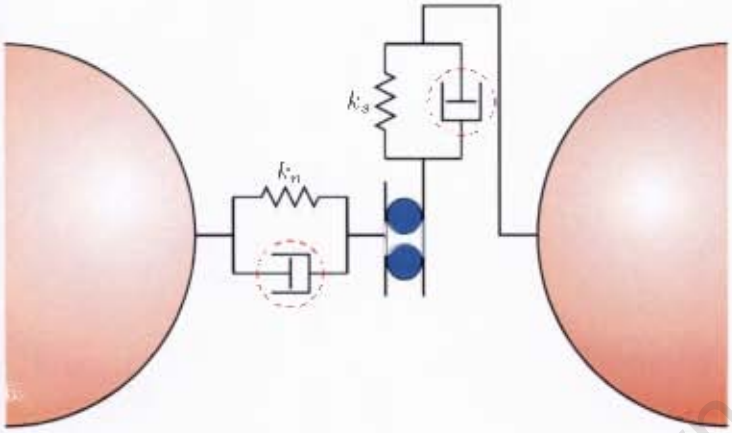


Figure 3.6: Linear spring-dashpot mode for normal and tangential forces. Springs are applied *only* during particle overlap, however, for illustrative purposes, particles are shown here as separated. The normal spring associated with stiffness k_n, when compressed, produces a reactive force that pushes the particles away from each other, until overlap is removed. Similarly, a tangential spring, associated with stiffness k_s, and its matching dashpot, results in a shear force that hinders the particles from sliding over each other. The presence of dashpots (ringed in red) is to limit the amount by which their matching springs can be shortened from compression. While particles may penetrate beyond this minimum distance, the force produced by springs never exceeds the limit imposed by their dashpots. This mitigates the production of unrealistically large forces in the normal direction, when the time step causes large particle overlaps. In the tangential case, once the dashpot limit is reached, particles begin to slide over each other (as suggested by the blue rollers).

rather than scalar quantity. Strictly speaking, $\mathbf{F}_s$ is a 2-vector, since it represents the total force restricted to *two degrees of freedom* in the tangent plane T. Together with F_n, it captures the total force acting in 3D space. However, we express $\mathbf{F}_s$ in 3-vector form above, since it simplifies the calculation of the total force to

$$\mathbf{F} = F_n\mathbf{n} + \mathbf{F}_s. \tag{3.10}$$

In Equation 3.9, the tangential displacement ξ_s is found by integrating the tangential velocity over simulation time as

$$\xi_s = \int \mathbf{v}_s \, dt. \tag{3.11}$$

In the case of two 2D spheres colliding, Equation 3.11 reduces to a scalar integral, which is easily solved. For the 3D case, however, the calculation is more complicated. As mentioned earlier, $\mathbf{F}_s$ has two degrees of freedom in T, which does not preclude T itself changing orientation. It is this change of T over time that makes calculating the ξ_s and $\dot{\xi}_s$ computationally expensive. This is seen by deriving a recurrence relation for $\xi_s(t + \Delta t)$ (see [Faz07]), specifically

$$\xi_s(t + \Delta t) = \xi_s(t) \quad (\xi_s(t) \cdot \mathbf{n}(t + \Delta t))\mathbf{n}(t) \cdot \mathbf{v}_s(t \cdot \Delta t)\Delta t. \tag{3.12}$$

This equation allows us to explicitly derive ξ_s as needed. The intuition behind this equation is that the first and last terms together represent a typical backward Euler integration scheme for updating ξ_s. The second term, however, adapts $\xi_s(t)$ slightly so that it will lie in $T(t + \Delta t)$.

The downside to explicitly calculating ξ_s is the large extra storage, retrieval, and read costs for calculating tangential force. Equation 3.12 needs the new tangential plane orientation given by $\mathbf{n}(t+\Delta t)$ and the new relative tangential velocity $\mathbf{v}_s(t+\Delta t)$, which we expect to calculate anyway at each time step. However, it also requires the previous calculated value for the tangential displacement $\xi_s(t)$ and the previous normal vector $\mathbf{n}(t)$. In addition, we must still compute $\dot{\xi}_s(t+\Delta t)$, which necessitates storing its previous value as well. Of course, these dependencies are for just one of many possible binary collisions the particle may participate in at time $t+\Delta t$.

Calculating tangential force in this way allows us to model both static and sliding friction [CS79, LH93], at a heavy cost to real-time interactivity. An alternative is to estimate static friction behaviour by using only dynamic (sliding) friction. One of the simplest ways of doing this is to use viscous damping. This produces a force in opposition to tangential velocity and proportional to it as well. Formally, it is given by

$$\mathbf{F}_s = -\gamma_s \mathbf{v}_s \tag{3.13}$$

where γ_s is a viscous damping term. Unfortunately, this can give problems because it does not necessarily obey Coulomb's law

$$||\mathbf{F}_s|| \leq \mu ||F_n|| \tag{3.14}$$

which says that the tangential force is bounded by the normal force. Put another way, the harder a solid object is pressed into another, the more force is needed to move that object transversely away from the contact point. This relationship is specified on a material basis by the coefficient of friction μ. Incorporating this dependence on the normal force and the coefficient of friction into Equation 3.13 gives

$$\mathbf{F}_s = -\frac{\mathbf{v}_s}{||\mathbf{v}_s||}\mu F_n = -\mathbf{s}\mu F_n. \tag{3.15}$$

However, this equation presents numerical difficulties with small relative velocities as well as a discontinuity when $\mathbf{v}_s = \mathbf{0}$. The solution to this is to combine Equations 3.13 and 3.15 [SDW96] to get

$$\mathbf{F}_s = \begin{cases} -\mu F_n \mathbf{s} & \text{if } \mu F_n < \gamma_s ||\mathbf{v}_s||, \\ -\gamma_s \mathbf{v}_s & \text{otherwise.} \end{cases} \tag{3.16}$$

This formula is computationally efficient to use, but supports only sliding friction. However, our irregular grain geometry should account for static friction through locking between grain surfaces, provided sliding forces work to prevent motion when the velocity is small. That is, we need the parameters of this equation balanced so that sliding behaviour is accounted for and reasonable agreement is achieved with contact dynamics simulations [LH93, BD98, RSD+97]. Doing this means selecting the "best" value for γ_s, since μ is already known. This choice is a heuristic one, but one where a reasonable estimate can be made [Faz07]. For example, we could consider the maximum velocity attained in free fall over the duration we expect a typical collision to last. If we then consider some particle participating in a collision, when its velocity lies above this value, we make the reasonable assertion that the particle is sliding and should experience sliding contact friction. Below this value we could then assume viscous damping.

For completeness, we mention an alternative approach to modelling tangential friction that differentiates static and sliding friction, unlike Equation 3.16. The distinction between static and sliding cases is usually made by applying a threshold [BYM05], which is usually taken to be the maximum force below which grains remain static.

Thresholding particle forces, however, is a history dependent operation, because we need to know what the forces were previously to check whether these have been exceeded. In terms of the tangential forces, previous work by Lee and Hermann [LH93] used a viscoelastic model that introduced springs when particles first come into contact. They also stored the subsequent displacement of the particles, which was used to viscously damp particle motion for sufficiently small relative tangential velocity. The equation describing this scenario is

$$\mathbf{F}_s = -min(\mu F_n, \gamma_s ||\mathbf{L}||) \frac{\mathbf{L}}{||\mathbf{L}||}, \tag{3.17}$$

where $\mathbf{L}$ represents the total displacement of the spherical particles from their initial point of contact. This procedure is computationally expensive due to its history dependence. The alternative we use is to introduce irregular grain geometry, as described in Section 3.1, where each grain is built of a number of particles. This allows us to simulate the cut-off distance behaviour of Equation 3.17 as geometrical locking between granules. That is, with sufficiently small relative tangential velocity, granules are unlikely to escape tangentially from their contact.

3.2.3 Contact model II: Hertz theory

For the normal forces better predictive models exist as well. Hertz contact forces between spheres is given in elasticity theory [LL86] as

$$F_n = -\gamma_n {\xi_n}^{\frac{1}{2}} \dot{\xi}_n - k_n {\xi_n}^{\frac{3}{2}}. \tag{3.18}$$

The stiffness coefficient k_n is given in terms of the properties of the materials in contact by

$$k_n = \frac{4}{3} E_{\text{eff}} \sqrt{R_{\text{eff}}} \tag{3.19}$$

where E_{eff} and R_{eff} are the effective mass and the effective radius of the particles, respectively. These parameters are given by [BYM05]

$$\frac{1}{R_{\text{eff}}} = \frac{1}{R_1} + \frac{1}{R_2}, \tag{3.20}$$

$$\frac{1}{E_{\text{eff}}} = \frac{1 - {\nu_1}^2}{E_1} + \frac{1 - {\nu_2}^2}{E_2}, \tag{3.21}$$

where E_i and ν_i are the Young's modulus and Poisson ratio for each particle, respectively. Young's modulus is a measure of stiffness for a material, provided we are considering a material that displays the same elastic properties in all orientations. It is determined experimentally from the

slope of a stress-strain curve during tests of tensile behaviour of a material. We use a Young's modulus between 10MPa to 100MPa as is used in the simulations of Bell et al. [BYM05].

The Poisson ratio is a measure of how materials change volume under stress. Most materials resist a change in volume more than they resist a change in shape. The reason for this is that at a molecular level, inter-atomic bonds realign and stretch as molecules move to accommodate a stress. This behaviour usually produces bond lengthening in the stress direction and shortening in other directions. The Poisson ratio for sand is somewhere between 0.2 and 0.45. Note that in our case all parameters have the same values for both particles.

An important issue to address with the Hertz model is that collision duration t_n is now dependent on normal impact velocity [SDW96] (without considering the viscous damping produced by $-k_n {\xi_n}^{\frac{3}{2}}$):

$$t_n = 3.21 \left(\frac{m_{\text{eff}}}{k_n} \right)^{\frac{2}{5}} ||\mathbf{v}_n||^{-\frac{1}{5}}, \tag{3.22}$$

where $\mathbf{v}_n$ here means the initial normal velocity and m_{eff} is the effective mass derived analogously to the effective radius given in Equation 3.20. Achieving sufficient numerical accuracy, therefore, requires that Δt be dependent on the maximum relative velocity that can be expected during the simulation.

In comparison, for the normal force discussed earlier in the linear-dashpot model and presented in equation 3.8, we can solve for the necessary collision duration analytically [SDW96] using some initial conditions, such as $\xi_n(0) = 0$ and $\dot{\xi}_n(0) = \mathbf{v}_n$ to get

$$t_n = \pi \left(\frac{k_n}{m_{\text{eff}}} - (\frac{\gamma_n}{2m_{\text{eff}}})^2 \right)^{-\frac{1}{2}}, \tag{3.23}$$

where k_n and γ_n are the values taken from the linear-dashpot rather than the Hertz model. The previous analytical method also gives the coefficient of normal restitution as

$$e_n = e^{-\frac{\gamma_n}{2m_{\text{eff}}} t_n}, \tag{3.24}$$

which we can solve to find the value for γ_n as

$$\gamma_n = 2m_{\text{eff}} \frac{-ln\, e_n}{t_n}. \tag{3.25}$$

That is, the normal damping depends on the contact duration t_n and the normal restitution coefficient e_n. The latter is a ratio of the normal velocities following and preceding a collision. That is,

$$e_n = \frac{\mathbf{v}_n^f}{\mathbf{v}_n^i}, \qquad e_n \in [0, 1], \tag{3.26}$$

where $\mathbf{v}_n^f$ and $\mathbf{v}_n^i$ are the final and initial normal velocities for a collision. The restitution coefficient e_n is a value that characterises dissipation due to a collision. That is, it describes the loss of kinetic energy, which during a collision is converted to other forms of energy, such as

sound waves or heat.

Restitution is an important property when considering viscoelastic models such as the Hertz model, as given in Equation 3.18, where there is a mixture of viscous damping and elastic behaviour. It has been found through experiment that restitution decreases with increasing impact velocity as given in the relationship $(1 - e_n) \propto v^{\frac{1}{5}}$ [KK87]. The Hertz model has $e_n \to 0$ with increasing $\mathbf{v}_n^i$, agreeing with experiment [SDW96].

3.2.4 Summary

The forces used in our granular model of sand as rigid particles are given by the following equations

$$F_n = -\gamma_n \xi_n{}^{\frac{1}{2}} \dot{\xi}_n - k_n \xi_n{}^{\frac{3}{2}}, \tag{3.27}$$

$$\mathbf{F}_s = -min(\mu F_n, \gamma_s ||\mathbf{v}_s||) \frac{\mathbf{v}_s}{||\mathbf{v}_s||}, \tag{3.28}$$

where the vector version of the normal force is given as $\mathbf{n}F_n$. The stiffness, damping, and normal restitution coefficients are calculated according to Equations 3.19, 3.25, and 3.26, respectively. The duration is derived as in equation 3.23 and sets the upper bound for Δt. In general, we want to have $\Delta t \ll t_n$, but for GPU simulation this is too much to ask.

The values for the parameters mentioned above and their relevance to time-step size are discussed further in Section 7.2.

3.3 Grain dynamics

The general position of an object can be given by taking the object in some coordinate frame, then rotating it about some vector (axis) and translating it relative to the origin of the coordinate frame. However, to simplify the equations of motion, we usually simplify this picture by assuming that a rotational axis goes through the object's center of mass.

The first two sections to follow describe the translational and rotational equations governing granule motion. Importantly, however, rotation must *precede* translation, when applying these equations to grain motion. In addition, later chapters of this thesis present algorithms that need explicit particle information. This information can be derived from the properties of parent granules, as described by the third section below.

3.3.1 Translation

We assume that all forces arising from intergrain and grain-surface contacts, as well as gravity, have been accumulated. For each particle p_i, we have this total force as

$$\mathbf{F}_i = \sum_{j \in \mathbf{C}} \mathbf{F}_{ij} + \sum_{j \in \mathbf{S}} \mathbf{F}_{ij} + \mathbf{F}_{i,\text{gravity}} \tag{3.29}$$

where $\mathbf{C}$ is the set of those particles in contact with particle i. $\mathbf{C}$, however, does not include particles from the same grain as p_i. $\mathbf{S}$ represents the particles on a particle-sampled surface in contact with particle i, such as a wall (see Chapter 6). Once all the $\mathbf{F}_i$ have been computed, then the force acting on grain $\mathbf{G}_k$ comprised of particles P_{i_1}, P_{i_2}, P_{i_3}, and P_{i_4} can be calculated as

$$\mathbf{F}^k = \mathbf{F}_{i_1} + \mathbf{F}_{i_2} + \mathbf{F}_{i_3} + \mathbf{F}_{i_4}. \tag{3.30}$$

We have omitted functional notation for readability, but recall that all forces calculated above are functions of time. In addition, we now omit the superscript from variables and assume we are working at the grain level. Thus, using Newton's equations of motion, we know that

$$\mathbf{F}(t) = \mathbf{a}m = \frac{d\mathbf{v}}{dt}m, \tag{3.31}$$

where $\mathbf{a}$, $\mathbf{v}$, and m are the acceleration, velocity, and mass of grain $\mathbf{G}_k$, respectively. Using a simple first-order integration scheme, such as the forward Euler method [PTV+92], we can derive the grain's new position and velocity as follows

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t, \tag{3.32}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t, \tag{3.33}$$

where $\mathbf{r}$ represents the grain's position as a function of time. This integration scheme is known as an *explicit method*, since $\mathbf{r}(t + \Delta t)$ and $\mathbf{v}(t + \Delta t)$ are defined explicitly in terms of known values. The backward Euler method [PTV+92, BW98], however, is given by

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t + \Delta t)\Delta t, \tag{3.34}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{1}{m}\mathbf{F}(t + \Delta t)\Delta t, \tag{3.35}$$

where the $\mathbf{F}(t + \Delta t)$ entails computing the force based on the grain's position $\mathbf{r}(t + \Delta t)$ and velocity $\mathbf{v}(t + \Delta t)$. It is an *implicit method* in that an equation must be solved in order to find $\mathbf{r}(t + \Delta t)$ and $\mathbf{v}(t + \Delta t)$. The backward Euler has the advantage of being *unconditionally stable*, allowing the use of large simulation time-steps without error magnification or divergence of the simulation into nonphysical behaviour. The advantage of larger time-steps is the simulation proceeds more quickly. However, a common approach is to estimate the implicit function with a Taylor expansion [PTV+92], which requires that a *sparse* linear system be solved at each

step [BW98]. The sparsity of the system permits the use of specifically tailored solvers [BBC+94].

The lightweight forward Euler method, however, as represented by Equations 3.32 and 3.33, is easy to implement. In addition, stability is reasonable as long as the Courant-Friedrichs-Lewy (CFL) condition [CFL67] is satisfied. This condition says that for a discretised space, such as the grid space we use in the simulation, a time-step exceeding some calculable quantity in the simulation should not be taken.

The intuition here is that information must be given enough time to "spread" through the space. That is, we should at least ensure that particles do not attain velocities that move them the breadth of a grid cell in single time step. We can handle this either by using sufficiently small time-steps or by limiting maximum particle velocity [2].

There are other integration schemes including the *leap-frog method* [FLS63] and *velocity Verlet* [Ver67]. The former method has the disadvantage of not keeping positions and velocity in synchronisation over time-steps and therefore an estimate to the grain's velocity is used. Velocity Verlet, however, does explicitly represent velocity and produces more accurate results than the forward Euler method. The disadvantage is that more information is stored and accessed from the previous time-step, which may slow down a GPU-based implementation with large particle counts.

Another variable to consider is the time-step size Δt. which could remain fixed throughout the simulation or vary with each step. Variable time-step algorithms, such as the Runge-Kutta-Fehlberg method [Feh69], where step-size is chosen such that the (local) error produced is maintained below some threshold. However, these high-order methods are computationally demanding. For one simulation step, velocity and position must be solved for different time intervals, which are then appropriately combined to minimize error.

Instead, we use the forward Euler method because it is fast and suited to implementation on the GPU, where recomputing the entire system, even twice, for a single frame of animation can severely hinder real-time interactivity. One problem with this method is that it tends to produce kinetic energy, resulting from its large error term [Faz07]. A collection of sand grains, however, is highly dissipative, which alleviates some of the problem. The expectation is that energy introduced by numerical error is offset by energy dissipation from intergrain and grain-surface collisions.

3.3.2 Rotation

In a similar fashion to the translation section, we now derive the rotational component of granular motion that arises from forces produced with intergrain and grain-surface collisions as well as from gravity. We can express the force produced by particles acting tangentially on grain $\mathbf{G}_k$ at the

[2] We speak of particles instead of grains, since the grid collision detection structure. which effectively discretises the simulation space, works at the particle level, not the grain level.

grain's particle constituent P_i as

$$\mathbf{T}_i = \sum_{j \in \mathbf{C}} (\mathbf{r}_i \times \mathbf{F}_{ij}) + \sum_{j \in \mathbf{S}} (\mathbf{r}_i \times \mathbf{F}_{ij}), \tag{3.36}$$

where $\mathbf{C}$ represents the collection of particles from other grains in contact with particle p_i and $\mathbf{S}$ represents the particles on a particle-sampled surface in contact with particle p_i. The vector $\mathbf{r}_i$ represents the relative position of the particle to the grain's centre of mass, that is, the centre of a regular tetrahedral grain. Similarly to summing translational forces, the torques $\mathbf{T}_i$ acting on the grain at each of its particle constituents are accumulated to produce the total torque $\mathbf{T}^k = \mathbf{T}_{k_1} + \mathbf{T}_{k_2} + \mathbf{T}_{k_3} + \mathbf{T}_{k_4}$ acting on the grain $\mathbf{G}_k$. Again, all vectors are a function of time, though $\mathbf{r}_i$ is of constant length, since grains are assumed to be rigid.

Using Newton's equation for the rotation of each grain, gives

$$\mathbf{I}(t)\omega_k(t) = \mathbf{T}^k(t) \tag{3.37}$$

where $\mathbf{I}$ is a three-by-three matrix called the *mass moment of inertia* for a grain, while ω_k and $\mathbf{T}^k$ are the angular velocity and total torque acting on the grain.

We may think of the matrix $\mathbf{I}$ as the rotational equivalent of the grain's "inertia", which is given by the scalar mass. For rotations in three dimensions, a scalar quantity like mass is generally insufficient, since a rigid body's behaviour depends on its shape and the rotation axis. For example, rolling a metal bar between your hands "feels" easier than spinning it like a baton. The intuition is that $\mathbf{I}$ affects how much angular velocity is produced from an applied torque. Specifically, the matrix $\mathbf{I}$ describes an intrinsic coordinate-independent property of the sand granule that determines the angular velocity produced from a torque $\mathbf{T}^k$ acting on the granule.

However, calculating the mass moment of inertia for platonic solids is a algebraically involved task, though automated calculation exists for triangulated closed surfaces [Kal06]. We merely state that the moment of inertia tensor for a solid regular tetrahedron is [Lou08]

$$\mathbf{I}(0) = \begin{pmatrix} \frac{1}{20}mR^2 & 0 & 0 \\ 0 & \frac{1}{20}mR^2 & 0 \\ 0 & 0 & \frac{1}{20}mR^2 \end{pmatrix} \tag{3.38}$$

where m is the mass and R is the distance from the center of a tetrahedron centered at the origin. Examining equation 3.37 again, we see that to solve for $\omega_k(t)$ means calculating $\mathbf{I}(t)^{-1}$. Fortunately, a simple relation gives the conversion from the local coordinate system of the granule to a world coordinate system as

$$\mathbf{I}(t)^{-1} = \mathbf{R}(t)\mathbf{I}(0)^{-1}\mathbf{R}(t)^T. \tag{3.39}$$

where $\mathbf{R}(t)$ is the rotation matrix describing the grain's orientation at time t. A rotation matrix

is the typical way to represent the orientation of a three-dimensional rigid body. It gives the objects current orientation by specifying the rotation to perform from a fixed reference state, which we store at the beginning of the simulation.

However, there is a large degree of redundancy with rotation matrices. Consider representing a 2D position. We could use a 2-vector, but a 3D or even nD vector works as well, provided we constrain it to move in a two-dimensional plane. In the 3D case this means constraining a 3-vector $\mathbf{v}$ by $\mathbf{v} \cdot \mathbf{n} = k$, where $\mathbf{n}$ is the plane normal and k is a constant. This representation is clearly wasteful, since two degrees of freedom suffices.

Similarly, the three-by-three rotation matrix $\mathbf{R}$ has nine degrees of freedom, when only three are needed: one degree of freedom for each rotation performed around three orthogonal axes. While the matrix could be appropriately constrained, a large amount of numerical error is associated with use of matrices for rotation [Ang06].

The widely used alternative is the quaternion (see Appendix B), which consists of four values that offer four degrees of freedom and one constraint that a quaternion be of unit length. This representation is numerically stable and is easily used in our integration scheme.

The change in a quaternion q due to a constant angular velocity ω (a 3-vector) acting over a short time-step Δt is given by

$$dq = \left[sin\left(\frac{|\omega(t)\Delta t|}{2}\right) \frac{\omega(t)}{|\omega(t)|}, cos\left(\frac{|\omega(t)\Delta t|}{2}\right) \right]. \tag{3.40}$$

Using this we express the quaternion at time $t + \Delta t$ as

$$q(t + \Delta t) = dq \times q(t). \tag{3.41}$$

Now, with rotational and translational motion described for the grain, we turn to describing the derivation of the properties of their particle constituents.

3.3.3 Deriving particle motion

The positions and velocities of particles can be derived from the properties of their parent granules. We begin by indexing grains according to their particle constituents, such that each grain $\mathbf{G}_i$ contains P_i. In addition, we assume that n grains have angular velocities ω_i relative to their centers $\mathbf{g}_i$ for $i \in \{1, 2, \ldots, n\}$. We also assume that each grain has an orientation specified as an orientation matrix R. This matrix is assumed to rotate a vector relative to the grain's original orientation into its relative position for the grain's current orientation, irrespective of grain translation. That is, given the relative positions $\mathbf{o}_j$ for $j \in \{1, 2, 3, 4\}$ of the particles belonging to the grain with center $\mathbf{g}_i$, we can express the absolute positions of particles in space as

$$\mathbf{r}_j = \mathbf{g}_i + R_i \mathbf{o}_j. \tag{3.42}$$

Thus, the velocity of the particle at the collision point is given as $\mathbf{v}_j\prime$

$$\mathbf{v}_j\prime = \mathbf{V}_j + \omega_i \times (\mathbf{p} - \mathbf{g}_i) \tag{3.43}$$

which uses the grain's linear velocity $\mathbf{V}_j$ and the rotational effect of its angular velocity on the contact point $\mathbf{p}$.

3.3.4 Summary

The set of equations governing the motion of grains during the simulation is given by

$$\begin{aligned} \mathbf{a}(t) &= \frac{\mathbf{F}(t)}{m}, & (3.44)\\ \mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \mathbf{a}(t)\Delta t, & (3.45)\\ \mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \mathbf{v}(t)\Delta t, & (3.46) \end{aligned}$$

where we solve for the acceleration given the total translational force $\mathbf{F}$ acting on the grain at time t and then compute the granules velocity and position at time $t + \Delta t$. For the grain's new orientation, we perform the following in order

$$\begin{aligned} \mathbf{R}(t) &\leftarrow q(t), & (3.47)\\ \mathbf{I}(t)^{-1} &= \mathbf{R}(t)\mathbf{I}(0)^{-1}\mathbf{R}(t)^T, & (3.48)\\ \omega(t) &= \mathbf{I}(t)^{-1}\mathbf{T}, & (3.49)\\ q(t+\Delta t) &= q(t) + \frac{1}{2}[\omega(t), 0]q(t)\Delta t. & (3.50) \end{aligned}$$

The first step in Equation 3.47 requires converting the old orientation to a rotation matrix using the conversion scheme given in Appendix B. The result is then used to find the inverse of the moment of inertia tensor at time t given the tensor stored at the start of the simulation. This result allows us to calculate the angular velocity at time t, which is then used to produce the new orientation $q(t + \Delta t)$. The new orientation is then stored for the next iteration. For the purpose of animation, we can rotate the relative positions of particles composing the grain according to this new orientation by applying equation B.8.

3.4 Conclusion

While DEM models exist for simulating granular material behaviours, they are computationally expensive, in part, because of spherical granules needing special static friction handling. In

this chapter, we described a DEM model for multiparticle granules that produces static friction intrinsically through irregular granule geometry. In addition, we presented a force model tailored to simulating sand behaviour through pairwise (particle-particle) interactions between grains.

Chapter 4

GPU-based simulation

In the previous chapter, we described a distinct element model (DEM) for sand, which provides a straightforward and efficient way to simulate physically-based sand behaviour. While other physically-based models have produced comparable sand behaviour, they have failed to realise sand animation in *real-time*[1].

In this chapter, we introduce a GPU-based simulation framework able to produce real-time physically-based sand behaviour for more than 256K granules (one million particles). The implementation of this framework relies heavily on graphics pipeline concepts and general-purpose GPU programming, both of which are reviewed briefly in a background section. The main focus of this chapter, namely GPU-based simulation, then begins with a section describing the *simulation loop*, which manages flow control among simulation components. All subsequent sections, deal individually with each of these components, describing the GPU-based algorithms and implementation details involved. In particular, we look at how to represent sand on the GPU; how to detect collision and calculate forces using this representation; and how to update this representation to reflect grain motion. In addition, nongrain rigid bodies are addressed, thus allowing inclusion of the sampled surfaces produced by the mesh sampler described in Chapter 6. However, a novel mechanism that avoids unnecessary storage of models is also described. This can be used when the surface is simple and fixed in position.

4.1 Background

The specificity of GPU design to graphics applications such as games entails using an architectural model that produces snapshots or images of virtual scenes [Ang06]. These scenes consist of

[1]We interpret real-time to mean speeds above 10-15 iterations per second, which applies to either animation or simulation rate. This we measure through a variable called *frame rate*. Importantly, we can trivially render nothing for each frame and thus measure simulation time alone. If we ignore physical calculation and render particles alone, then the frame time indicates animation speed. In Chapter 12, we quantify this and other framework performance results, as well as explore various limiting factors, such as the influence of granule count on frame rate.

geometric elements, specifically: points, lines, triangles, and quads, which may be positioned and oriented together to form more sophisticated objects, such as surfaces. The appearance and visibility of this geometry depends on properties, such as colour, assigned to them and the positioning of light sources within the scene. The final essential element is the *virtual camera*, which is positioned and oriented within the scene, providing a user with an on-screen rendition of the virtual world.

The production of this final on-screen image, given the scene geometry, can be achieved through an algorithmic approach known as the *rendering pipeline* [Ang06]. The pipeline is divided into several stages [Hai06], which gradually transform geometry step-by-step into image elements. A graphics API such as OpenGL [SA08] implements the full rendering pipeline in software. The components of this the pipeline, their key operations, and the data operated on are illustrated in Figure 4.1.

In the presence of graphics hardware supporting one or more pipeline stages, OpenGL can offload emulation work to hardware instead, thus forgoing slower emulation. However, hardware may also support features not part of the core OpenGL API or the rendering pipeline *per se*. In these special cases, *extensions* to the OpenGL API must be loaded [GLE08] in order to allow an application access to unique hardware features.

In recent years, the full rendering pipeline has been incorporated into the GPU, allowing OpenGL applications to offload the entire graphics workload onto graphics hardware [Ang06]. Much like the parallel behaviour of the CPU pipeline, the hardware-implemented rendering pipeline also benefits from multiple stages that perform different stage operations in parallel [SA08]. This pipeline parallelism is further supported by orthogonal expansion of stages, where a single stage contains multiple processing elements that work in parallel, as illustrated in Figure 4.2.

The arrival of programmable stages, supporting multiple pixel and triangle operations simultaneously, has allowed a *stream processing* model to be applied to the GPU [OLG+07]. This is due to the SIMD (single-instruction multiple-data) approach taken by the modern GPU to transforming vertices and shading pixels. This results in individual processing and floating-point units on the GPU running the same set of instructions, each acting on the same dataset (vertex positions, for example), but at different positions in the data. Moreover, these instructions are modifiable by uploading user-defined code to the GPU.

The stream processing view of the GPU is used in this chapter to perform parallel physics calculations, detect collisions, and produce granule motion on the GPU. However, this entails meeting three underlying demands inherent in a physically-based simulation. Physical parameters need to be stored, calculated, and updated. The GPU supports these requirements through the use of *textures* and *vertex buffer objects (VBO)* for storage; *shaders* for calculation; and *rendering with framebuffer objects (FBO)* for updates. These structures and their behaviours are explored in further detail below.

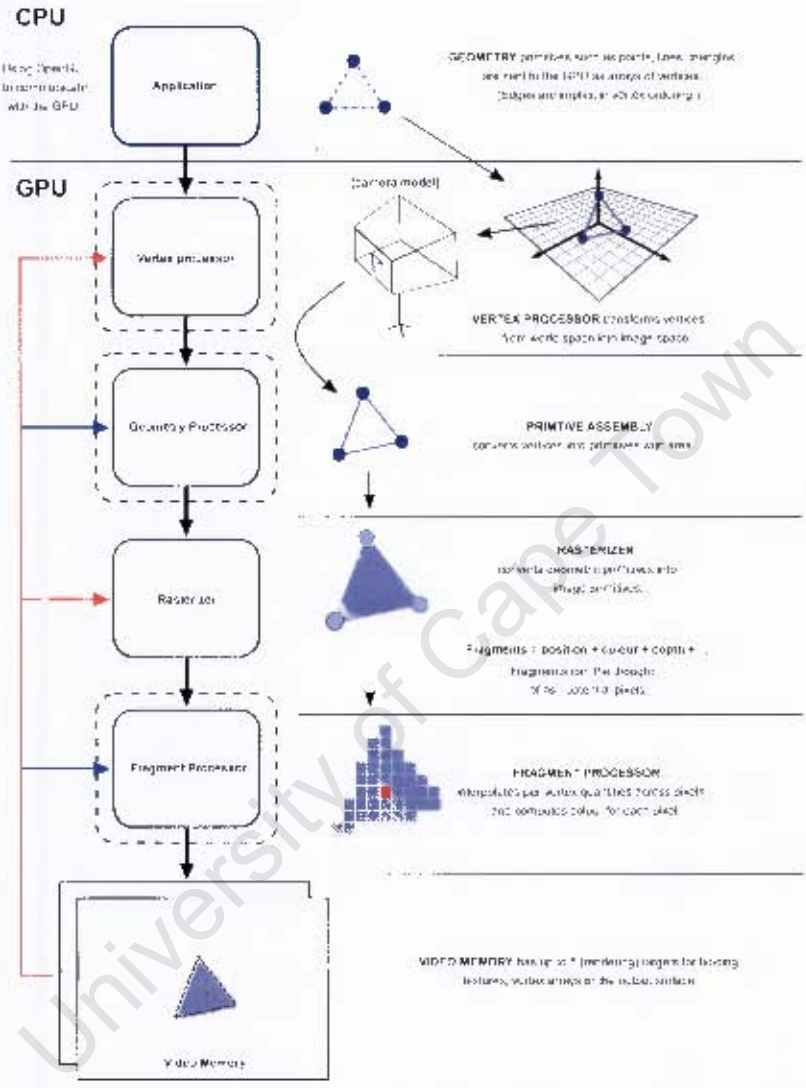


Figure 4.1: The rendering pipeline. An OpenGL application initiates rendering by passing *vertices* to the pipeline. Vertices may either represent individual points or geometric primitives, where geometry requires vertex ordering. That is, two ordered vertices implies a *line*, while three ordered vertices implies a *triangle*. Quads and other more complex surfaces, in turn, are represented by collections of triangles that share edges or vertices. All geometry, however, is ultimately decomposed by later pipeline stages into *fragments*, or potential picture elements. Each of these picture elements or *pixels*, represents one of the indivisible colour elements that compose the final image in video memory. The objects in video memory that store these images are called *framebuffers*. Multiple framebuffers might exist when various effects are desired. However, one of these usually holds the final image sent to the user's screen. In the past, framebuffers existed in memory accessible only to the vertex and rasterisation stages, as indicated by the red paths. However, two other stages now have the ability to access this information, as indicated by the blue extensions. Stages ringed by a dashed line are *programmable*, a concept explained later in the chapter.

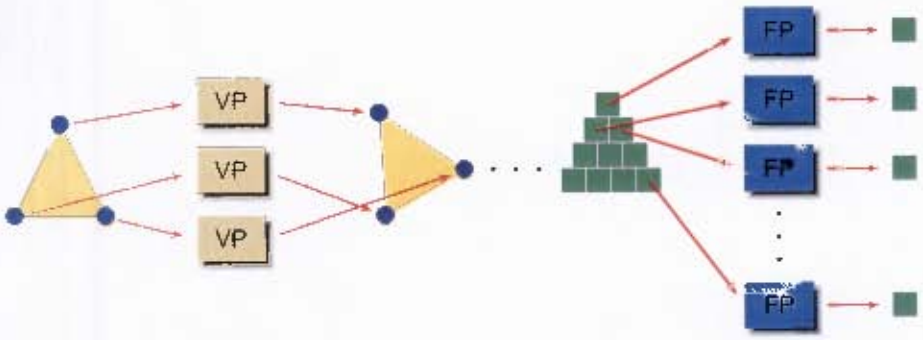


Figure 4.2: Pipeline parallelism and orthogonal stage parallelism. The vertex stage may have multiple vertex processors (VP) that perform the *same* transformation on *all* vertices simultaneously and independently. Subsequent geometry processing, therefore, has access to the entire primitive (the rotated yellow triangle) after one vertex processing step, rather than three steps. Similarly, after rasterisation, multiple fragments generated from another triangle (fragmented green triangle) are processed simultaneously and independently by corresponding fragment processors (FP), which also independently write their shaded pixels out to the framebuffer. Pipeline parallelism is achieved by synchronous operation of the vertex and fragment stages. That is, while the vertices from one triangle are being transformed, fragments from another triangle are being shaded.

4.1.1 Storage arrays

Textures provide a way for information to be stored on the GPU, allowing GPU-based programs rapid access to physical parameters, without suffering the time cost of CPU-to-GPU transfer. The texture, however, is not alone in providing GPU-based storage. The *vertex buffer object* (VBO) is another GPU-memory resident data structure, which has unique advantages and certain shortcomings compared to textures.

VBO

The vertex buffer object (VBO) [HCA06] is a well-known and widely supported OpenGL extension, which has since become a core part of the OpenGL API [SA08]. It provides a container type for storing vertex data in a graphics card's high-performance memory.

We may think of this container simply as an a one-dimensional array of values. When specifying a VBO, the programmer must indicate the type of data it stores, such as floating-point or integer values. During rendering, however, the arrangement of this data is needed as well: be it a list of points, vertices, or triangles. The concept is illustrated in Figure 4.3

The VBO extension also exposes OpenGL functions for transferring this data between memory spaces. That is, by storing vertex data on the graphics card, rather than in the host's machine memory, VBOs avoid repeatedly having to copy this data between the two.

Unfortunately, VBOs lack read caching, since the pipeline expects the vertices to be sent as a contiguous list. In addition, VBOs cannot serve as rendering targets, which precludes data

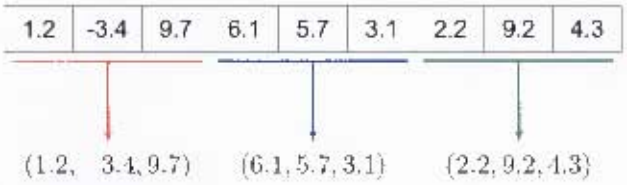


Figure 4.3: Vertex buffer object. The array (series of boxes) holds the floating-point values, which are copied by the OpenGL application to the GPU. During rendering, this data is sent into the rendering pipeline, using a left-to-right ordering. The pipeline must be informed of what the data represents. Here we have assumed a list of vertices is sent. However, this could equally be interpreted as a triangle, specified by its three 3-vector vertices.

storage directly on the GPU.[2] The absence of these properties has meant looking to textures as a means to store, read, and update data directly on the GPU.

Textures

Textures are a key data structure in GPU programming and preferred in this thesis over the VBO. Both these data structures store similar data in GPU memory. The important difference is how, when, and where this data enters the rendering pipeline.

Textures are array-like storage areas which allow access to the GPU memory assigned to them. They consist of one, two, or three-dimensional arrays of texture elements, or *texels* [Ang06]. These elements come in vector formats with a choice one to four floating-point values per texel. However, the choice of texel format is uniform across an entire texture, so that we may specify a 3D texture with 4 vector texels or a 2D texture of 1-vector texels. Figure 4.4 illustrates the case of a five-by-five texture with 4-vector texels.

When creating a texture, an application must specify width, height, and depth for the 2D and 3D cases. In the past, OpenGL and hardware limited texture dimensions to powers of two, enabling more efficient texture filtering and packing [MB05]. However, the trend toward larger texture memory on the GPU [NVI08a], as well as the absence of special filtering requirements in many applications, has ushered in the *rectangular texture* [KS05]. This widely supported OpenGL extension permits arbitrary texture dimensions within hardware-supported limits.

Unlike vertices, which were are ordered and issued to the pipeline, texel data is read through calls to a texture unit (typically by shaders running in the pipeline) [Owe07]. The advantage to a texture storage approach is the presence of *texture caching*, which increases texel read rate for shaders that employ a spatially local access pattern [PF05]. The simulation described in this thesis performs texture reads of *all* the data in a grain or particle texture, at once. It also performs

[2] The situation has been ameliorated recently by a unified approach to GPU architecture. In particular, a recent OpenGL extension, called the texture buffer object [Bro08], implements a VBO attachment as a data source for a texture unit, thus allowing shaders cached access to vertex data. Furthermore, the new geometry processing stage adds *stream out* facilities to the GPU. An OpenGL extension, called transform feedback [LHW08], supports this behaviour by allowing a VBO to be bound as a stream output target.

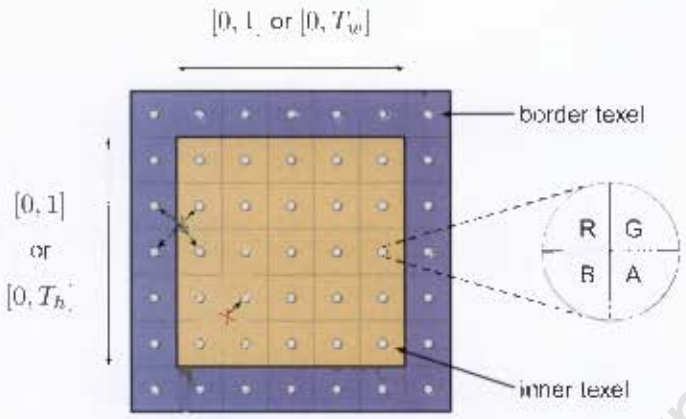


Figure 4.4: Five-by-five texture with 4-vector texels. The texture surface (tan region) is represented by a five-by-five block of texels (small tan squares). Texels store their data at their centers (white dots), rather than over their entire surface. The texels above store a 4-vector of (R,G,B,A) (red, green, blue, and alpha) at their centres. Since textures are surfaces, however, each texture must have a *filtering* mechanism defined for reading from an off-centre position. *Nearest-neighbour* filtering, for example, given an arbitrary point (red cross), returns the value of the closest texel centre. Alternatively, filtering by *interpolation* entails the use of surrounding texel centres to calculate the value at an arbitrary point (green cross). Textures, therefore, include a boundary region of texels surrounding the texture surface. These are used when interpolation is necessary or texture access is clamped to the boundary. Clamping occurs when a position is read outside of the normalised region $[0, 1] \times [0, 1]$ for standard textures and the unnormalised region $[0, T_w] \times [0, T_h]$ for *rectangular textures*.

spatially-local reads of the grid collision detection structure. Thus, a texturing approach is an attractive way to increase throughput per read operation. Another important feature of textures is that they may serve both as a data source *and* target, when an FBO and shader are used, as discussed in Section 4.1.3.

4.1.2 Parallel calculation

Modern GPUs have application-programmable vertex, primitive, and fragment pipeline stages, which allow programmers to replace default pipeline behaviours with their own methods. This is supported now by various C-like programming languages, including the OpenGL Shader Language (GLSL) [KBR08] and NVIDIA's Cg [FK03], which give high-level programmable access to the underlying pipeline functionality. Most newer languages are predated by various assembly languages, which continue to provide low-level access to the GPU [L+06, B+07]. They are useful when hardware specific optimisations are needed. In general, however, better design is achieved with high-level languages, which provide abstraction away from the hardware and allow programmers to focus on algorithmic details instead [NV106, NV108b].

High-level graphics pipeline abstractions

High-level graphics languages, however, still operate within the graphics context, requiring that programmers use rendering idioms to produce general-purpose computation on the GPU. *Metaprogramming* languages, however, support further abstraction away from graphics concepts, presenting programmers with a general-purpose environment for stream processing. Various metaprogramming languages, such as Brook [BFH+04] and Sh [MQP02, MTP+04], have been developed with these goals in mind. Importantly, metaprogramming languages ultimately compile to rendering code that executes within the graphics context.

Recently, certain high-end GPUs [NVI08a] have also acquired *architectural* support for stream processing, called the Compute Unified Device Architecture (CUDA) [CUD08]. Programs produced using the accompanying API compile to machine code that executes within a nongraphics context on the GPU. In particular, the GPU is seen entirely as a concurrent thread-based processor [Hal08].

The advantage of using a metaprogramming language or an architecturally supported API, is that graphics resources are virtualised, instead of being managed by the programmer [BFH+04, Hal08]. In addition, a metaprogramming language is subject to logical analysis by the compiler, whereas before it was the programmer's responsibility to ensure logical behaviour, while using the rendering pipeline in a general-purpose but nonlogical way. Architectural support, however, obviates having to understand graphics pipeline features, instead requiring an understanding of a purely thread-based environment.

The disadvantage of a metaprogamming approach is that programmers no longer explicitly control resources, where particular applications may benefit in performance from specifically tailored resource management. In addition, desired results are often achieved by nonlogical rendering schemes. An example of this is given in Section 4.4.3, where the simulation stores particle IDs through non-trivial use of depth and stencil buffers over multiple rendering passes.

This thesis does not use metaprogramming or architecturally-supported APIs. In particular, multiple pass rendering mentioned above, requires *atomic operations*, since multiple execution contexts read from and write to the same memory location. While supported transparently in the graphics context through buffers, atomic operations have only recently acquired general-purpose architectural support [SA07, NBG+08]. In addition, since our purpose is to animate sand, working within the rendering pipeline presents more opportunities to couple simulation and animation.

Thus, the simulation described in this thesis accesses GPU parallel processing units through programs that are written in a *shader programming language*. These programs use the rendering pipeline both for physics-based calculation and high-fidelity rendering, making them key to achieving real-time simulation.

Shader programming

Many high-level shader languages such as GLSL have syntax that resembles standard C code, including C-like functions, data types and control-flow constructions. Similarly, main functions are mandatory, but in this case one needs to be defined for each shader stage replaced. A *shader*, therefore, refers to a single main function and its associated supporting (non-main) functions. Usually the term is qualified by the stage name the code intends to replace. That is, shader code must adhere to the particular set of constraints, such as what it can read and what it must write, predicated on whether it expects to act as a vertex, geometry, or fragment shader [KBR08].

In the interest of alleviating clutter, actual OpenGL code for setting up and using shaders is avoided. There are many widely available sources that give specific details on how this achieved [WND97, Ros06]. Still others provide details on shader-based algorithms [Fer04, PF05, Ngu08]. The pseudocode used in this thesis assumes an underlying OpenGL and GLSL implementation. However, other shader programming languages, such as Cg [FK03], which support OpenGL interoperability and have similar functionality to GLSL, can be used instead.

The compiled shader code runs inside the rendering pipeline, accepts vertices or fragments, and outputs modified vertices or pixels, respectively.

4.1.3 Parallel updates

The rendering pipeline rasterizes transformed geometry to produce fragments. Those fragments that pass various checks, such as depth testing, go on to form a 2D array of pixels in a memory area called the *framebuffer*. The framebuffer contains a collection of 2D arrays that separately accept pixel colour, depth, or other associated data. Thus, apart from the colour buffer type, whose image usually appears on screen, several other logical buffer types exist, including the depth and stencil buffers. The graphics driver software could update a user's screen with the contents of any one of these buffers. However, the OS-provided framebuffer, a colour buffer, is the default storage area for the final on-screen image.

A recent OpenGL extension called the *framebuffer object* (FBO) [JS08] provides a way for a user to create extra framebuffers. These serve as *nondisplayable* render targets for OpenGL pipeline output. An FBO instance has attachment points for colour, depth, and stencil buffer types. Importantly, these attachment points accept either a texture or a default storage type called a *renderbuffer*. Both of these can be used as underlying storage for the FBO's colour, depth, or stencil buffers. Textures, however, can be read by shader programs (user-written GPU code) at a later time, while renderbuffers are nonreadable in this context.

An OpenGL program using one texture as a source and another as a target bound to an FBO is thus able to emulate a read-compute-write paradigm inside the rendering pipeline. For computations that only depend on previously updated information, two textures are enough. In particular, an updated target texture serves as a new source texture, while the previous source

texture, now storing defunct information, is overwritten with new updates.

Texture to texture transfer

Achieving texture-to-texture transfer within the rendering pipeline, however, requires a mechanism for mapping source and target texels to one another. The most straightforward approach is to map texel centres to corresponding centres. Achieving this in the rendering pipeline requires using an *orthogonal projection* during rendering, as illustrated in Figure 4.5.

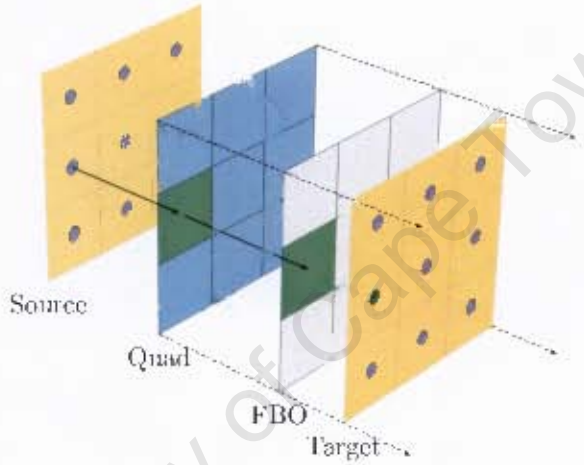


Figure 4.5: One-to-one quad-texture mapping for texture updates. Prior to rendering, the quad is assigned texture coordinates in a one-to-one fashion, so that texel centres from the source texture will match equally to fragments (blue squares), which are produced by rasterisation of the quad during rendering. These fragments initiate per-fragment shaders (that is, user-written code), which has access to the fragment's assigned texture coordinate and texture source. Once relevant texel data is retrieved and computation finishes, each fragment is written as a pixel to a framebuffer or an FBO. The use of an *orthogonal projection* (indicated by dashed lines) leads to a one-to-one correspondence between fragments and the pixels they produce. In the case of an FBO, an underlying texture is used as the store for pixel data. While pixels occupy *surface area* (white squares), they store a uniform value across that surface. Thus, the FBO maps the data represented at the pixel to corresponding texel centres on a target texture. Due to orthogonal projection, both source and target textures have the same dimensions.

Multiple updates simultaneously

Often a shader-based calculation must retrieve multiple results from previous calculations, which are stored on different texture surfaces. While shader programs and hardware allow reading multiple times from *different* textures, previous hardware allowed writing to only a *single* output texture. Thus an application would have to resend all the geometry multiple times to store results for each surface. This iterated process is not only slow, but forces the CPU to synchronise its surface writing to the application process running on the CPU, reducing GPU parallelism.

However, modern graphics cards [NVI07] allow *multiple render targets* in a programmable fragment shader used with an FBO. Thus, with multiple render target support, a single pixel can shade matching positions on multiple FBO-attached target surfaces with only a single pass of geometry through the pipeline, as illustrated in Figure 4.6.

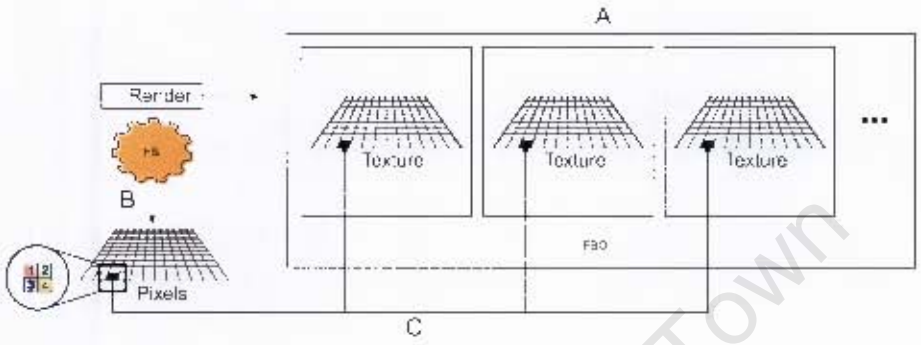


Figure 4.6: We may have up to 8 textures bound to an (a) FBO instance. The (b) pixels produced by a fragment shader (FS) can consist of up to 4 values; that is, a 4-vector of red, green, blue, and alpha (RGBA). For each fragment input, the shader can produce up to 8 4-vectors, (c) which it can store in the separate textures bound to the FBO.

Selective updates

In some cases, we may wish to control where texel updates occur, such as when selectively storing values that meet some criteria. That is, we need a mechanism that *rejects* pixel production from a matching fragment instance. In the graphics context, the *depth* and *stencil* renderbuffers, like their non-FBO counterparts, enable or disable drawing to a target colour surface on a per-pixel basis [MB05, Ang06]. This also implies that depth and stencil buffers have the same dimensions as the target colour surface. In addition, depth and stencil buffers store only a single value for each matching pixel element.

A ***depth buffer*** chooses whether to allow a fragment shader to write (draw) to a surface pixel based on a fragment's depth value, which is derived from rasterisation of a geometric primitive. In particular, the fragment's depth and the depth of the pixel stored in the depth buffer must satisfy a prespecified binary relationship, which is chosen through an OpenGL command. The two used in this thesis with FBOs are the *less-than* and *greater-than* functions. The values held in the depth buffer are in the range $[0, 1]$ by default. However, unnormalised values can be used through an OpenGL extension [BS08] on supporting hardware.

The ***stencil buffer***, like the depth buffer, is used to conditionally select whether fragments are written to a surface. However, instead of floating-point values, the stencil buffer stores nonnegative integer values in the range $[0, 2^n - 1]$ where n is the number of bits per pixel [WND97]. A *stencil test* involves the comparison of this nonnegative value to a reference value set by OpenGL, using a *less-than* or *greater-than* function, among others. An example of depth and

stencil interaction is illustrated in Figure 4.7.

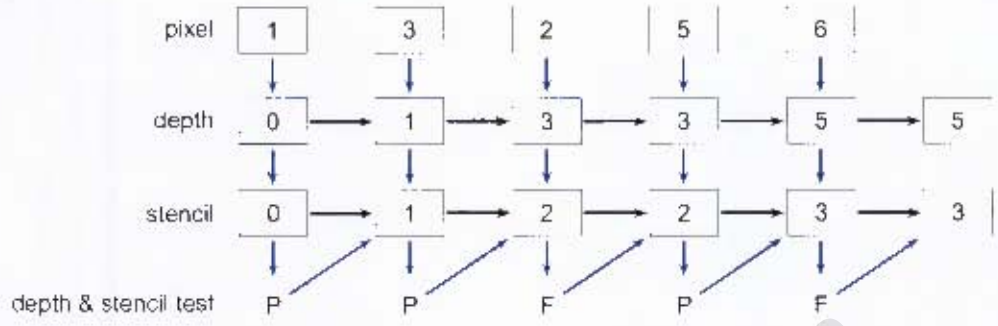


Figure 4.7: Depth and stencil buffer example. The top row represent multiple fragment values that are written to the *same* pixel position in the framebuffer. The middle and bottom rows represent the framebuffer depth and stencil value, respectively, for that pixel position. We assume the depth test is set to *greater-than*, that is, to accept only fragments of depth greater than the stored depth value. Unlike depth, however the stencil test uses a *reference stencil value* rather than the fragment information. In this example we set the stencil reference to 3 and the stencil test to *greater-than*. This means only those fragments where the reference value is greater than the fragment's matching stencil value are accepted. Further, a stencil value can be updated based on the the passing or failing of the depth and stencil tests. In this example, a successful depth and stencil test for a fragment leads to an *increment* of the matching stencil value, while any other failure-pass combination results in no change to the stencil.

The importance of depth and stencil buffers becomes apparent in Section 4.4.3, where fragments writing to the same pixel position must be ordered. Such an ordering is necessary since user-code running on the GPU has specifically altered rendering leading to coincident pixel writes.

The discussions provided above provide enough background to begin discussing the GPU-based simulation itself.

4.2 Simulation loop

The simulation can be decomposed into a series of events, or *simulation loop*, namely, setting up initial material state as well as handling and observing state changes in the course of simulation. The handling of state changes may be further decomposed into detecting collisions among granules, producing forces from these collisions, and updating granule motion. Observation of updated state entails visualising this motion. The latter, however, needs special treatment that is left to Chapter 5. The flow of information for the GPU-based simulation is illustrated in Figure 4.8.

This is now used as the basis for implementing a GPU-based algorithm, as described in the following sections.

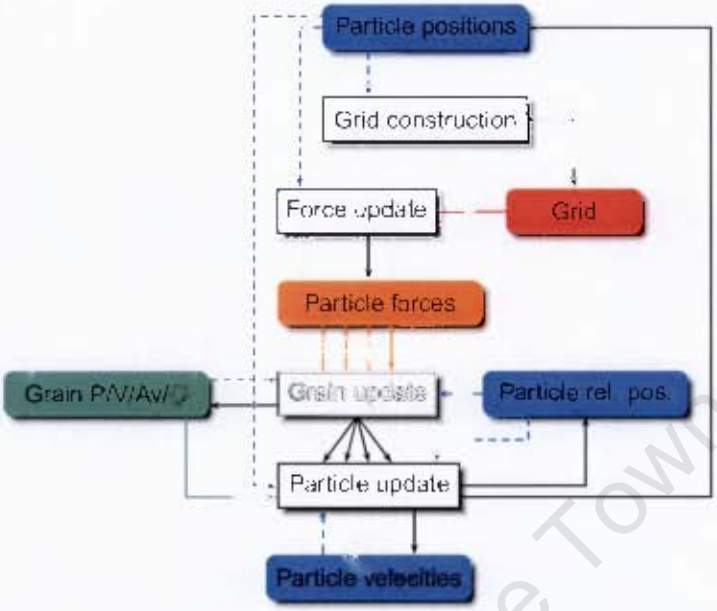


Figure 4.8: GPU-based DEM algorithm. Coloured boxes represent GPU-based storage that holds old and updated values on separate textures. Stored values include grain position (P), velocity (V), angular velocity (Av), and orientation (O), as well as particle positions, velocities, and relative positions (rel. pos.). These stores are assumed to be initialised before the start of the simulation. Dotted coloured lines transmit *old* values, while solid lines carry values updated during the *current* time-step. For solid black lines, however, the source of this data is shader-based calculation, rather than textures.

4.3 Distinct geometry storage

The simulation setup involves loading granule data from a file, which includes the world position of granules; the relative positions of the particles composing each granule; the orientations of granules; as well as their linear and angular momenta. Note that relative particle positioning, that is, relative to granule centres, implicitly represents granule orientation. Nevertheless, we store both for time efficiency purposes, as detailed later. This file may also contain data for nongranule rigid bodies. We store these in the same way as for granules, except nongranule bodies have an associated particle count, whereas granules are assumed to always consist of four particles.

Performing real-time rigid body simulation on a GPU requires that most of the simulation data exist in GPU memory. Specifically, the shader code running on the GPU needs rapid access to granule position and velocity information, amongst other data. Two high-performance data structures that meet this criteria are textures and VBOs. Section 4.1.1 discusses these structures and the reasons for using textures instead of VBOs for grain, particle, and rigid body storage.

4.3.1 Granule and particle storage

The natural choice of grain representation is to use a single spherical particle for each grain. For DEM simulations, which typically use spherical particles, theory suggests that tangential forces dominate in heap formation [WRD98]. The tangential forces meant here are static and dynamic friction. As our discussion of Equation 3.17 at the end of Section 3.2.2 pointed out, a spherical particle model of static friction exacts a heavy computational cost that can hurt real-time interactivity.

Instead, we use rigid particle arrangements to form sand granules, as discussed in Section 3.1 and depicted in Figure 3.1. The force model used for particle-based contact among grains is summarised in Section 3.2.4. It includes both repulsion forces that act normal to particle contacts, keeping grains separate from one another and tangential forces that slow down tangential velocities. In the latter case, the geometry approximates part of the static friction by preventing grain boundaries from shifting when subject to slow tangential velocities.

In addition, the *tetrahedral* grain model is an efficient representation, since it is the smallest arrangement of two or more particles where the moment of inertia for each primary axis is equal. This means the moment of inertia for a tetrahedral grain is simply a scalar times an identity matrix. Thus, we can forgo the matrix-vector multiplication used in Equation 3.37 and perform cheaper scalar-vector multiplication instead.

Now that we know how grains are represented, we want to make their positions, momentums, and their other physical properties available to code running on the GPU. In particular, this data must be rapidly accessible, which means storing it in GPU memory. We use the texture as a GPU memory resident data store that provides both rapid read and write access to its contents (see Section 4.1.1).

Granules

Grain storages requires that we assign to each granule a unique identification number $g_i = 0, 1, ..., g_{\max}$. Each g_i is mapped one-to-one to a unique two-dimensional texture coordinate as shown in Figure 4.9. The equation used to move back and forth between the one-dimensional and two-dimensional (texture surface) orderings are

$$(x, y) = (id - w \cdot \lfloor \frac{id}{w} \rfloor, \lfloor \frac{id}{w} \rfloor), \tag{4.1}$$

$$id = x + y \cdot w \tag{4.2}$$

where (x, y) and w are the texture coordinate and width, respectively, and id holds the value g_i. The floor function "$\lfloor\ \rfloor$" is defined by $\lfloor x \rfloor = \max \{n \in \mathbb{Z} \mid n \leq x\}$ for a floating-point value x. That is, it produces an integer from a floating-point value. As we shall see later, fragment programs are used to manipulate and update both granule and particle data stored in textures, where each program instance manipulates the data of a single granule or particle. Thus, the previous equation

gives a fragment program a means to a calculate the id of its associated particle or granule.

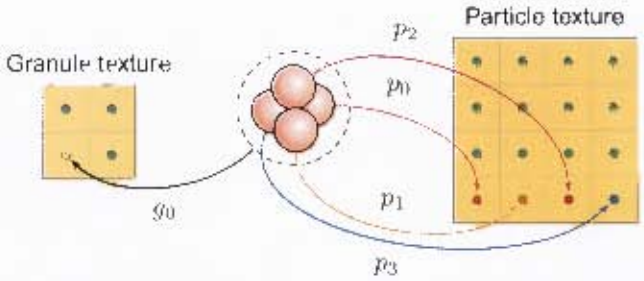


Figure 4.9: Mapping granules and particle properties to texels. In the absence of nongranule rigid bodies, every quartet of particles, starting from the bottom left of the texture and moving right, is associated with a single granule. Thus, granule properties, such as velocity, are ordered in the same way as the matching properties of their particle constituents.

Thus, for each granular property (namely, position, linear velocity, angular velocity, and orientation) we create a texture. For each grain, we store its individual properties to their matching textures, but always at the same two-dimensional coordinate.

In Section 4.1.1, a texture is described as a two-dimensional array of 4 vectors, which nominally stores a colour in the form $(red, green, blue, alpha)$, usually written (r, g, b, a). The latter is unimportant to us, save for the fact that each vector element holds a single-precision floating point value that a shader instance running on the GPU can either read from or write to depending on how rendering is set up. In a shader programming language, such as GLSL [KBR08], we may refer to a stored texture value as (x, y, z, w) instead of (r, g, b, a), a practice we follow for the remainder of this chapter.

Particles

Particle properties are stored and accessed in much the same way as for grains. Each particle is assigned a unique identification $p_i = 0, 1, 2, ..., p_{max}$, where each p_i is mapped to a unique two-dimensional texture coordinate. However, as shown in Figure 4.9, particles have an additional ordering constraint that requires particles from the same grain to have contiguous ids. The four-particle ordering itself, however, is unimportant. That is, grain properties do not depend on the order in which particles properties, such as force, are accumulated, only that they are accumulated from the correct four particles.

The alternative is to "hard code" relative positions and not load this data from file. However, runtime relative positions allow the simulation to support extensions to quintet granules or other particle counts and arrangements, that is, without having to modify the simulation algorithm itself.

The mapping between particles and their texture is specified by equations 4.1 and 4.2 used in the grain-texture mapping, except that particle texture coordinates and width are used instead, so that id holds the value p_i.

As with grains, particle textures[3] are needed for position, linear velocity, relative position, and force, with individual texels matched to individual particles.

For the case where a particle property is used to update a grain property or vice versa, a mapping between particle and grain ids is needed. Since each granule is associated with a quartet of particles, as illustrated in Figure 4.9, the transformation between particle and parent grain ids involves a simple and fast calculation, namely

$$p_{i,j} = 4g_i + j \quad \text{for} \quad j = 0, 1, 2, 3 \ , \tag{4.3}$$

$$g_i = \lfloor \frac{p_k}{4} \rfloor, \tag{4.4}$$

where $p_{i,j}$ is the one of four particles composing grain g_i and p_k is a particle that is known *a priori* to belong to a grain (opposed to a nongrain rigid body). The main function of equation 4.4 is to check whether two particles are from the same grain, since collision checking and force calculation are not carried out for same-grain particles. Equation 4.3 is used primarily to locate the forces acting on grain particles. These forces can then be accumulated into a total force acting on the grain.

However, both previous equations can be avoided by taking advantage of the fourth coordinate of the 4-vector properties of granules and particles. For each granule, we may store its id in the w-coordinate of the granule's associated position texel. In the granule's linear velocity texel, however, we use the fourth coordinate to store the id of the first particle in the quartet constituting the granule. In this way, no extra calculation is needed, as both granule position and linear velocity are accessed for particle updates, making the id values automatically available (see Section 4.6). Similarly, a particle position and velocity can store the ids of the particle and its associated granule, respectively. Figure 4.10 illustrates this idea.[4]

Id conflicts

A problem may arise when storing particle and rigid-body ids from the interval $[0, n-1]$ instead of $[1, n]$. If the particle or rigid-body count is less than the number of texels in matching property textures, then some texels are not used. This thesis assumes that all unused texels are set to a value of $(0, 0, 0, 0)$. Thus, in the case of a velocity texture, for example, the values stored by an unused texel cannot be distinguished from a particle or rigid body with id 0 and velocity $\mathbf{0}$.

[3]Two of the first three textures are strictly unnecessary, since the values they store are derivable from the properties of their parent grains. That is, given relative particle positions and absolute grain positions, absolute particle positions can be found. Alternatively, since relative positions are needed for force calculation on the grain, both grain and particle absolute positions are needed. In addition, since either a relative or absolute position texture is used, a separate (CPU-bound) rendering pass must be performed to update their values. Given the high texture fill rates of modern GPUs [NVI08a], writing texels to multiple surfaces incurs little overhead versus single surface writes. Thus, all particle properties can be updated at the same time. Finally, particle velocity requires reading all granule properties or simply a single value stored at the same time particle positions were stored. The latter is the preferred approach in this thesis.

[4]Note that each texel element (x, y, z, w) is a 32 bit floating-point value. Since the id value is stored in the w-coordinate, the space of possible ids depends the IEEE 754 floating-point standard, as supported by the GPU [IEE85]. Fortunately, this guarantees an exact representation of any integer with an absolute value less than or equal to 2^{24}, which clearly provides enough id-space for our needs.

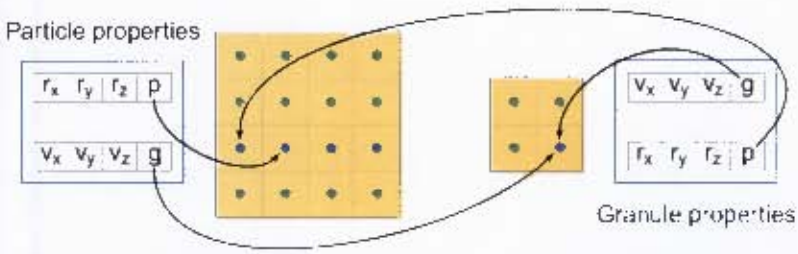


Figure 4.10: Explicit storage of particle-granule mappings. On the left, we see how the ids stored in a particle's position and velocity (previously read from a texture or received as input) can be used to access related properties, that is, from the *same* texel position on another particle texture. The associated granule id g can be used to find parent granule properties as well. An analogous idea applies to a given granule position and velocity (illustrated on the right), except that the particle id p is for the *first* particle constituent associated with the granule id g.

The solution used in this thesis is to increment particle and rigid-body ids by one, before they are stored to texture, while decrementing ids by one, when they are read from the texture at a later stage. We shall refer to this as the *increment-store read-decrement* (ISRD) numbering. Alternatively, the simulation could store and manipulate ids exclusively in the interval $[1, n]$. However, equations such as 4.3 and 4.4 rely on particle and grains with ids starting at 0. Adjusting these equations, to account for the new interval, amounts to the same calculation costs as in ISRD numbering.

Thus, we assume for the rest of the chapter that all *shader based* manipulation performed on id values for particles, granules, and general rigid bodies, uses ISRD numbering.

4.3.2 General rigid body storage

A general rigid body can be viewed as a grain with an arbitrary particle count. Thus, nongrain rigid bodies have their positions, linear velocities, angular velocities, and orientations stored in the same textures as granule properties. Similarly, the positions, velocities, and relative positions of particles matched to nongrain rigid bodies are stored alongside the properties of the granule particles.

Thus, we continue to assign ids to rigid bodies and particles in the same way as for grains and their particles, using

$$b_i = g_{\text{last}} + i \quad \text{for} \quad i = 0, 1, 2, \ldots, \tag{4.5}$$

$$q_i = p_{\text{last}} + i \quad \text{for} \quad i = 0, 1, 2, \ldots, \tag{4.6}$$

up to programmatic maximum body and particle counts, $b_{\max}$ and $q_{\max}$. The values g_{last} and p_{last} are the ids for the last stored grains and grain particles, respectively.

For nongrains, the mapping between particle and rigid body ids, given in equation 4.3, continues to work, provided the rigid bodies contain exactly four particles. Once a body with id b_k and a

different body particle count is stored, then bodies with ids $b_{k+1}, b_{k+2}, b_{k+3}, ...$ no longer resolve to the correct starting particle. However, provided we know the particle counts a_i for all nongrain rigid bodies with ids b_i, then a more general equation gives us

$$q_{s,j} = 4b_0 + \sum_{i=0}^{j-1} a_i + s \quad \text{for} \quad s = 0, 1, 2, 3, ..., a_j - 1\,, \tag{4.7}$$

where b_0 refers to the id of the first nongrain rigid body and $q_{s,j}$ is the id for the $(s+1)$th particle of the nongrain rigid body b_j.

However, the reverse operation, that is, going from an arbitrary rigid body id to a particle id, is not trivial. In addition, equation 4.7 involves a summation, which entails substantial calculation for larger quantities of nongrain rigid bodies, unless precomputation is used. In both cases, an easier mechanism, as illustrated previously for granules and their particles (see Figure 4.10), is to store ids explicitly in the fourth component of position and velocity texels, thereby obviating the use of equation 4.7 and its reverse.

A remaining problem is that particle count, mass, and mass moment of inertia must be given explicitly for each nongrain rigid body, since we cannot assume these values are the same for all bodies, as we did for granules. These values may be transformed into a fixed-sized n-vector, such as

$$b_i \rightarrow [m_i, n_i, \mathbf{M_i}], \tag{4.8}$$

where each general rigid body b_i is assigned a mass m_i, particle count n_i, and a serialized moment of inertia matrix M_i (written as a 9-vector rather than a three-by-three matrix). For all nongrains, the matching b_i can be stored consecutively, in the same order as granules and using a fixed number of adjacent texels. Since b_i represents 11 consecutive floating point values, every texel triplet can be used. While one floating-point value is wasted, the advantage is knowing precisely where the 11-vector information can be located. Otherwise, keeping track of the changing pattern of overlap for consecutive values is needed.

In addition, either particle count or mass could be stored in the fourth coordinate of a body's angular momentum texel, instead of in b_i. However, since orientation is represented as a 4-vector (quaternion), not enough space is available to store both values.

Similarly, a body's particle constituents have no space left to store their rigid mass, other than in the fourth coordinate of the relative position vector. However, this is not accessed during force calculation, since granule properties are not needed. Thus, the part of b_i holding the body mass and particle count must be accessed during force calculation to compute the fraction of the body's total mass held by a single particle constituent. In either case, one texture read is required.

Before force calculation can proceed, however, collisions need to be detected among particles. As discussed in Section 3.2, forces arise from particle collisions. Therefore, the next section discusses how to locate particle collisions with the particle storage format described above.

4.4 Collision detection

Simulating granule behaviour means finding the forces that act on them. That is, we need to find colliding pairs of grains and grain-surface collisions. Our approach models geometric surfaces and arbitrary rigid bodies as particle arrangements, which means we need only detect particle-particle collisions to find all possible contact forces in the simulation. To avoid having to examine every potential pair, we use a grid acceleration structure. It has the advantage of a straightforward GPU implementation as well as producing fast construction and query operations.

4.4.1 Grid structure

A grid, as an underlying data structure for particle-based collision detection, enables a simulation to search around an arbitrary point in simulation space in order to locate nearby particles. This is achieved by *discretising* the simulation domain into a 3D rectilinear grid of cells, called *voxels*, where each voxel contains the ID of any particle having its centre inside that voxel.

Thus, each particle position is associated with some voxel. Searching inside and around that voxel produces a list of occupants, which represent potential sources of collision. For a discussion on the relative merits of this scheme compared to other collision detection approaches, please see Section 2.5.2.

4.4.2 Grid storage

As discussed in Section 4.1.1, textures provide an attractive way to store data in GPU memory. A natural representation of simulation space, therefore, might be to map each voxel to a unique texel in a 3D texture. While this schema makes cell storage and lookup straightforward, 3D textures have various limits. For example, writing to a 3D texture is highly CPU bound, since only one 2D "layer" (the set of voxels with the same z-coordinate) at a time, can be rendered to and updated. Each rendering pass is begun by the CPU and *all* the particles must be sent to the pipeline for each pass, since the grid structure itself provides the spatial ordering on particles and the grid is not available during its own construction. Finally, not all hardware is guaranteed to support 3D textures.

An alternative employed in this thesis is to use a *flattened* 3D texture. That is, we map a 3D texture onto a 2D surface. The advantage is we get a higher degree of fragment processor parallelism because of reduced rendering passes [HBS+03, LLW04]. Following this approach, we map the region of 3D space making up the simulation domain to a 2D texture surface, with one-to-one correspondence between voxels and texels, as illustrated in Figure 4.11.

While the idea is simple, subtle difficulties may arise. These are best observed, as well as overcome, once we have an understanding of the parameters involved in both spatial domain representation and grid texture storage.

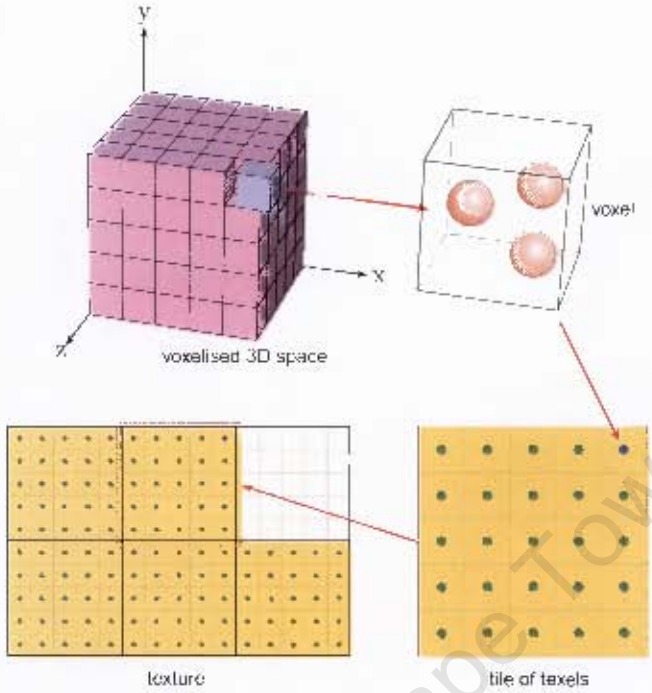


Figure 4.11: Simulation space mapped to a flattened 3D texture. The simulation space is divided into equally-sized cuboids or voxels. Walking along the z-axis away from the origin, we encounter "layers" of voxels having the same z-coordinate. In this instance, there are five layers, where each layer consists of a five-by-five grouping of voxels. To each layer the simulation assigns a subsection of the grid texture, called a *tile*. Each tile consists of a five-by-five grouping of texels, so that one texel stores the data for a unique voxel. Tiles are arranged from the bottom left of the grid texture moving to the right or beginning a new row, as z-values increase. Finally, the simulation maps the ics of particles whose centres occupy the same space as a voxel to the texel associated with that voxel.

Simulation space parameterisation

Suppose the simulation domain is a cuboid C (that is, a parallelepiped with rectangular faces) having width C_w, height C_h, and depth C_d. Similarly, each cell (voxel) c composing the cuboid has width c_w, height c_h, and depth c_d. The relationship between these values is given by

$$\begin{aligned} C_w &= k_w c_w, \\ C_h &= k_h c_h, \\ C_d &= k_d c_d, \end{aligned}$$

for positive integers k_w, k_h, and k_d, which count the number of voxels along the width, height, and breadth of the grid, respectively. Thus, C consists of exactly $k_w \cdot k_h \cdot k_d$ cells and each depth layer $C_d(i)$ of C has $k_w \cdot k_h$ cells for depth $i = 0, 1, 2, ..., k_d - 1$. In addition, note that the dimensions of C and c can be noninteger values.

However, a problem remains in trying to simulate the properties for real-world granules that exist on the scale of millimeters. The physical quantities resulting from simulation, such as force, may

be subject to rounding and truncation error, because the GPU supports only single-precision floating-point computation [NVI08a].

The solution used in this thesis is to adjust the grid dimensions and cell dimensions to meet the peculiarities and limits of the graphics system and GPU. To address GPU floating-point limits, we use larger than "real-world" magnitudes when writing particle and grain data to texture or when rendering particles. However, as physical calculations demand proper physical units, we introduce *scaling factors*, one for each dimension. These are used to scale down particle properties, such as position and size, from which other physical quantities are calculated. Properties that result from these calculations are then scaled up before storage.

In addition, the simulation in this thesis uses *cubic* cells, so that separate scaling factors for each dimension are unnecessary, so that a single scaling factor, c_s, suffices for all physical values. To further simplify calculation, we may use cells of unit side length, that is, with $c_x = c_y = c_z = 1$. This has implications for particle diameter, because the grid must use a finite texture to store its particle content. Specifically, the grid maps the ids of particles in each grid cell to a unique texel, which limits cells to at most four particles. To update cells correctly, we set the unscaled particle radius, for all particles, to a little less than half the distance between the antipodal corners of a unit-sided cell, that is, to $\sqrt{3}-\epsilon$ for some small ϵ. This allows four particles to fit into a single cell. We also notice that particle diameters are close to unity, that is, a cell's side length. This means that setting the previously described scaling factor to a value roughly equal to the diameter of a real-world grain will scale the whole system to real-world physical dimensions.

Grid texture storage

The texture surface, used for grid storage, has positive integer width T_w and height T_h, which provides $T_w \cdot T_h$ one-by-one texels for cell storage. Two conditions apply to the texture surface. First, is that $k_w \cdot k_h \cdot k_d \leq T_w \cdot T_h$ so that sufficient texels exist to store all cells. Second, we must have $k_w \leq T_w$ and $k_h \leq T_h$ so that at least one depth layer $C_d(i)$ can fit inside the texture. In addition, we are free to use rectangular textures, as discussed in Section 4.1.1, provided they are within hardware limits. The simulation in this thesis was developed and tested on a graphics card[5] supporting textures with both $T_w \leq 8192$ and $T_h \leq 8192$.

To create an easy mapping between simulation and texture spaces, we tile a texture with layers $C_d(i)$ in the order $i = 0, 1, 2, ..., k_d - 1$. That is, thinking of each layer as a tile, we fill texture space (T_w, T_h) by placing tiles side by side, starting at the bottom left with tile (layer) $C_d(0)$ and moving to the right for subsequent tile placement. This is performed until either a tile cannot fit into the remaining space (if any) or no tiles remain. In the former case, remaining tiles are placed in the next row directly above the finished row using the same procedure as before. The result of such a tiling is illustrated in Figure 4.11.

We are still free to select different texture dimensions, provided the tiling procedure produces

[5]NVIDIA 8800 GTX [NVI08a]

an arrangement that fits. However, such an arrangement may not form a rectangle, nor cover the entire texture surface, which means many texels are left unused (unmapped). The questions remains as to the "optimal" choice of texture dimensions, that is, leading to the least space wastage.

In Appendix A.1, a procedure for finding such an optimal covering is given. Among other guarantees, it makes possible the assumption that a texture used for grid tile storage fits tightly to the total width of the tile arrangement. Specifically, $T_c = \frac{T_w}{k_w}$, the number of tiles per row of tiles in the texture, is an integer.

Given the texture dimensions, we need a mapping facility for transforming an arbitrary position $\mathbf{p} = (p_x, p_y, p_z)$ (relative to the origin) in simulation space to a two-dimensional texture coordinate (s, t) on the grid. First note that $\mathbf{q} = (\lfloor \frac{p_x}{c_x} \rfloor, \lfloor \frac{p_y}{c_y} \rfloor, \lfloor \frac{p_z}{c_z} \rfloor)$ represents the grid index for the cell enclosing $\mathbf{p}$. This vector corresponds to the corner of the cell closest to the origin, which uniquely identifies each cell. Thus we can find the matching texture coordinates by

$$s = (q_z \ \% \ T_c)k_w + q_x, \tag{4.9}$$

$$t = k_h \lfloor \frac{q_z}{T_c} \rfloor + q_y, \tag{4.10}$$

where '%' represents the *modulus* operation, which returns the remainder after integer division. In shader programming, this operation may not always be available since not all graphics cards support integer operations. Therefore, an alternative calculation is

$$s = \left(q_z - q_z \lfloor \frac{q_z}{r} \rfloor \right) k_w + q_x, \tag{4.11}$$

which uses the floating-point *floor* operation '$\lfloor \ \rfloor$' instead.

Thus, equations 4.9 (or 4.11) and 4.10 enable the simulation to write a particle's id, based on its cell position, to a matching texel. This allows the simulation to update the grid data structure, as we see in the next section.

4.4.3 Grid updates

The first issue is finding a way to update the grid texture using the rendering pipeline. The method applicable in this case is *render-to-texture*, where a texture is set as the target surface using an FBO and the fragment shader draws (writes) to the texture surface, storing particle ids in texels (see Section 4.1.3). However, a fragment shader cannot alter the position of the surface point to which it writes. This means that the output texel cannot be selected using a fragment shader alone. A vertex shader is needed as well, to position fragments so that a texel, matched to a particular cell, receives the ids for the particles in that cell. Figure 4.12 illustrates this approach.

Thus, we see that by implementing equations 4.9 (or 4.11) and 4.10 in the vertex shader, 3D

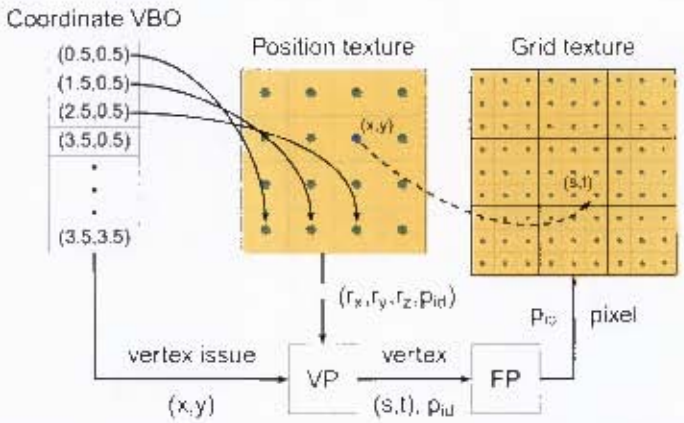


Figure 4.12: Mapping particle positions to the grid structure. The coordinate VBO contains 2D vertices that we set to point to unique texel centres in the particle position texture. These vertices are sent to the rendering pipeline and its vertex processor (VP), where each input (x, y) initiates a vertex shader instance. Each vertex shader uses its input (x, y) as a texture coordinate to look up the matching data from the particle position texture. The retrieved texel data gives particle position and id information (r_x, r_y, r_z, p_{id}). The vertex shader then transforms the 3D simulation space position into the 2D grid position (s, t), where the particle id p_{id} must be stored. The latter two results are outputted from the VP as a vertex (s, t) and its associated data p_{id}. The vertex is rasterised to a fragment at position (s, t) on the output surface and sent to the fragment processor (FP) with its associated data p_{id}. The fragment shader instance spawned by the fragment then writes p_{id} to the surface as a pixel. The colour of that pixel is set to $(p_{id}, p_{id}, p_{id}, p_{id})$.

particle position are transformed into 2D grid positions by the vertex shader and particle ids are written to the correct texels by the fragment shader. However, multiple particle ids may be written to the same texel location. This entails using a special technique to *order* particle ids for writing.

Ordering particle IDs

Suppose that a cell can contain at most four particles and that c_k is one such fully occupied cell. In particular, c_k contains particles with ids p_0, p_1, p_2, and p_3. Without loss of generality we may assume $0 < p_i < p_{i-1}$ for $i = 0, 1, 2$.[6]

In addition, when we invoke a rendering pass, the fragment shader handling particle p_i will attempt to write a 4-vector colour (p_i, p_i, p_i, p_i) to the unique 4-vector texel $\mathbf{t_k}$ matched to a 4-vector grid cell $\mathbf{c_k}$. The latter mapping is due to the *vertex shader* correctly positioning where the fragment shader will send its output. Further, every $\mathbf{t_k}$ is initialised to $(0, 0, 0, 0)$ before the update procedure.

OpenGL and graphics hardware support masking colour components when writing colours to a surface. For example, a colour mask $\mathbf{m}$ of $(1, 0, 0, 0)$, means only the R-channel (red) of the

[6]Particle ids are assumed to be nonzero, since this section and those that follow describe the *storage* of particle ids. This requires that particle ids first be incremented by one, before storage (see ISRD numbering in Section 4.3.1). Thus, we are able to identify empty voxels as those with 0 texel components.

(R,G,B,A) pixel is written out to the texel. For the fragment shader writing (p_i, p_i, p_i, p_i), a successful update produces $\mathbf{t_k} = (p_i, 0, 0, 0)$.

For the situation in which four particles occupy the same grid cell, exactly four different shader instances will write their matching particle ids to the same texel. Setting the mask so that $\mathbf{m}[j] = 1$ for some j (meaning the jth element of $\mathbf{m}$ counting from 0), we know that component $\mathbf{t_k}[j]$ is written to four times. In addition, we know the order of writing, since it is the same as the order vertices are issued to the rendering pipeline, that is, the order they are stored in the VBO. As a result of this ordering, $\mathbf{t_k}[j]$ holds the last particle id written. Thus, to produce the correct value in the first component, we sets an initial mask to $(1, 0, 0, 0)$ and render vertices from largest to smallest, based on their particle ids, as illustrated in Figure 4.13.

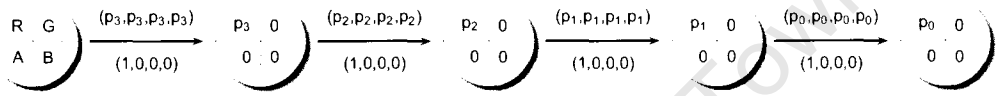


Figure 4.13: Storing p_0 in the first texel position. The texel centre stores the 4-vector (R,G,B,A) where all components are initialised to zero, that is, $R = G = B = A = 0$, at the start of the grid update. The diagram illustrates *one* rendering pass in which four fragment shader instances write their colour data (p_i, p_i, p_i, p_i) to the same texel centre. This occurs whenever four particles occupy a single grid cell, since a cell is mapped by calculation to exactly one texel centre. The order of these writes is the same as the order corresponding vertices are sent to the pipeline, that is, the order of vertices in the VBO. For one rendering a pass, a single mask (below each arrow) is applied to *all* writes. The result here is for particles sent to the pipeline in reverse numerical order of their ids.

However, supposing the first pass wrote p_0 to $\mathbf{t_k}[0]$, trying the same rendering technique for a subsequent mask, such as $(0, 1, 0, 0)$, produces $(p_0, p_0, 0, 0)$ in $\mathbf{t_k}$. Clearly, this technique alone is not sufficient to store all particle ids correctly, only the smallest id.

The previous approach fails because it lacks a mechanism for comparing what is already stored to what is being stored. However, Harada et al. [HKK07] demonstrate how to produce these comparisons, using *depth* and *stencil buffers* as additional particle id filters (see Section 4.1.3 for a discussion of these buffers in an FBO context). This technique assumes both buffers are attached to the FBO used for updating the grid texture (itself, attached to the FBO's colour buffer). In addition, the stencil buffer is initialised to zero and the depth buffer is initialised to the value $p_{\max} + 1$, where $p_{\max}$ is the largest stored particle id[7].

We also assume the presence of a vertex shader, as mentioned previously, for setting the pixel target of the matching fragment shader. The latter shader will attempt to write both a "colour" (p_i, p_i, p_i, p_i) and a *fragment depth* $d_k = p_i$. The method of Harada et al. [HKK07] now proceeds in four stages, each initiated with a rendering pass. For each pass, all the vertex data is sent to the GPU with the FBO and shaders activated.

The ***first rendering pass*** places p_0 at $t_k[0]$. This is achieved by using a colour mask $(1, 0, 0, 0)$,

[7]We are assuming depth values are not normalised. This is a valid assumption, since the shader writing these values can normalise them using the maximum particle id $p_{\max}$ to give ${p_i}' = \frac{p_i}{p_{\max}+1}$ where ${p_i}' \in [0, 1]$. Shaders that read these values from texture then produce the unnormalised id by a reverse calculation. Alternatively, an OpenGL extension supported on many GPUs, allows writing of unnormalised depth values to the depth buffer [BS08].

disabling the stencil buffer, enabling the depth buffer with a *less-than* comparison function, and then rendering the vertices from smallest to largest ordered by particle ids. Figure 4.14 illustrates this step.

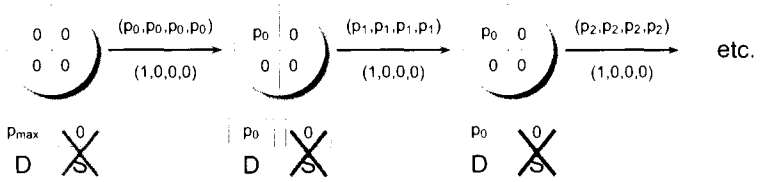


Figure 4.14: First rendering pass. Both depth testing and depth updates are active, while the stencil buffer is inactive. The depth test is set to pass for depth values *less than* the stored value. In addition, a value passing the depth test pass replaces the stored depth value. Thus, once the value p_0 is written to both the surface and the depth buffer, all other (subsequent) id values mapping to the same depth position (that is, texel position) will be greater than p_0 by definition, thereby failing the depth test.

The ***second rendering pass*** places p_1 at $\mathbf{t_k}[1]$. This is achieved by using a colour mask $(0, 1, 0, 0)$ and switching the depth buffer comparison function to *greater-than*. In addition, stencil buffer is now activated with a *greater-than* comparison function and reference value of 1. That is, the stencil test passes when the reference value is greater than the stencil value s_k at the pixel. The stencil *increment* function is set to increase the stored stencil value s_k by one when both the depth and stencil test pass. Finally, the vertices are rendered as before. Figure 4.15 illustrates this step.

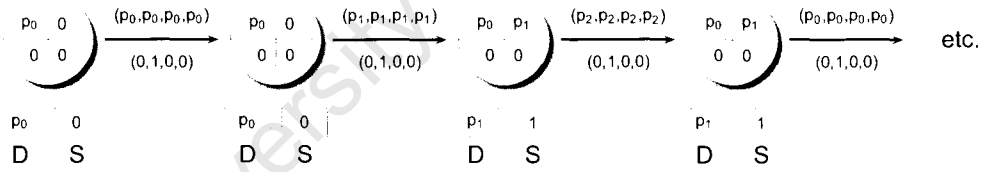


Figure 4.15: Second rendering pass. Both the depth (D) and stencil (S) tests use a *greater-than* comparison. The depth test passes for values greater than the stored depth. The stencil test is given a *reference value* of 1, so that a pass occurs when the reference value is greater than the stored stencil value. If both the depth and stencil tests pass, the stencil update function *increments* the stored stencil value. For the first masked vector, the matching fragment shader instance writes a depth value of p_0, failing the depth test. The next write passes the depth test, since $p_1 > d_k$. In addition, the stencil test passes, since $1 > s_k = 0$. Since the fragment passes both tests, the depth is updated to $d_k = p_1$ and the stencil value s_k is incremented by 1. Any subsequent writes will involve particle ids that pass the depth test, but fail the stencil test, so no writing or updating occurs.

Both the ***third*** and ***fourth render passes*** proceed analogously to the second pass. The third pass clears the stencil buffer (so that $s_k = 0$ for all k) and renders particles using the next write mask $(0, 0, 1, 0)$. The fragment with depth p_2 is the first to pass both tests. Thus, the next fragment p_3 fails the stencil test and does not overwrite p_2. Finally, the fourth pass clears the stencil buffer, sets the write mask to $(0, 0, 0, 1)$, and renders the particles again, so the fragment with depth p_4 is the first and last fragment to pass both tests.

Thus, excepting the first pass, we have used the depth buffer to keep track of what was stored successfully in the *last* pass, while using the stencil buffers to inhibit further storage once the

correct particle id is stored *during* a pass.

4.4.4 Grid collision detection

Collision detection is much easier now with the heavy work of constructing the grid out of the way. For each particle instance, its position in the simulation domain, $\mathbf{p} = (p_x, p_y, p_z)$, is converted to a position in grid space, $\mathbf{p}' = \mathbf{p} - \mathbf{o}$, where $\mathbf{o}$ is the vector pointing from the origin of the domain to the origin of the grid space. This covers the case when the domain and grid are not coincident, but continues to assume parallel sides and matching orientations. In addition, the grid is assumed to *cover* or contain the entire simulation domain.

The simulation in this thesis uses a grid of the same size, origin, and orientation as the simulation domain, giving $\mathbf{p}' = \mathbf{p}$. In addition, the simulation assumes cube-shaped grid cells of unit side length, that is, $c_x = c_y = c_z = 1$.

Thus, the cell index is given by $\mathbf{q} = (\lfloor \frac{p_x'}{c_x} \rfloor, \lfloor \frac{p_y'}{c_y} \rfloor, \lfloor \frac{p_z'}{c_z} \rfloor)$, for which equations 4.9 (or 4.11) and 4.10 give the matching texel coordinate (s, t). The same texel coordinate retrieval can be done for the 26 cells surrounding the particle's cell. For each texel coordinate, the matching texel is retrieved and the particle ids it contains are converted to actual particle positions. The latter is achieved through equations 4.1 and 4.2, which govern granule ids, but work analogously for particle ids by replacing granule texture width with particle texture width.

Lastly, we must distinguish between zero values and the particle of id 0, since the latter may be confused with an empty component rather than the id of a cell occupant. This confusion is avoided by always incrementing a particle id before storing it and decrementing the id when it is read from a texel. Thus, a texel component of value 0 implies the absence of a particle, rather than the presence of particle p_0.

A runtime problem might arise if large forces compress more than four particles into a single cell. Provided such forces remain within expected physical limits, the simulation can continue running without instability. The issue here is that a fifth or sixth particle will not have its id stored in the texel matched to its grid cell. This makes these particles invisible to surrounding particles, which cannot detect their presence in the grid. On other hand, these invisible particles still detect other particles whose ids reflect in the present and nearby cells. These forces serve to push the errant particles away from preexisting cellular occupants, keeping the simulation stable. Unfortunately, this can create oscillating behaviour in the material.

The algorithmic detail for collision detection is described in Pseudocode 4.1, assuming a fragment shader implementation, as is needed for force calculation.

The fragment shader, given in Pseudocode 4.1, retrieves its grain id for free, since this value is contained in the associated particle's velocity, which is needed for later force calculation. This assumes that at least one positive collision occurs, thus resulting in a force calculation. The assumption is valid, since positive collisions are expected to occur for the majority of particles,

Pseudocode 4.1 Grid collision detection (for force calculation)

```
GPU:
  Fragment program:
    p ← read position texture using fragment texture coordinate
    v ← read velocity texture using fragment texture coordinate
    myId ← p_w      //fourth component stores the particle's id
    myGrainId ← v_w      //fourth component stores the particle's parent's id
    2r ← particle diameter
    (g_w, g_h, g_d) ← grid dimensions      //width, height, and depth
    c_s ← scaling factor

    //the following ensures only voxels inside the grid volume are looked up
    if (p_x ≤ r) then x_1 ← 0 else x_1 ← −1
    if (p_y ≤ r) then y_1 ← 0 else y_1 ← −1
    if (p_z ≤ r) then z_1 ← 0 else z_1 ← −1
    if (p_x ≥ g_w − r) then x_2 ← 0 else x_2 ← 1
    if (p_y ≥ g_h − r) then y_2 ← 0 else y_2 ← 1
    if (p_z ≥ g_d − r) then z_2 ← 0 else z_2 ← 1

    for z = z_1 to z_2 do      //loop depth values first to access texels on one grid tile
      (s, t) ← derived from particle position p and z      //grid texel position of centre voxel
      for y = y_1 to y_2 do
        for x = x_1 to x_2 do
          T ← read grid texture at position (s + x, t + y)      //4-vector texel returned
          for i := 0 to 3 do
            id ← T[i] − 1
            (p, q) ← convert id into particle texture coordinate
            v' ← read velocity texture at (p, q)
            if id ≠ −1 and id ≠ myId and v_w ≠ myGrainId then
              p' ← read position texture at (p, q)
              if p and p' interpenetrate then
                scale particle position p and other particle position p' by c_s
                ...compute forces here...
              end if
            end if
          end for
        end for
      end for
    end for
```

when they compose granules forming a sand pile. The *other* particle's velocity is also read, initially to obtain an associated grain id. However, this value is less likely to be used subsequently. This situation can be improved when no general rigid bodies are used in the simulation, that is, when each body is composed of exactly four particles. In this case, checking whether particles are from the same grain amounts to checking if the values $\frac{myId}{4}$ and $\frac{id}{4}$ are equal, assuming integer division (where the division operator discards remainders). Of course, at least one general rigid body, the floor surface, needs to be present. However, this can be handled implicitly outside of rigid body computation, as discussed next.

Implicit rigid bodies

Simple stationary boundaries in the simulation, such as the floor and walls, admit uncomplicated representations, either as implicit functions or as flat rigid bodies. In the latter representation, we create the rigid body explicitly (as described later in Chapter 6) and store this data in a file. The simulation then loads this at runtime in the same manner as normal rigid bodies. The only difference is that a body's variable properties, such as velocity and orientation, are kept constant to prevent movement.

The explicit approach has the convenience of ignoring surface-based properties like normals, as particle-particle contacts drive all grain-surface interaction. In contrast, for an implicit function approach, we need to have computable expressions for two surface quantities. These are the closest surface point to any given point in space and the normal at any given surface point. Having the first expression enables the simulation to decide, given a particle's position, if it lies close enough to the surface to produce contact. In this event, the second expression allows calculation of the normal contact force.

Thus, we are able to treat each implicit surface point as a particle having zero radius. However, this would produce a smooth surface, which defeats our goal of using particulate roughness to emulate static friction. Instead, we sample the implicit surface evenly, to produce a finite quantity of particles of non-zero radius. These together comprise a "thick" particulate version of the underlying surface without any holes. Importantly, this requires a third property for each implicit function, namely, having a coordinate chart (2D representation) for the surface, which we can sample evenly with grid points (particle positions). The process is illustrated in Figure 4.16 for a plane surface.[8]

Whether collisions occur among grain particles or between a grain particle and a rigid body or implicit particle surface, forces must be computed.

4.5 Force on particles

Force calculations are applied when collisions are located. In particular, the force calculation to follow is assumed to occur at the place indicated previously in Pseudocode 4.1. The latter code thus forms *part* of a fragment shader for force calculation. In particular, once a collision is detected a function in the fragment shader can be called for computing the resulting force. This function is described in Pseudocode 4.2 and represents an implementation of the DEM force model (summarised in Section 3.2.4). This function is assumed to have access to the variables it accesses either through global scope within the force fragment shader or through its arguments.

[8]Finding the four closest particles entails projecting the vector $\mathbf{p} - \mathbf{s}$ onto the plane's surface to obtain the vector $\mathbf{p}' = (\mathbf{p} - \mathbf{s}) - ((\mathbf{p} - \mathbf{s}) \cdot \mathbf{n})\mathbf{n}$ where $\mathbf{n}$ is the normal vector for the plane. This vector position for the particle on the plane is divided into components along the directions given by $\mathbf{a}$ and $\mathbf{b}$. These components can then be used to locate the one or two closest plane particle centres along each direction. Thus, one, two, or four possible collision tests may result. These correspond to the cases where a particle lies directly above one, between two, or at the centre of four plane particles.

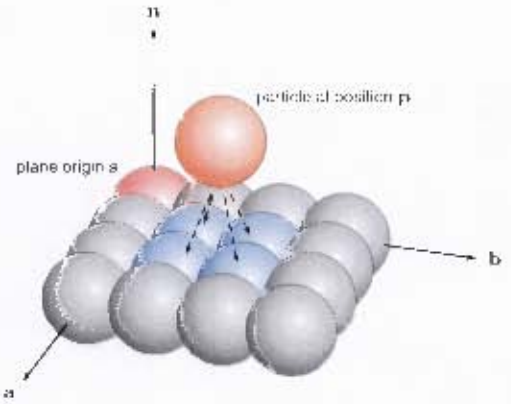


Figure 4.16: Collision detection for an implicitly represented wall or floor. The implicit representation begins by orienting and positioning a finite plane having normal $\mathbf{n}$ and origin $\mathbf{s}$ (centre of the red sphere). The sides of the plane are given by the vectors $\mathbf{a}$ and $\mathbf{b}$. The magnitude of the latter two vectors indicate the length of these sides. Alternatively, $\mathbf{a}$ and $\mathbf{b}$ can be unit vectors, which requires that we explicitly store the side lengths a and b. Given the radius r_p of the particles implicitly forming the plane and the radius r_g of the grain's particle, we can find those plane particles (blue spheres) that might produce a collision with the grain. Before finding these particles, however, the simulation tests whether $r_p + r_g$ is less than the distance between the grain particle's centre and the plane itself (not the particles making it up). If this is not the case, then no collision is possible. Otherwise, a collision *might* occur, except when a particle moves into a depression, formed at the centre of four plane particles.

Pseudocode 4.2 Force calculation

Fragment program:

input: time step dt
input: particle diameter L *//assume all particles same size*
input: current particle's mass m_1, position $\mathbf{p_1}$, and velocity $\mathbf{v_1}$
input: other particle's mass m_2, position $\mathbf{p_2}$, and velocity $\mathbf{v_2}$
input: effective Young's modulus E_{eff}
input: restitution coefficient e_n
input: viscous damping term γ_s
$\mathbf{f} \leftarrow (0, 0, 0)$
$\mathbf{n_d} \leftarrow \mathbf{p_2} - \mathbf{p_1}$ *//direction from the particle's centre to other particle's centre*
$\mathbf{n} \leftarrow \text{normalise}(\mathbf{n_d})$
$m' \leftarrow 1/m_1 + 1/m_2$
$m_{\text{eff}} \leftarrow 1/m'$ *//effective mass of particles*
$\gamma_n \leftarrow -m_{\text{eff}} \log(e_n) / dt$
$k_n \leftarrow (4/3)\, E_{\text{eff}} \,\text{sqrt}(L/4)$
$\xi_n \leftarrow \max(0.0, L - \text{length}(\mathbf{n_d}))$
$\mathbf{v_c} \leftarrow \mathbf{v_2} - \mathbf{v_1}$
$\mathbf{f} \leftarrow \mathbf{f} - (\gamma_n \,\text{sqrt}(\xi_n)\, \text{dot}(\mathbf{v_c}, \mathbf{n}) + k_n \,\text{sqrt}(\xi_n)\, \xi_n)\, \mathbf{n}$ *//add normal force*
$\mathbf{v_s} \leftarrow \mathbf{v_c} - \text{dot}(\mathbf{v_c}, \mathbf{n})\, \mathbf{n}$
$\mathbf{f} \leftarrow \mathbf{f} - \min(\mu k, \gamma_s \,\text{length}(\mathbf{v_s}))\, \text{normalize}(\mathbf{v_s})$ *//add tangential force*
output: final force $\mathbf{f}$

Typical values for the effective Young's modulus E_{eff}, restitution coefficient e_n, and viscous damping term γ_s, are 3.0×10^9, 0.65, and 10.0, respectively.

The result of this stage is a force texture containing the total force acting on each *particle*. The next section discusses how this is used to produce grain motion and update particle properties.

4.6 Grain motion

The previous stage stores the total force acting on each particle in the simulation. These forces need to be accumulated and converted into motion at the grain level. The equations governing this conversion are summarised in Section 3.3.4. They provide the simulation with a grain's new linear velocity, angular velocity, orientation, and position, based on the total force acting on the grain.

Grain property update

The total force summation for grains is simplified by decomposition of forces into their linear and rotational components. The linear force acting on a grain is simply the summation of the total forces acting on each of its particles (see Section 3.3.1). Similarly, the rotational force is the summation of the *torques* that act on the grain at each of its particles (see Section 3.3.2). However, the latter involves taking a cross-product of a particle's relative position (to the grain's centre of mass) and the force acting on the particle, before accumulating the result to the total torque. The fragment shader algorithm for performing both linear and rotational force accumulation, as well as grain property updates, is described in Pseudocode 4.3. When rigid body position and velocity are updated in the code, we assume that the fourth component is unaltered. That is, the final position and velocity written to the output texture surface have the same fourth components as the position and velocity that are read from the matching input textures. Finally, the *rigid body property* texture holds the values that are not uniform across nongranule rigid bodies, as discussed in Section 4.3.2.

Additionally, the simulation can damp the rotation of grains experiencing low angular velocity. This avoids jitter from inaccuracy in the low-order integration scheme, which would otherwise lead to slow oscillation of supposedly motionless particles. The calculation is described in Pseudocode 4.4, which is inserted into Pseudocode 4.3 right before the final angular velocity $\mathbf{w}$ is written to a colour surface.

Particle property update

Relative particle position could be derived using the new grain orientation to transform particle relative positions lying in *model space* to their new relative positions. Provided each grain has the same model-space positioning, the canonical relative positioning could be made part of the

Pseudocode 4.3 Grain property update

Fragment program:

//Values read from their matching textures, based on current fragment coordinate:
$\mathbf{p} \leftarrow$ rigid body (RB) position
$\mathbf{v} \leftarrow$ RB linear velocity
$\mathbf{w} \leftarrow$ RB angular velocity
$\mathbf{q} \leftarrow$ RB orientation
//Read from RB property texture
$n, m, I \leftarrow$ particle count n, RB mass m, and mass moment of inertia I

$id \leftarrow p_w - 1$ *//ISRD numbering*
if $id == -1$ **then**
  discard this fragment *//no updates to the output texel position occur*
end if
$\mathbf{p} \leftarrow \mathbf{p} \cdot c_s$ *//scale RB position by scaling factor*
$\mathbf{f} \leftarrow (0,0,0)$ *//final force*
$\mathbf{t} \leftarrow (0,0,0)$ *//final torque*
for $i := 0$ **to** $n-1$ **do**
  $\mathbf{r} \leftarrow$ read particle texture at coordinate specified by particle id $pid + i$
  $\mathbf{r} \leftarrow \mathbf{r} \cdot c_s$ *//scale relative position*
  $\mathbf{f}' \leftarrow$ read force texture at coordinate specified by particle id $pid + i$
  $\mathbf{f} \leftarrow \mathbf{f} + \mathbf{f}'$
  $\mathbf{t} \leftarrow \mathbf{t} + \mathbf{r} \times \mathbf{f}'$ *//vector cross-product in the second term*
end for
$\mathbf{v} \leftarrow \mathbf{v} + (\mathbf{f} \cdot dt)/m$
if length$(\mathbf{v}) \cdot dt > 1$ **then**
  $\mathbf{v} \leftarrow \frac{1}{dt}$normalise$(\mathbf{v})$
end if
$\mathbf{q}' \leftarrow$ normalise(q) *//compute normalised orientation (quaternion)*
$R \leftarrow$ computeRotation(q') *//find rotation matrix representation*
$I^{-1} \leftarrow R \cdot I \cdot$ transpose(R)
$\mathbf{w} = I^{-1} \cdot \mathbf{t} \cdot dt$ *//matrix times vector times scalar*
if length$(\mathbf{w}) \cdot dt > \frac{\pi}{2}$ **then**
  $\mathbf{w} = \mathbf{w} \cdot (\frac{\pi}{2}/dt)/$length$(\mathbf{w})$
end if
// ...Insert Pseudocode 4.4 here...
ColourOutput[0] := $\mathbf{r}$
ColourOutput[1] := $\mathbf{w}$
ColourOutput[2] := $\mathbf{p} + \frac{\mathbf{v} \cdot dt}{c_s}$ *//unscale position*
ColourOutput[3] := computeOrientation$(\mathbf{q}, \mathbf{w}, dt)$

Pseudocode 4.4 Damping step (within grain update shader)

Fragment program:

if length$(\mathbf{w}) < \epsilon$ **then**
  if length$(\mathbf{w}) > v_{damp}$ **then**
    $\mathbf{w} = \mathbf{w} -$ normalise$(\mathbf{w}) \cdot v_{damp}$
  else
    $\mathbf{w} = \mathbf{0}$
  end if
end if

shader or uploaded from the CPU as a uniform variable.

This assumes that all grains are the same, which decreases the generality of the simulation. One alternative, is to store the original relative positioning of each grain and derive the current relative positions using the current grain orientation. The second alternative is to keep relative positioning up-to-date when other particle parameters are being updated. The latter case has the advantage that each relative position update is carried out in a separate particle shader, whereas the first alternative means bringing four particle relative positions up-to-date for each grain, decreasing parallelism. The simulation, therefore, adjusts relative position to match orientation on a particle-by-particle basis.

The update of particle properties, including relative positions, is based on the result of the previous grain updates. The derivation of particle position and velocity is described in Section 3.3.3. Pseudocode 4.5 produces these updates.

Pseudocode 4.5 Particle property update

Fragment program:
  input: scaling factor c_s
  $\mathbf{r} \leftarrow$ relative particle position read from matching texture at fragment coordinate
  $\mathbf{v} \leftarrow$ particle velocity read from matching texture at fragment coordinate
  $g_{id} \leftarrow v.w - 1$ *//fourth coordinate of particle velocity gives parent grain id*
  if $g_{id} == -1$ **then**
    discard this fragment *//no updates to the output texel position occur*
  end if
  $(s,t) \leftarrow$ convert g_{id} to 2D texture coordinate
  $\mathbf{p} \leftarrow$ rigid body (RB) position at (s,t)
  $\mathbf{v} \leftarrow$ RB linear velocity at (s,t)
  $\mathbf{w} \leftarrow$ RB angular velocity at (s,t)
  $\mathbf{q} \leftarrow$ RB orientation at (s,t)
  $\mathbf{r}' \leftarrow$ store the result of rotating $\mathbf{r}$ by the quaternion $\mathbf{q}$
  ColourOutput[0] $:= \mathbf{r}' + \mathbf{p}$ *//p.w stored in fourth coordinate*
  ColourOutput[1] $:= \mathbf{v} + \mathbf{w} \times (\mathbf{r}' \cdot c_s)$ *//v.w stored in fourth coordinate*
  ColourOutput[2] $:= \mathbf{r}'$ *//r.w stored in fourth coordinate*

4.7 Summary

The prominent role, but poor representation of sand in many real-time games has previously received little attention. Drawing on nontrivial rendering techniques and GPU-based programming, this chapter has demonstrated a framework for simulating the physics of a high level-of-detail granular sand model, entirely on the GPU. In this way, both collision detection and grain property updates are accelerated to real-time performance inside the rendering pipeline. In addition, we have described a novel mechanism for representing simple surfaces implicitly as particles. The advantage of the implicit representation is the implicit rigid body surfaces and their particle constituents do not consume GPU texture memory. Thus, texture space is freed for additional, explicitly represented rigid bodies and their particles. The final output, while

compelling in its granular behaviour, lacks visual appeal. This deficiency is addressed in the following chapter.

Chapter 5

Rendering granular material

In the previous chapter we described a real-time GPU-based framework for simulating large numbers of interacting sand grains. We now turn to the task of visualising the motion of these grains and their particle constituents. The focus of this chapter is real-time visualisation and high-quality lighting of sand.

Unfortunately, standard rendering techniques do not produce real-time results for large granule counts. Naïvely representing particles as spheres, for example, leads to poor GPU performance, since spheres must be approximated by polyhedrons. For each polyhedron, a higher vertex count produces more circular silhouettes and a smoother appearance. However, all vertices for each polyhedron must pass through the rendering pipeline. In addition, each visible vertex requires expensive lighting calculations for diffuse shading and possibly specular effects. The problem here is that using one additional vertex for a poorly approximated particle representation produces a multiplicative increase in vertex workload and vertex-based calculation.

In this chapter, we address this problem with a novel technique requiring only one vertex for each particle. In addition, the demonstrated method produces diffuse and specular lighting, as well as grain self-shadowing and environmental shadowing, in real-time.

The chapter begins with a brief review of previous work addressing the visualisation of granular materials. In addition, relevant and well-known computer graphics rendering techniques, used in the rendering implementation, are explained.

5.1 Background

This thesis is concerned with the visualisation of particles composing granules. In particular, the framework requires that visualisation uses particle positions as they evolve, in real-time. In addition, granules require lighting so that they appear three-dimensional as well as shadow their particle constituents, other granules, and their environment. Previous work on both these topics

is reviewed briefly in the following sections.

5.1.1 Visualisation

Many approaches to rendering granular materials have focused on visualising the sand pile as a whole. Continuum visualisation, specific to granular material, has been achieved through *offline* ray tracing and texturing of saved simulation data [ZB05]. In contrast, height-map approaches have demonstrated real-time results [Nor06, DBM07], including additional particle effects [Nor06]. Height map representations for hydraulic erosion of purportedly sandy terrains has also been demonstrated in real-time using GPU acceleration [ASA07, MDH07].

Of greater interest, however, are the few techniques for visualising granular material at the level of individual grains. A recent technique visualises two-dimensional DEM data with the aim of evincing physical properties of grain interaction for each frame of captured data [MSW+08]. Microstructure is visualised as glyphs showing force behaviour between single-particle granules. However, the focus is on visualisation of scientific data, not on producing realistic looking sand for real-time graphics applications.

A related technique for visualising particle data is *point splatting* [PZvB+00, RL00, ZvBG01]. This involves projecting a disc for each point into screen space. Various techniques have been used to shade, blend, or filter the resulting fragments to produce a seemingly complete unaliased surfaces. The advantage here is that points are not connected to one another and therefore are processed as independent geometry. In contrast, polygonated models result in increasingly costly scanline conversion, as the vertex count increases, since more triangular faces must be converted line-by-line into fragments [LW85]. More recent work has used a GPU to accelerate phong shading and environment mapping for surfel-based visualisation [PZvB+00].

Point splatting inspired the technique described later in this chapter. However, instead of blending, we give discs implicit spherical geometry, thereby modelling spherical particles and allowing the framework to shade and light granules as a whole. The lighting techniques used in this thesis are based on well-known real-time algorithms, which are discussed in the next section.

5.1.2 Real-time lighting

Shadows are important in animation for the feeling of depth they provide to otherwise flat objects. Scene shadowing usually involves one of two possible procedures. Either we apply a physically correct lighting model such as ray tracing [WH80] or use rasterization. In general, ray tracing is slower [Gla89], though GPU-based implementation are beginning to provide real-time performance [PBM+05]. Nevertheless, we use rasterisation, as it is generally understood to produces real-time results at present [WSB+01].

For rasterisation, we render a scene to screen space, where each pixel of this screen space is lit independently of all others, irrespective of whether they originate from the same surface (as

discussed at the beginning of Section 4.1). However, determining the light reaching that pixel is not simple, at this stage of the process, since explicit geometry information for the whole scene is no longer available within the rendering pipeline. Many techniques that address lighting, therefore, look at how to relate the colour result of a pixel to the scene geometry at this stage of the rendering process. We briefly review reflectance and shadowing with this rendering pipeline limitation in mind.

Phong reflectance model

Unlike computationally expensive ray tracing algorithms that model reflectance behaviour based on a theoretical understanding of light, the Phong reflectance model takes a cheaper empirical approach that boasts compelling lighting at a fraction of the cost [Pho75, Bli77]. The model, however, is limited to reflectance behaviour alone and does not address complex light behaviour, such as refraction. In particular, the model distinguishes between three types of reflection that occur with variations in local surface microstructure, as illustrated in Figure 5.1.

Figure 5.1: Phong reflectance model. (a) Ambient reflection represents a measure of the indirect lighting in a scene. Specifically it refers to the light reflected off nonshiny surfaces, such as carpeted floors or matte-finished walls, which illuminates other areas of the scene not in direct view of the light source. Note that the uniform colouring produces a "flat" torus. (b) Diffuse lighting gives a matte appearance to surfaces in direct view of the light source. (c) Specular reflection models perfectly reflected light behaviour and produces mirror-like glossy surfaces, typical of shiny metals. (d) The full phong reflectance model is the additive combination of the individual reflections.

The *specular* component of the reflectance model, however, is computationally expensive. The problem is a dependence on the angle between the viewer and the ray reflected off the surface, which must be recalculated for *each* pixel during a frame of animation. Instead, this thesis uses the *Blinn-Phong* model, which estimates this angle in a *view-independent* way, provided we consider the viewer and light source to be at infinity[1] [Bli77]. This approach is the default for directional lighting in OpenGL. The advantage is the estimated value is calculated only once for each light source and reused at each pixel in the frame. In addition to its computational efficiency, recent work has shown this model gives a better fit to reflectance data, as captured from real-world materials [NDM04].

Nevertheless, these models are limited to *local illumination* and fail to account for effects due to

[1] The viewer is placed at infinity for lighting purposes only.

global surface geometry, such as shadowing. The absence of a theoretical underpinning to these models precludes a natural extension to shadowing behaviour. Thus, specific techniques, such as shadow mapping, are needed to account for global lighting behaviour.

Shadow mapping

Shadow mapping or projective shadowing is a technique for producing scene shadowing that has been used extensively, since its introduction two decades ago [Wil78]. It is based on the use of a depth buffer with texture storage, which is now supported in hardware [SKvW+92] and accessed through an OpenGL extension [Pau02]. This technique is used widely in games because of its real-time performance. It has also found use in movie animation, such as Toy Story. However, it is only one of many different strategies for producing lighting in a scene and therefore it has some clear limitations and advantages over other techniques including shadow volumes [Cro77].

Shadow mapping meets an earlier requirement that an efficient query at the level of a pixel shader can decide the lighting characteristics of the matching pixel. It works by performing a test of whether a pixel is visible from the point of view of light sources in the scene. For each light source, a comparison is made to a matching depth image, which is stored in a texture in GPU memory. Specifically, a certain point in the depth image is queried and the value returned is compared to the distance between the light source and surface point represented by the pixel. If this depth image value is smaller, then the surface point is occluded by something closer to the light and the pixel is not lit by that light source. Figure 5.2 illustrates this idea.

Shadow mapping is advantageous in its ability to support other lighting models such as *phong shading* and specular highlighting without interfering with their quality. For example, specular highlights do not appear in the shadowed regions, even though shadow mapping does not concern itself with enforcing this.

However, while shadow mapping and the phong reflection model work well together, they both perform poorly for large granule counts. The next section, therefore, begins with the focus of this chapter, which is the acceleration of the aforementioned algorithms to meet the specific needs of granule visualisation and lighting.

5.2 Enhanced particle rendering

For a smooth three-dimensional object, a triangulated mesh is only a piecewise linear approximation to the actual surface. In addition, the linear components (triangular faces) that compose the object may each possess a normal. The latter is important in many lighting algorithms, where the surface normal for a point determines the amount of light the point reflects in the camera's direction. Thus, by increasing vertex count, we produce both a *tighter* and *smoother*

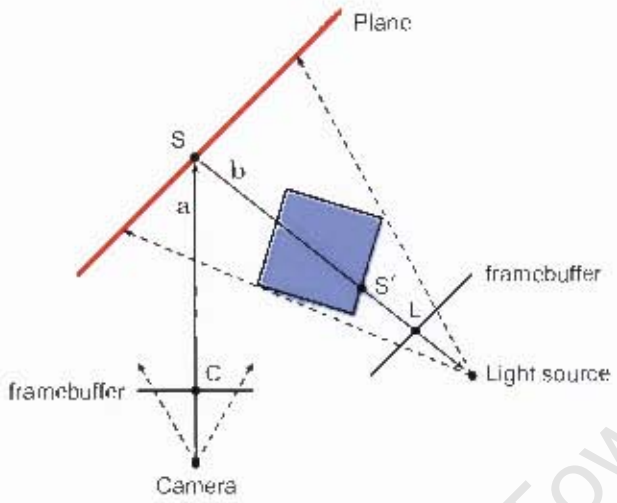


Figure 5.2: Shadow mapping algorithm. In this illustration, a plane and cube are present in the scene (as seen from above). To begin, the algorithm produces a framebuffer of scene depth values as determined by a light source's point of view. Since the cube is in front of the plane, relative to the light source, the plane area delimited by the ends of the dashed arrows (emanating from the light source) must be in shadow. Now, if we look at a position in the light's framebuffer L matching to a surface point on the cube S', we read values that are smaller than expected for the the matching plane surface point S. The next step is to render the scene into a colour framebuffer, as usual, from the camera's point of view. For a given framebuffer pixel C, matching to a surface point S, we can determine whether it is in shadow, by transforming the ray a in camera space into the ray b in light space. Since this gives the depth of S relative to the light source, we can compare this against the value stored at L. In this case, we find the transformed depth is larger than the value stored at L, indicating a closer and intervening surface point (S') exists between S an the light source, putting S in shadow.

approximation to the underlying surface.

This might suggest that lighting the smooth curved surface of a sphere requires a mesh with a large vertex count. However, the sphere is unique in that its centre alone determines the normal for any given point on its surface. In turn, all surface points are determined once the radius of the sphere is known as well. This implies that we can light a sphere given only its position and radius. However, the rendering pipeline must produce an image consisting of multiple fragments (or pixels) for each sphere, rather than a single fragment representing one vertex (the particle centre).

One way to solve this problem is to send a quad for each sphere, aligned to the viewer's direction, thus ensuring that sufficient fragments are available to form a spherical image. This technique is called *billboarding* and is illustrated in Figure 5.3.

However, four times the amount of geometry is being sent for one particle position. The solution is to exploit a recent extension that enables the rasterisation of points into size-varying squares [Cra03]. For this to work, we need the vertex shader to specify the size of the point to be rasterised, based on the particle's distance to the camera. The scaling factor to use for the sides

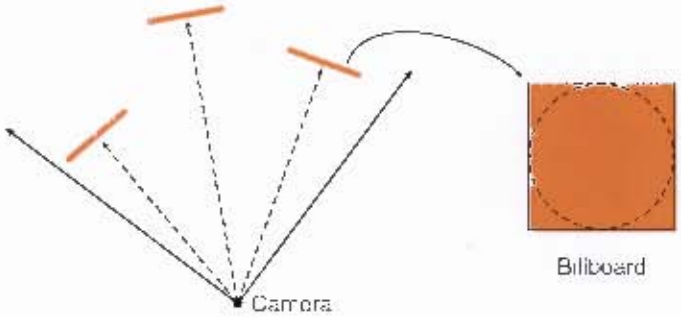


Figure 5.3: A billboarding approach to particle visualisation. For each particle, one square quad centred at the particle's position is rendered. A vertex shader must transform the four corners so that each quad is perpendicular to the view direction of the camera (dashed black lines). A fragment shader must then discard those pieces of the quad that fall outside of a circle centred within the quad. This creates the illusion of a sphere, as the camera moves, since the end result is always a circle, implying a spherical object in three dimensions.

of the square to achieve this effect is

$$s = \frac{w_h}{\tan\left(\frac{1}{2}F\frac{\pi}{180}\right)D} \tag{5.1}$$

where w_h is the height of the OpenGL output screen seen by the user. D is the distance to the viewer in the scene, and F is the viewer's field of vision. The latter can be visualised, in two dimensions, as the angle between the black arrows emanating from the camera's position in Figure 5.3.

Thus, the vertex shader, using equation 5.1, sets the appropriate point size as well as transforms the particle position to two-dimensional screen space (for rasterisation). The square produced from this information by the hardware after vertex processing is interpolated uniformly with a square set of texture coordinates in the range $[0, 1] \times [0, 1]$. Rasterisation converts each texture coordinate into a unique fragment, which in turn produces a matching fragment program instance. Each instance or shader receives information on both its position and the associated particle's position within screen space. Based on this information, if the fragment's coordinate lies more than a unit distance from the particle's centre, the shader will discards its output. For each unobscured particle, the collective result is a circular array of pixels in screen space. However, we go a step further and view this as a projection of a hemisphere onto screen space. This interpretation allows the fragment shader to calculate the height and normal of the hemisphere at the point directly above the fragment's two-dimensional coordinate.[2] Taking the previous two values relative to the particle's position in the 3D scene, the shader is free to apply the usual

[2]Alternatively, we might precalculate the heights and surface normals and store these values at matching coordinates in two textures, called a normal map and a nailboard [Sch97], respectively. Each fragment shader now reads a single texel from each texture to gain enough information to light its surface point. However, this has the disadvantage of coupling rendering to texture memory reads. Moreover, interpolation is necessary because *more* texture coordinates are needed for particles lying *closer* to the viewer and *vice versa*. Thus, given the simplicity and low computational cost, we recalculate the position and normal for each fragment.

lighting algorithm to calculate the matching pixel's colour. The collective result is a convincingly lit sphere at the particle's position in screen space. The specific algorithm followed is detailed in Pseudocode 5.1. The billboard resulting on screen from this process is illustrated in Figure 5.4.

Pseudocode 5.1 Point expansion algorithm

Vertex program:
  input: $\mathbf{v} \leftarrow$ vertex representing centre of a particle
  input: $r \leftarrow$ the particle's radius
  input: $s \leftarrow$ the point's scaling factor
  input: $c_s \leftarrow$ particle scaling factor
  $\mathbf{e} \leftarrow$ transform $\mathbf{v}$ into eye space, *without* projecting to screen space
  $\mathbf{v}' \leftarrow$ transform and project $\mathbf{v}$ to screen space
  PointSizeOutput $:= \frac{r \cdot s}{c_s \cdot \text{length}(\mathbf{e})}$
  PositionOutput $:= \mathbf{v}'$
Fragment program:
  input: $(s, t) \leftarrow$ texture coordinate generated by hardware
  $(s', t') \leftarrow (s, t) \cdot (2, -2) + (-1, 1)$ *//transform the coordinates to range* $[-1, 1] \times [-1, 1]$
  $r \leftarrow \sqrt{s'^2 + t'^2}$
  if $r > 1$ **then**
    discard fragment *//fragment not written to surface*
  end if
  $z \leftarrow \sqrt{1 - r^2}$ *//find the z component of the spheres surface point*
  ...centre (s', t') *and* z *together specify surface point, thus calculate lighting...*

Figure 5.4: Producing three-dimensional normals from the billboard. Using the billboard's centre, we can map each point of the billboard, corresponding to the position of a single fragment, up to a matching position on a hemisphere. This produces a normal that we can use in the Phong reflection model to produce the three-dimensional shading seen on the right.

One possible rendering problem is z-fighting [MB05], which can occur with closely overlapping geometry and especially when surfaces are flat and parallel to one another. The result appears as seemingly random pixels showing through the front surface with the colours of the ostensibly occluded surfaces behind. This is caused by the depth-buffer lacking the necessary precision to distinguish depth values of closely positioned geometry. Furthermore, as depth is view-dependent, the effect varies according to changes in the scene or viewpoint. Fortunately, given uniformly coloured spherical geometry, interpenetration artifacts are hard to spot. In addition, for this error to occur, particles would need to occupy almost the same position in space. Since forces become increasingly large with greater particle overlap and prevent particles from overlapping more than their radii, we do not consider z-fighting a problem.

Shadow mapping can now be trivially accelerated by simply rendering the circular regions once,

without any lighting being performed, to produce the depth image. This works because the light's position "sees", that is, the depth buffer records, only the closest particles based on the order of overlap. Importantly, while different pixels from the same particle's surface *necessarily* have different depths, this is irrelevant in our case, because particles cannot self-shadow themselves. Thus, we may ignore the three-dimensional geometry of *particles* and the granules they compose, however, we must render other scene components as proper three-dimensional objects, when constructing the depth image.

Higher quality lighting is also possible using ray tracing, though at the cost of real-time performance. The principles behind ray tracing are discussed later, in a related visualisation context (see Section 6.4). However, the next section briefly discusses the applicability of those methods to ray tracing a particle set.

5.3 Ray tracing

Ray tracing offers a high-quality, although computationally intensive alternative to rasterisation. In ray-tracing sand, the main problem is locating ray-particle intersections. The tracing algorithm, given a ray's origin and trajectory, must locate particles that lie in the ray's path. The simulation, however, stores particle positions in a two-dimensional texture without consideration for three-dimensional positioning in the simulation domain. Fortunately, we have already solved this problem by storing particle ids in a spatially meaningful way within a grid (as presented in Section 4.4). Reusing the grid data structure, we can adapt the tracing algorithm to query grid cells ahead of the ray, thus amassing candidate particle ids. The algorithm sorts the candidates by their distance to the ray, which means a positive collision test immediately identifies the closest intersected particle.

Note the ray-particle intersection test itself is fast. It simply requires checking if the ray passes within a radius of the particle's centre. The cost of determining grid occupancy, however, accrues as the ray progresses through the simulation domain. Moreover, the time taken by the slowest ray[3] to finish equals the minimum possible frame time. Thus slow rays lead to volatile frame rate as the viewpoint changes or the granular pile evolves. Rays that exit the domain without a collision may then intersect other objects in the scene or ultimately take on the scene's background colour. However, intersection testing with nongranular material, such as walls, needs different handling, which might not be amenable to acceleration. For these reason, we have avoided a ray-tracing approach to visualising the simulation.

[3]Such a ray, for example, might traverse through antipodal corners of the grid, performing collision tests at each step, but always failing to hit a particle.

5.4 Summary

Illuminating a large granular sandpile is computationally expensive, especially with naïve approaches. In this chapter, we demonstrated a method to accelerate the rendering of the thousands of granules. In particular, we used the underlying particle representation of granules, an implicit sphere model, and a recent OpenGL extension to produce both surface reflection behaviour and shadowing.

Chapter 6

Arbitrary rigid bodies

In Chapter 4, we described a GPU-based sand simulation able to produce real-time granular behaviour by directly modelling the mechanics of grains. The simulation algorithm we proposed, handled all physical interactions, be they among granules or between granules and arbitrary objects, using grain-grain collision physics. The underlying assumption, of course, is that we already have a *particulate* representation for each arbitrary (nongrain) object or surface used in the simulation.

For walls, floors, and other objects with simple geometric boundaries, we proposed using an *implicit* representation. However, for 3D geometry in general, finding a mathematical description for an implicit surface, one that admits efficient runtime particle sampling, is not straightforward. In these cases, explicit particle representations are needed. The difficulty, however, remains having to manually transform an arbitrary surface or mesh into its particle-sampled counterpart.

In this chapter, we make three contributions to automated mesh transformation. The first is to take a GPU-based technique [ED06] that converts an arbitrary closed triangle mesh to a discrete field representation and extend it to use modern GPU features. In particular, we show how the geometry stage of the GPU rendering pipeline allows offloading more of the algorithmic handling of triangles to the GPU. This results in less CPU-bound processing and faster field generation. The second contribution is a GPU-based particle sampler that simulates particle motion and repulsion inside the field. We demonstrate that injection of enough particles close to the mesh surface leads to a dense and even surface sampling. The final contribution is a *hybrid ray-casting-rendering* technique that simultaneously visualises both the field and the sampling process on that field. In addition, it uses information from rendering the samples to accelerate field visualisation. The purpose of the visualisation is to provide immediate feedback on the resulting particulate mesh representation without having to run a later sand simulation.

6.1 Background

The two fundamental problems we need to overcome are the generation of a signed-distance field and the particle-based sampling of an implicit surface represented in the field. Signed-distance fields have a wide variety of uses in computer graphics including multibody dynamics [GBF03], collision detection for cloth [BMF05, HK06] and polygons [SGG+07], and shadows [KN01]. For a survey see [EHK+04, JBS06]. We are interested here in using signed-distance to demarcate the region in which particle sampling will occur. In addition, the distance values form a gradient potential, which drives particles onto the implicit surface.

While we aim to achieve this entirely on the GPU, much work has already been done on serial processing algorithms and partial GPU implementations that still involve a large amount of CPU processing or preprocessing. We briefly touch on relevant previous work in these areas.

6.1.1 Signed-distance field

A distance field is a discrete scalar grid of voxels (much like the grid of cells discussed in a previous section) in which the model is placed and usually centered. In addition, each voxel (or cell) stores its distance to the closest geometric feature of the model placed in that field. Geometric features in this case could be a vertex, edge or face of polygon, whichever is closer.

Brute force

Given a mesh, a naïve approach to generating a distance field is by computing for each voxel the distance to the closest polygon in the mesh. In effect, this creates a scalar field in which points with zero value lie on the mesh and non-zero valued points represent the shortest distance to the mesh from that position. Figure 6.1 gives a two-dimensional representation of a distance field built by the naïve approach. Speeding up this approach may involve using bounded hierarchies of geometrical features to decrease search times when testing distances to find the minimum. A *signed*-distance field adds the requirement that for a closed mesh (one without any holes in the surface), the distances stored inside that mesh have negative sign, while those outside are positive.

The problem, however, is that we generally want to produce a high resolution grid. That is, ideally, the voxels representing the values 0 in the field should as closely as possible approximate the original mesh, which is achieved by using finer voxelisation. However, producing a high resolution grid means computing the values for many voxels and finding the closest surface point for each of those voxels. Even when we require values in the narrow-band around the model up to a distance n away, the problem is then finding those cells that distance away or less. One way to get around this problem is a change of perspective.

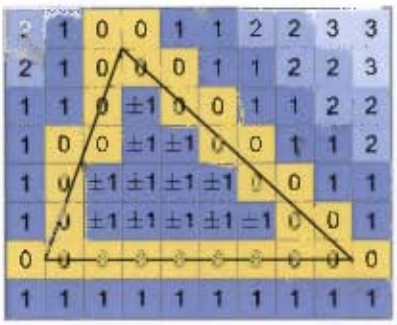


Figure 6.1: 2D discrete scalar distance fields. Considering only positive values, the diagram represents a distance field generated by the brute-force approach. Picking negative ± 1 values gives a signed-distance field.

Distance meshing

One solution is to consider the problem from the geometric perspective and compute distances away from the geometry. Specifically, following [HIKL+99], *distance functions* are used for vertex, edge, and polygon. These functions are typically hyperboloids or cones, which are meshed. These are rasterised, per slice of the field, into a depth buffer. Thus, at the end of one rasterisation pass, the depth buffers contains the *Voronoi regions* [dBvKO+00] for geometric features of model geometry of the model that lie in that slice. The downside of this method is its dependence on the detail of the distance function meshing, depth buffer resolution, and size of the meshes for the distance functions (where larger meshes means slower running times).

Scan conversion

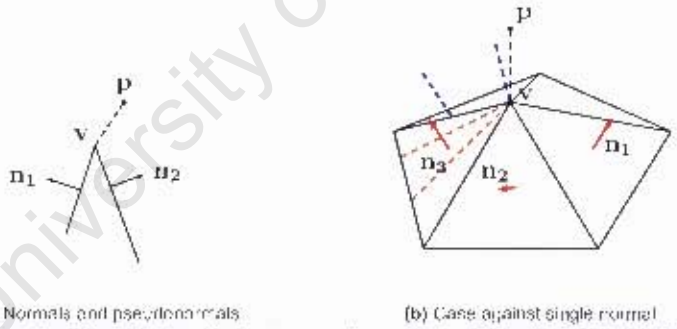
Another problem is that signs have not been computed. One way to handle this is using *scan conversion* which relies on Voronoi regions, but in this case their 3D counterparts. It involves scan-converting the geometric boundary region containing points that lie closer to a specific geometric feature on the model than to any other geometric feature. This is nothing more than 3D Voronoi regions with geometric features, as before, meaning each polygon's face, vertex, edge, or some combination of these. The overlap of these regions forms a thin shell around the model that scan-converts to a field around the model.

Scan conversion for the 3D case means performing a 2D scan conversion individually for slices of the 3D space. However, a two-pass algorithm is usually needed [DCB+04]. For one of these passes the input mesh (or the relevant subpart of it) is transformed into a collection of geometric boundary regions (forming the narrow-band thin shell), which are then converted by 2D scan conversion into voxels (instead of the 2D case where pixels are the output). For that layer, each voxel contains the shortest distance to the its closest geometric feature, which by definition it knows from the previous step (its a point inside a Voronoi region).

However, there are three sources of error that require a subsequent pass to alleviate. These are from planarity testing, bounded volume construction, and folds in the model polygons. These can lead to incorrect sign computations and, in some cases, incorrect distance values as well. In particular, the algorithm used in this thesis does not cater for folded polygons (instead we expect well-behaved meshes).

Beyond scan conversion

The problem of sign and distance error mentioned for scan conversion does not occur for a voxel positioned closest to exactly one face. For example, given a voxel at position $\mathbf{p}$ that is closest to a face with normal $\mathbf{n}$, then computing its distance *and* the correct sign is achieved easily with $(\mathbf{p} - \mathbf{v}) \cdot \mathbf{n}$. Since normals are undefined for vertices and edges there is no immediate way to calculate the correct sign for voxels closest to a vertex or edge. There are some ways we may try to get around this, however, there are special cases where naive approaches break down. These cases and their solution, the use of *pseudo-normals* [BA05, ED06], are illustrated and discussed in Figure 6.2.



(a) Normals and pseudo-normals (b) Case against single normal

Figure 6.2: An example of producing the incorrect sign when calculating "normals" for vertices and edges. In (a), a voxel at position $\mathbf{p}$ is equally far away from both faces incident at $\mathbf{v}$ (an edge viewed side on in 3D). If this were truly 2D geometry, then we could simply average the normals to produce a correctly signed result. This idea fails in the 3D case, as illustrated in (b). Now $\mathbf{p}$ is equally close to all five black triangles and we shall assume that $\mathbf{n}_3$ alone would produce the incorrect sign. We might try to mitigate the effect of such a vector by taking the average of the normals, giving each equal weighting. However, if we arbitrarily subdivide one of the faces as illustrated by the brown lines, then the incorrect normal dominates the average. Instead, the solution is to use the *angle* spanned by the corner of each incident face to weight its normal when calculating the average. The resulting normals (blue dashed lines) are known as *pseudo-normals*.

Once the signed-distance field has been constructed, we must sample the implicit surface represented in the discrete field. In particular, we favour methods that sample a surface using *particles* rather than dimensionless points. The next section discusses previous work in mesh sampling.

6.1.2 Particle sampling

In one sense, a model is already sampled by its vertices. That is, taking each vertex as a sample point, we place a particle there of large enough radius that the particles together form a closed surface, as illustrated in Figure 6.3. However, when vertices are sparse, particles are necessarily very large, thus distorting the models shape. This also does not suit the implementation in this thesis, since particles are assumed for grains and thus all rigid bodies to have a small and uniform size. For sufficient sampling, however, vertices may have irregular spacing due to irregular triangulation. One way to approach this problem is to resample the mesh.

Figure 6.3: Particle-sampled representation of geometry. The spherical mesh on the right must be approximated by a particulate rigid body (particle-sampled surface), as seen on the left. This allows the simulation framework to handle surface-granule interactions naturally, through particle-particle force calculations.

For example, a repulsion scheme where particles repel one another but move over the mesh surface may produce a uniform sampling [Tur91]. The problem with this approach is that it does not address the case of having large triangles and relatively small particles. When this occurs, then the final sampling interpolates the model and collectively reproduces the mesh's triangulated appearance. While this matches well to the mesh, the mesh itself is in many cases regarded as a piecewise linear approximation to the true object. Preferably, we would like to have our particles arranged uniformly across the surface *implied* by the mesh.

One extension of the previous idea treats the mesh as a sampled version of the implied object, to derive the surface that is implicit in this sampling [WH94]. Next, particles are constrained to move over the implicit surface, as constructed from the polygonal surface, as well as repel one another, leading to a uniform sampling. The disadvantage to this is that there is large computational cost involved in each time step, since an implicit surface fitting is needed. A signed-distance field (another implicit representation), on the other hand, is computed once and can be reused. This is only applicable to the static case, as addressed in this thesis. In addition, the signed-distance field is a discretization of an implicit surface and therefore not smooth in the same sense.

A special case, however, is when the surface has a known implicit equation. The same idea of physically-based distribution of particles over the implicit surfaces then applies but now using [dFdMT+92]. They scatter particles through three-dimensional space and have forces that pull them to the surface and forces the repel them so that they uniformly sample the surface. However, under certain conditions the differential equations for the process can be stiff, leading

to slow convergence or producing artifacts in the sampling.

In modelling nonreal-time granular material, [BYM05] propose using a simplified repulsion model of [WH94] but using a signed-distance field as the underlying implicit surface representation. This is the approach adopted in this thesis. However, we extend this by executing sampling entirely on the GPU allowing real-time visualisation of the sampling process.

6.1.3 Surface extraction

Before sampling occurs, however, particles must be placed in space. In particular, a physically-driven process requires that particles start within the *sub*volume of three-dimensional space occupied by the signed-distance field. Otherwise, the only active force is particle repulsion, without any surface attraction, leading to diffusion rather than sampling.

One option for initial particle placement is to situate particles directly on the original triangular faces and send these as the initial positions to the GPU. Since a uniform distribution applied independently to each flat triangular face does not equate to the final uniform sampling, a random initial sampling can be used [BYM05].

An alternative approach is to extract a surface from the signed-distance field and use this instead for the initial sampling positions before simulation-based sampling. One reason to do this is that the original mesh might not be uniformly triangulated, leading to irregular initial sampling. In particular, this may lead to the use of too few particles initially and a sparse final sampling. Another reason is that a low grid resolution distorts the implicit surface. Thus, extracting the new surface and sampling the result, might lead to better final uniform sample placement. Of course, this is a poor alternative, since it fails to sample the original surface closely.

This idea of isosurface extraction has been studied a great deal in computer graphics. One of the first algorithms for isosurface extraction is the *marching cubes* algorithm [LC87], which places surface elements in each voxel based on that voxel and surrounding voxel content. A key feature of this method is that geometric elements (polygons) in each voxel match up to form a contiguous surface. A problem with this method, however, is poor triangulation. This is handled by an extension known as *marching tetrahedra*, which produces near uniform triangulation and lower triangle count than marching cubes, under reasonable conditions [TPG99]. The tetrahedral method of Labelle et al. [LS07], in particular, *guarantees* topological and geometric accuracy under certain conditions, including a smooth isosurface of bounded curvature.

Many real-time GPU implementations exist for isosurface extraction [RDG+04, CHC+06, TSD07]. Most methods, however, are structured around producing real-time visualisation, which often entails handling only view-dependent extraction [CHC+06, TSD07] as opposed to extracting the entire object or surface. Nevertheless, trivial modification of these algorithms produce the entire extraction.

6.2 Field creation

As mentioned in the introduction, our method is based on work by [ED06]. Their algorithm can be broken into two stages. The first stage is the construction of a tetrahedral thin-shell. This shell encases the model in a band of user-specified thickness. In addition, this band is composed entirely of tetrahedrons. The band is constructed on the CPU. This works by taking each triangular face and fitting an oriented bounding box to it, as illustrated and described in Figure 6.4.

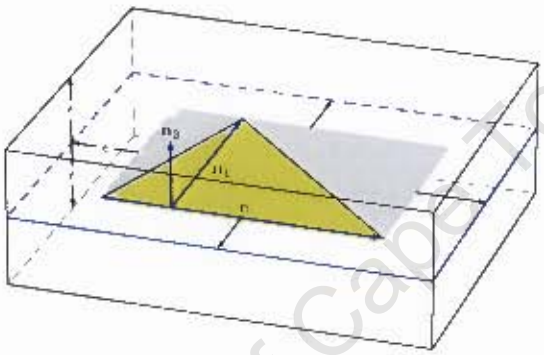


Figure 6.4: Fitting an oriented bounding box around a triangular face. The vector $\mathbf{e}$ represents the triangle's longest edge, while $\mathbf{n}_1$ is the height of the triangle measured orthogonal to $\mathbf{e}$. Finally, vector $\mathbf{n}_2$ is orthogonal to both previous vectors. Together these vectors form a local coordinate system for the triangle, namely, $(\mathbf{u}_e, \mathbf{u}_1, \mathbf{u}_2)$, which are the unit vectors of $\mathbf{e}$, $\mathbf{n}_1$, and $\mathbf{n}_2$, respectively. The close-fitting rectangle (gray) surrounds the triangle in the plane $(\mathbf{u}_1, \mathbf{u}_2)$. To generate the oriented bounding box, the rectangle is extended on all sides by some ϵ (indicated by the black arrows). Its four corners end up forming the corners of a larger rectangle (blue border). These new corners are used to make the eight corners of the box by ϵ-extension of the larger rectangle (blue border) in $\pm\mathbf{u}_2$ directions.

The value ϵ, introduced in Figure 6.4, represents the maximum distance away from the mesh *in one direction* the signed-distance field will reach. Thus, a user-specified narrow-band of thickness 2ϵ is built from all bounding boxes. However, as we shall later see, intersection is greatly simplified if we use a tetrahedron rather than a cube. Fortunately, every resulting box can always be decomposed into five tetrahedra as illustrated in Figure 6.5.

The process so far is summarised in Pseudocode 6.1.

Pseudocode 6.1 Create thin shell

```
CPU:
    L := empty list of tetrahedrons
    for all triangular faces F_i ∈ mesh do
        generate five tetrahedrons T_1, T_2, T_3, T_4, T_5 from F_i
        add all T_j to L
    end for
    sort(L) by increasing minimum z-value
```

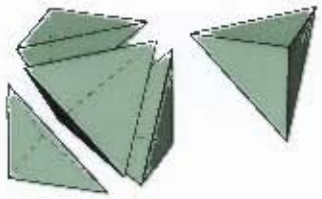
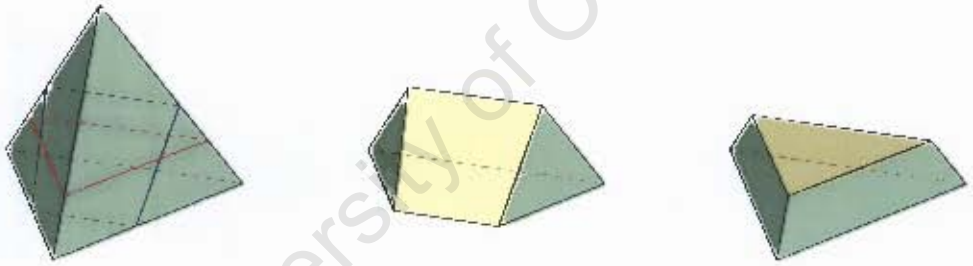


Figure 6.5: Cube dissected into five tetrahedrons. An analogous cuts produce five tetrahedrons for a rectangular faced bounding box, or parallelepipeds in general.

The next step involves filling exactly one slice of the signed-distance field with voxel-distance data. This is done by CPU-intersection testing, where specific planes are intersected with tetrahedrons to produce triangular cross-sections. Figure 6.6 illustrates the resulting triangle or quad intersections. The latter intersection is decomposed trivially into two triangles so that all intersection tests produce to triangle output. Degenerate intersections are discarded.



(a) Tetrahedron (b) Quadrangular slice (c) Triangular slice

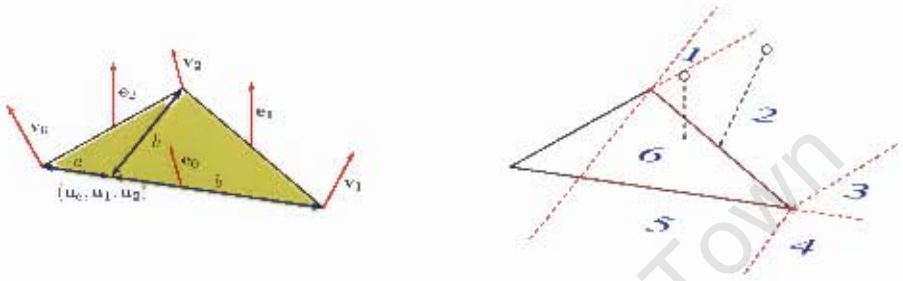
Figure 6.6: Planar slices through a tetrahedron. Three degenerate cases are not included here. These are plane-vertex, plane-edge, and plane-face intersections.

The planes used for the slicing are usually taken to be z-axis aligned, though conceptually, any slicing direction could be used. However, we use a cuboid space to make signed-distance field construction and particle-simulated sampling easier. This means axis-aligned slices use more of the texture surface for storage than for other directions, when large fields are generated.

Another advantage is that to carry out the slicing algorithmically, we simply sort the z-coordinate values for a tetrahedrons corners $\mathbf{p_0}$, $\mathbf{p_1}$, $\mathbf{p_2}$, and $\mathbf{p_3}$, with the z-value being tested. Depending upon where the z-value is positioned we can immediately establish which of the lines $\mathbf{p_i p_j}$ were intersected. The remaining work is to perform line-plane intersection testing.

The quad, however, is subdivided trivially into two triangles. This means that only triangles are produced. We shall refer to these as *intersection triangles* to distinguish them from the original triangle used for constructing the bounding box. Intersection triangles are sent to the GPU to be rasterised. Note that triangles produced by the intersection and their resulting fragments carry

information through the pipeline concerning the originating triangle. This is used by the fragment shader to calculate the shortest distance between that fragment (representing a voxel in the slice) and the originating triangle (given by variables passed along with the fragment). Figure 6.7 illustrates what variables are needed and how they are used for calculating the distance, as would be carried out by a fragment shader.[1]



(a) Variables available to a shader (b) Case handling

Figure 6.7: How a fragment shader computes distance and sign. Figure (a) shows original triangle data accessible to each intersection triangle and their resulting fragments, that is, shader instances. The values a, b, h (blue) are scalars representing the combined base length $a + b$ and the height h of the triangular face. The vectors $\mathbf{e}_i$ and $\mathbf{v}_j$ are the edge and vertex pseudonormals, respectively. Finally, $(\mathbf{u}_e, \mathbf{u}_1, \mathbf{u}_2)$ is the local coordinate system situated as described for the bounding box construction. (b) The coordinate system available to the fragment shader, as shown in (a), divides three-dimensional space into six regions as shown. The region in which the particle lies does effect the feature used for finding the minimum distance (black dashed line). Given that feature, the associated pseudonormal by comparison to (a) will be used to determine the sign.

The entire process, from taking an initial triangular face from a model to converting this into fragment program output to be stored, is summarised in Pseudocode 6.2.

Pseudocode 6.2 Scan-conversion of tetrahedrons

```
CPU:
    L ← list of all tetrahedrons T_i
    for z := z_min to z_max do
        for all T_i ⊂ L where T_i ∩ z ≠ ∅ do
            find triangular cross-sections t_1, t_2 from z ∩ T_i    //two when quad intersection
            render(t_1, t_2)    //note that GPU only used at this stage
        end for
    end for
GPU:
    Fragment program:
        Compute shortest signed distance s from this fragment to the original triangle
        ColourOutput :- (d, s)    //3-vector (pseudo)normal and scalar signed distance
        DepthOutput :- |s|    //unsigned magnitude
```

Notice that the surface (pseudo)normal (3-vector) is stored as well as the signed distance. Using

[1] This algorithm is possible because geometry and vertex programs are able to pass along additional information about triangles they operate on. This information is available to all the fragments and associated fragment program instances produces when a specific triangle is rasterised. In GLSL, for example, such variables may be referred to as *flat* (as opposed to 'interpolated'), which means that each fragment shader spawned by the rasterised triangle receives the *same* information sent by originating vertex or geometry shader.

a 4-vector texel allows us to store the 3-vector and the scalar value together for each matching voxel, thus providing particles in the sampling process both direction and distance information.

In addition, the unsigned magnitude is written to the depth buffer. This must occur, since oriented bounding boxes generated earlier for each face must overlap, since the mesh is closed and faces are adjacent. Overlapping boxes, lead to overlapping tetrahedrons and multiple fragments being generated for the same voxel. Thus, by setting the depth buffer to an initial maximum value and the depth test to pass only lesser depth fragments, the final signed-distance texels will hold the correct values for their matching voxels. Notice that with increasingly large values of ϵ (see Figure 6.4), more overlapping fragments are produced, leading to more wasted calculation, since only one of the competing fragments produces the correct values. On the other hand, ϵ must be large enough to ensure that the shell is computed correctly for voxels closest to edges and vertices.

6.2.1 Enhanced GPU algorithm

The previous method handles the expensive distance calculation simultaneously for all voxels by parallel execution on the GPU. However, this is not the only expensive part of the process. The expansion of triangular faces to oriented bounding boxes (see Figure 6.4) and subsequent intersection testing (see Figure 6.6) involve significant computation as well. Importantly, these computations occur independently for each triangular face suggesting a possible avenue for parallelisation.

A natural question is: can we implement this on the GPU? The answer is yes, if two problems can be solved. This first problem is that the number of intersection triangles generated for each face is not known ahead of time, unless significant parts of the computation are run as a preprocessing step on the CPU, defeating the purpose of parallel GPU performance. Intersection count is necessary, because a vertex program can only transform triangle information sent to it. That is, we can load data for each triangular face (into a texture, for example), but we must send the maximum number of possible intersection triangles for each face. For valid intersections (perhaps associated with particular tetrahedron positions for the bounding box and marked by an id value asserting its validity), the vertex program performs a transformation sending the vertex to its correct position. Thus, the entire bounding box calculation is run three times for each valid intersection triangle.

The second problem is that this must be performed for *each* slice of the space holding the signed-distance field, since 3D textures can only be rendered to a slice at a time, among other limitations (as discussed previously in Section 4.4.2).

This thesis solves these two problems. The first is handled by exploiting the new geometry stage. This stage of the rendering pipeline processes an entire geometric primitive for each invocation, such as an entire triangular face, which it may transform using the same operations available to the vertex stage. Secondly, the geometry stage can emit zero, one, or several primitives for a

single input primitive invocation. That is, for each triangular face sent to this stage, multiple intersection triangles may leave.

However, this still will involve multiple computations for each triangular face, depending on how many planes it intersects. This problem is solved by using a flattened 3D texture similar in design to the flattened grid structure used for collision detection in section 4.4. However, instead of storing particle positions, we store distance and sign-values. The geometry stage now simply outputs all possible intersections. A vertex shader, similar to the one used for collision detection, transforms slices so that they are rasterised to correct layer represented in the grid.

The result is that a single rendering pass using the triangular faces alone suffices to construct the entire signed-distance field. The process is given in Pseudocode 6.3.

Pseudocode 6.3 Enhanced scan-conversion

```
CPU:
    for all triangular faces F_i ∈ triangulated mesh do
        render(F_i)    //GPU called here
    end for
GPU:
    Geometry program:
        input: face F
        generate parallelepiped P from F
        decompose P into five tetrahedrons T_1, T_2, T_3, T_4, T_5
        for all T_i do
            for z := argmin_z(z ∩ T_i ≠ ∅) to argmax_z(z ∩ T_i ≠ ∅) do
                find triangular cross-sections t_1, t_2 from z ∩ T_i    //two when quad intersection
                emit(t_1, t_2)    //passed down pipe for rasterisation
            end for
        end for
    Vertex program:
        Input: Vertex v_j of an intersection triangle t
        transform v_j from 3D space to 2D texture coordinate (s, t)
        PositionOutput := (s, t, 0)    //orthogonal projection onto texture surface
    Fragment program:
        ...same as for previous version...
```

With the signed-distance field constructed, we now turn to looking at the sampling processes performed on the field.

6.3 Sampling

The idea for sampling is to allow the physical repulsion between particles and the gradient of the signed-distance field to drive the sampling process, as illustrated in Figure 6.8. To produce this behaviour we need to mathematically determine how to move the particles. Based on the method of Witkin et al. [WH94], Bell et al. [BYM05] have proposed a simpler version of the particle velocity model. This approach ignores force action and instead estimates the final particle

velocities that might result from such forces. Specifically, a particle's velocity v is the sum of the velocity v_f resulting from the gradient in the signed-distance field acting on the particle and the velocity v_r that arises from repulsion between particles. Thus, we have $v = v_f - v_r$.

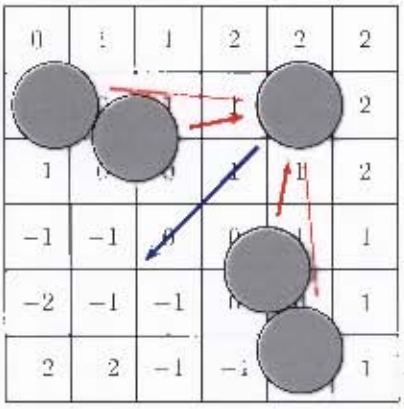


Figure 6.8: A particle moving in the signed-distance field. The outer particle is driven by a force (blue) that operates along the gradient of the signed-distance field from high to low *absolute* values. This directs the particle towards the implicit surface represented by zeros. The particle also experiences an additive repulsive force (red) that drives it away from other particles. The latter force comprises the repelling action of all bordering particles. Moreover, repulsion falls off with increasing distance between particles. This implies the need for a fine balance between the strength of the field and strength of bordering particle influence.

Using a velocity-based model, the first velocity component involves particles moving towards the implicit surface, as represented by voxels with signed-distance values of $(\mathbf{0}, 0)$, that is, a 3-vector surface "normal" (in this case, effectively no normal) and a scalar signed distance. For a particle at position $\mathbf{p}$, the field velocity towards the implicit surface is given by

$$v_f = -s(\mathbf{p})\mathbf{d}(\mathbf{p}) \tag{6.1}$$

where s is the signed distance stored at the voxel containing the particle and $\mathbf{d}$ is the surface normal for the unspecified surface point[2] closest to the that voxel. The outcome is that particles lying in positively and negatively signed voxels move towards the zero voxels that collectively estimate the geometric surface.

Finally, we ensure particles distribute themselves uniformly by applying a repulsion velocity given by

$$v_r = \sum_{p_i \in \mathbf{S} \setminus p} \phi(||p - p_i||) \frac{p - p_i}{||p - p_i||}, \tag{6.2}$$

where p is the particle whose velocity we are computing, S is the set of all particles with positions in the compact support of a *kernel function* ϕ. In particular, ϕ should be a real-valued function that vanishes at infinity. One natural kernel is the *Gaussian function*. Indeed, this thesis employs the Gaussian function, but deviates from Bell et al. [BYM05] by approximating the compact

[2]The surface point itself is not stored, nor do we need it, as the normal alone determines the particle's trajectory.

support using a cut-off distance. That is, while ϕ is nonzero for all particle positions, we only consider particles lying in the 27 voxels adjacent to and including the voxel containing the particle with position p. To make this approach valid, we simply ensure that ϕ is sufficiently small for distances of more than r, where r is twice the diagonal distance between antipodal corners of a cubic voxel. The appropriate Gaussian function and its derivation can be found in Appendix A.2.

6.3.1 GPU Algorithm

The GPU-based algorithm needs two position and two velocity textures corresponding in size, so that matching texel coordinates store properties related to the same particle. In addition, a single VBO stores a set of 2-vector texture coordinates that collectively would cover any of the aforementioned textures. Section 4.4.3 describes the same grid update process as used here.

After updating the grid, another rendering pass produces corresponding fragments for each texel position. The fragment instances, representing individual particles, read the grid texture to accumulate bordering particle influence in a manner analogous to the neighbourhood search described in Section 4.4.4. The same fragment instance also reads from the signed-distance field texture, which is created once off, as previously described in Section 6.2. The values it accumulates here effect the particle's velocity as well. The final velocity is then multiplied by a time-step and added to the old particle position to give its new position. Choosing the correct time step as well as other properties, such as particle radius, is discussed briefly in the next section. The procedure followed by the algorithm is given in Pseudocode 6.4.

Notice that signed-distance values are interpolated in the fragment program. The method we employ is called *trilinear interpolation* and is illustrated in Figure 6.9. It calculates the normal direction and signed distance for *all* positions in the field, not just the voxel centres, as stored in the texture.

In the cases where hardware does not support trilinear interpolation directly, we can calculate this directly, albeit at significant computational cost. Given the eight texel values T_{xyz} located at the centre of cells C_{xyz}, we can calculate the interpolated value I at position (x, y, z) as

$$\begin{aligned} I(x,y,z) = {} & T_{000}(1-x)(1-y)(1-z) \\ & + T_{100}x(1-y)(1-z) \\ & + T_{010}(1-x)y(1-z) \\ & + T_{001}(1-x)(1-y)z \\ & + T_{101}x(1-y)z \\ & + T_{011}(1-x)yz \\ & + T_{110}xy(1-z) \\ & + T_{111}xyz \end{aligned} \tag{6.3}$$

Pseudocode 6.4 Enhanced particle sampling

```
CPU:
    set two render targets to one particle texture and one velocity texture
    activate shader program for updating particle position and velocity
    attach to texture units: input position, velocity, signed-distance field, and grid textures
    render one quad with texture coordinates mapping fragment positions to matching texels
    deactivate shader program
    reset render targets to one for output to screen
GPU:
    Fragment program:
        p ← read position texture using fragment texture coordinate
        v ← read velocity texture using fragment texture coordinate
        p' ← 0
        v' ← 0
        for all 27 voxels w_i do
            read grid texture at coordinate matched to w_i to get particle ids j_1, j_2, j_3, j_4
            for all particle ids j do
                p_j ← read particle position texture at the coordinate matching id j
                add to v' the velocity-based influence of p_j on position p
            end for
        end for
        (d_0, s_0), (d_1, s_1), ..., (d_7, s_7) ← read 8 closest texels to p from field texture
        (d, s) ← perform trilinear interpolation on (d_i, s_i) at p
        add to v' the velocity-based effect of the field value (d, s)
        p' = p + v'δt    //where δt is the time step
        ColourOutput[0] := p'    //stored to render target 0 (position texture)
        ColourOutput[1] := v'    //stored to render target 1 (velocity texture)
```

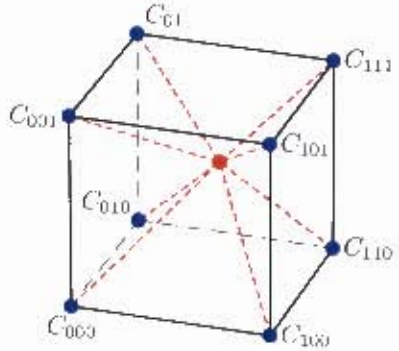


Figure 6.9: Trilinear interpolation for a cube. A sample point $\mathbf{p}$ (red) dot is chosen in 3D space. Actual voxel centers are at C_{xyz}. Unlike the collision detection grid texture, the signed-distance field texture does not use the texel value to represent the entire contents of the voxel. The texel value $(\mathbf{d}, s)$ (normal direction, scalar distance) represents these values at the centre of the voxel. Therefore, we must interpolate to find the matching $(\mathbf{d}, s)$ value, even if $\mathbf{p}$ is enclosed in a single voxel.

Notice we assume a cubic arrangement with unit side length. For the general case, we would simply scale the arrangment before performing Equation 6.3.

However, the advantage of using a 3D texture over the flattened 3D texture is having fast hardware-supported trilinear interpolation available. This method needs to read the eight texture values corresponding to the centres of eight *bordering* voxels. If these values are stored in a 3D texture, then due to their spatial locality, texture prefetching will accelerate reading of all these values. In contrast, using a flattened 3D texture, only 2D caching for each set of four adjacent voxels is possible.[3] In addition, the shader must perform the interpolation calculation explicitly, since hardware lacks native support for flattened 3D textures.

Specifying time step size, particle radius, and particle count

There is no clear guideline to choosing a "good" time step. However, we might presume that a particle travels no more than a cell's breadth during a single time step. This is justified by the same stability reasons given for particle movement in the sand simulation (see Section 3.3.1). Using this idea, we calculate the magnitude of the maximum velocity that could act on a particle and divide this into the smallest side length of a cell, thus yielding a reasonable time step.

However, calculating the magnitude of the maximum velocity is not straightforward. We may derive an estimate by recalling that this velocity is necessarily produced by summing two other velocities. The first is the velocity contributed by the signed-distance field. In this case, we seek the maximum scalar value occurring in the field. We can find this value by copying the completed signed-distance field from the GPU to the CPU and searching for the maximum absolute value.[4] We shall assume this velocity acts in the same direction as the second velocity contribution, namely the velocity due to repulsion from neighbouring particles. To calculate the latter, we assume the existence of enough bordering particles to fill the nine signed-distance field cells lying to one side of the particle's cell.[5] This allows us to calculate the maximum repulsive velocity pushing the particle in a single direction. Adding the magnitude of this result to the signed-distance field result gives the final magnitude. Importantly, this value overestimates maximum velocity, since the field and repulsive forces typically act against one another, as in the case of a single particle pushing others aside while the field pushes it towards the implicit surface.

Choosing a "good" particle radius is another problem, which is complicated by grid cells storing at most four particle ids (due to a 4-vector texel limit). In the sand simulation, we set the particle radius so at most four particles fit into a cell and, in addition, we use both collision detection and repulsion *forces* to keep interpenetration to a minimum. In the *velocity* model, however, we cannot

[3]The one advantage to 2D caching is that the cache is less likely to be saturated when many particle positions are interpolated in the signed-distance field.

[4]We might calculate the maximum scalar value *before* producing the signed-distance field by recalling that a user sets the shell size for the field produced around the model. Assuming cubic field cells of unit side length, we use the fact that shell size limits the largest extent of the field. This limit is an estimate for the maximum distance value we seek. This value is accurate for face distances, but may slightly underestimate corner distances, since these arise at the intersection of the bounding boxes forming the shell.

[5]The nine cells form one face of a cube of 27 cells in the signed-distance field.

easily prevent interpenetration, which otherwise leads to more than four particles occupying a voxel. When this occurs, a grid query will only detect the particles with the lowest four ids, leaving the other higher-id particles "invisible" to neighbours.

All is not lost, since the undetected particles still experience a repulsion force induced by *their* neighbours, namely the same-cell lower-id particles and those in bordering cells. This allows for later separation when compressive behaviour might have eased. However, tighter packing induces a much larger repulsion velocity, which might overcome the field gradient resistance. This leads to particles rising off the surface, against the gradient. To address this, we might use a different or modified kernel function that increases the weighting of distant particles. This is an attempt to produce a greater total repulsion velocity, which may inhibit particle clumping in the first place. A second approach is to use a thinner shell and to set all field cells to a special "empty space" value, before starting field generation.[6] Now, the sampling process, specifically the fragment shader, can detect when a specific particle has pushed into an unsigned region of the field. The particle's interaction with other particles is ignored and the particle itself is repositioned into a special unoccupied corner cell of the domain, such as the origin cell. Of course, this now repeats indefinitely, effectively removing that particle from the sampling process. Importantly, storing the final model now requires that we consider only those particles not lying within the designated "waste" cell.

The latter provides an easy mechanism to rid the sampling of excess particles. However, we still need to address choosing a particle count large enough to sample the implicit surface without holes. Reusing the idea of copying the signed-distance field from the GPU to CPU, we can count in CPU-readable memory the quantity of field cells with distance zero. This value, multiplied by four, gives an estimate of the maximum number of particles we could use. Thus, the problem of achieving a finer sampling, is now handled by choosing higher resolution grids for the signed-distance field and collision detection. The only limit to this approach is the maximum 3D texture size available on the GPU.

Finally, we note that particles can be injected into the sampler at runtime. This involves ensuring the position and velocity textures are large enough to accommodate any additional particles. We proceed by first copying the texture data from the GPU to the CPU and then insert additional particles with unique ids.[7] Similar to initialisation at startup, the final texture data is uploaded to the GPU. The simulation then proceeds in the next time step with the additional particles appearing instantaneously in their assigned spatial positions. This procedure provides a way to position particles into gaps where repulsion and field-gradient velocity have failed to move them. However, this thesis does not address algorithmically locating such gaps.

We now turn to visualising the field and sampling process.

[6] For example, we might set all field cells to $((1,0,0),0)$. This value is not a valid result of field generation, since zero-valued cells *are* the implicit surface and thus have no associated normal. Of course, field generation will overwrite these default values where appropriate.

[7] Instead of using only unoccupied texels, we may reuse the texel storage and ids of "waste" particles that have moved irretrievably into empty field space.

6.4 Visualisation

Field visualisation calls for methods such as those mentioned in Section 6.1.3, which were found to be unsatisfactory for implicit surface *extraction*. Here, however, we are no longer concerned with geometry *per se*, but rather with creating a visual representation of geometry. The latter distinction is important, since it informs many rendering tactics, such as culling geometry not visible to the user.

The technique of throwing away unseen geometry is supported in hardware through depth buffering (see Section 4.1.3). This allows the particle visualisation to mitigate unnecessary fragment calculation for nonvisible particles. For field visualisation, the same particle depth buffer is exploited with an additional culling technique to accelerate field visualisation.

6.4.1 Particles

Particle visualisation proceeds in the same way as described in Chapter 5, except that shadow mapping is unnecessary. However, a first rendering pass with no lighting needs to be performed so that the depth buffer can be saved. The depth image is generated from the camera's viewpoint, which allows for optimised field visualisation as explained in the next section. The code for creating the depth texture containing the depth image is given in Pseudocode 6.5. The code listing lacks vertex and fragment program code, since these are the same as for sand visualisation, except that no specular, diffuse, or shadow mapped lighting is computed.

Pseudocode 6.5 Particle depth storage

```
CPU:
    activate point-sprite expansion
    activate FBO associated with particle textures
    unset FBO render targets    //no colour output is captured by FBO
    set FBO depth buffer to point to a depth texture
    activate shader program for point to particle expansion without lighting
    render the VBO that stores particle positions
    deactivate shader program
    reset render targets to include colour surface output
    deactivate FBO
    deactivate point-sprite expansion
```

Once the depth texture is stored, particles are rendered again, without shadow mapping, to the final colour surface. Diffuse lighting may be used to provide the user with a visual aid to particle positioning in the field (relative to the light direction), though this is not strictly required. The next step is to "fill" in the gaps between particles, through which the signed-distance field is visible.

An example of the visualisation, during the sampling of a trefoil model, is given in Figure 6.10. Here, we see the progression from an initially poorly sampled surface to a fully covered surface.

Figure 6.10: Visualisation of the sampling process running on a trefoil model.

6.4.2 Signed-distance field

For scalar field visualisation, we might apply the surface extraction methods discussed in Section 6.1.3. These methods bring out the latent surface by affixing geometry to the zero distance values in the field. In particular, they perform this in a limited view-dependent way, so that effort is expended only on what the user might see, rather than extracting all geometry unnecessarily. The triangulated geometry that is extracted can be visualised by rasterisation to the screen. Implementing this in real-time is straightforward and needs at most a single vertex shader [GJJ05].

An alternative to rasterisation that bypasses geometry production is to "colour in" the final screen directly. One such method, called *ray casting*, shoots or casts a ray out from the camera's viewpoint through each framebuffer (screen) pixel. A ray may strike a geometric surface in the scene or pass by without collision. When a collision occurs, the pixel associated with the colliding ray is assigned the colour of the surface point intersected by that ray. When rays miss all scene geometry, they are assigned a default background colour. The resulting framebuffer image, much like a photograph, captures the rays of light radiating from the part of the scene it sees.

A similar approach is used for visualising the cuboid of voxels containing the signed-distance field on the GPU. In particular, a screen-sized quadrilateral is rasterised to the framebuffer, resulting in the production of fragment shader instances for each screen pixel. Each fragment carries out ray intersection testing against the voxelised cuboid. Again, rays are modeled as if they originate at the camera position and travel in the camera's view direction to pass through their matching pixels out into the scene.

This thesis employs a ray casting technique to validate signed-distance field production and aid the user in judging the sampler's correctness. However, while ray casting is simple to implement, geometry extraction is attractive because it provides the user with a model of the implicit surface that admits a natural comparison to the initial model. The problem is these techniques often employ *different* geometric primitives and algorithms to extract the surface, which leads to different model output. This would mean choosing the "best" technique, which may differ according to input model. Ray casting ensures we see the same underlying signed-

distance field. Furthermore, we add nothing new to the discussion by addressing GPU-based geometric extraction, which has already received much attention in the literature.

Unlike geometric ray casting, however, we do not color a fragment's pixel according to the intersection point. Instead, for each pixel, the associated fragment shader instance calculates the path taken by fragment's ray through the voxelised cuboid. When an occupied voxel is struck, its contents indicate the presence of the field, as illustrated in Figure 6.11. The pixel value is coloured an appropriate nonbackground colour. In contrast, if the ray passes through the cuboid without striking an occupied voxel, then its pixel is set to the background color. Voxels are only considered to be occupied if they contain a zero-distance value. The result, therefore, is an image of the implicit surface in the framebuffer and ultimately on screen.

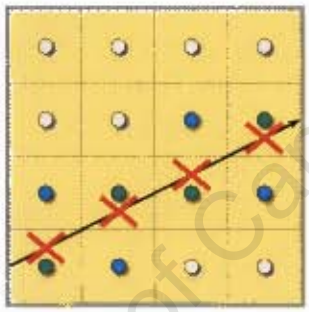


Figure 6.11: Progressive sampling of texels by a ray. The ray L (black line) is cast in a z-plane through the cuboid, thereby sampling one layer of voxels (yellow squares) by reading their associated texture values (green dots). Since no intersection testing is performed inside the cuboid, the ray relies on regularly spaced samples (red crosses) to know when it is ins de the next voxel. This means some of the intersected voxels and their texel values are missed (blue dots). This is not an issue since at least one sample is taken, terminating the ray as needed. A related prob'em occurs when samples fall on a corner, edge, or face shared by adjacent voxels. Th's is handled by the usual arbitration schemes, such as picking the voxel lying closest to the origin.

Exploiting particle positioning

Notice that we can simply render the particles over the image that results from ray casting. However, this means wasting fragment calculation for those pixels that are covered by subsequent particle pixels. A better approach is to render the particles first, as assumed in the particle visualisation, and use the depth information to avoid unnecessary fragment calculation. The algorithm for ray casting combined with depth checking is described in Pseudocode 6.6.

The code renders only four vertices, that is the four corners of the quad. However, direction for each ray is stored as a **varying** shader variable, which means that when the quad is rasterised, direction is linearly interpolated across the quad's surface, thus producing individual direction rays for each resulting fragment.

Pseudocode 6.6 Ray casting

CPU:
  activate shader program for ray casting
  attach to texture units: signed-distance field texture and particle depth texture
  render a quad with texture coordinates mapping one-to-one to framebuffer pixels
  deactivate shader program
GPU:
  Vertex program:
    Input: untransformed vertex **p**
    Define: **r** as a **varying** variable *//value is interpolated across a triangle*
    Define: **o** as a **flat** variable *//value is unchanged across a triangle*
    $\mathbf{o} \leftarrow M^{-1} \cdot (0,0,0,1)$ *//finds camera's position using modelview matrix M*
    $\mathbf{r} \leftarrow$ **normalise**$(\mathbf{p} - \mathbf{o})$ *//cameras view direction to vertex*
    Output position := project vertex **p** to framebuffer surface
  Fragment program:
    Input: $\mathbf{o} \leftarrow$ camera origin
    Input: $\mathbf{r} \leftarrow$ interpolated direction
    $d \leftarrow$ read the depth texture using this fragment's texture coordinates
    if d altered from default depth value **then**
      discard the fragment *//no further calculation and pixel not written*
    end if
    $s \leftarrow \frac{1}{n_{\max}}$ *//step interval size*
    $(t_n, t_f) \leftarrow$ calculate near and far intersection distances, respectively
    $L \leftarrow (t_n + \frac{c}{2}) \cdot \mathbf{r} + \mathbf{o}$ *//line L at a position half cell width into cuboid*
    for $i \leftarrow 1, n_s$ **do**
      ...
      $(u, v) \leftarrow$ convert (t_n, t_f) to texture coordinates for signed-distance field
    end for

Enhanced GPU algorithm

A problem with the previous approach is that if the cuboid does not take up much of the screen real estate, then many ray-cuboid intersection misses result, leading to wasted fragment calculation. In contrast, the ray casting approach used in this thesis produces exactly those fragment instances whose corresponding rays do intersect the signed-distance cuboid, saving unnecessary fragment calculation.

The method begins by rasterising the camera's view of the signed-distance cuboid with the usual perspective projection. That is, we render a polygon model of a cuboid that has rectangular faces of the same dimensions as the *abstract* cuboid represented within the signed-distance field texture. The fragments resulting from rasterisation indicate *exactly* which pixels produce a ray intersecting the cuboid. Therefore, no ray calculation is wasted in this approach, as illustrated in Figure 6.12.

The code required for this is the same as in Pseudocode 6.6, except that the cube is rendered instead of a quad. An example final visualisation is given in Figure 6.13, which views one frame of the sampling process either with the signed-distance field alone or with particle sampling included.

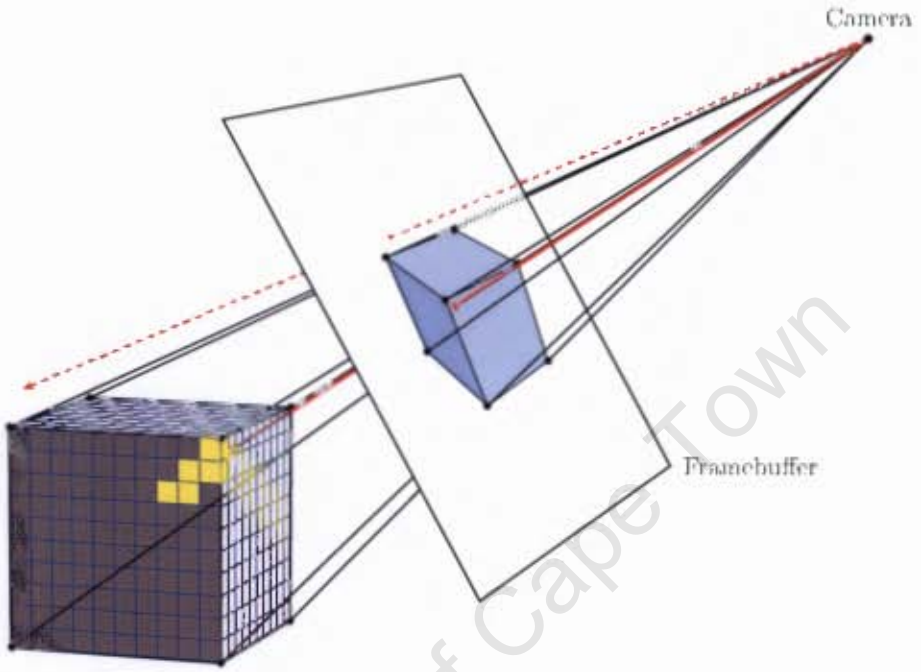


Figure 6.12: Ray casting through voxelised cuboid containing the signed distance field. Ignoring the implied subdivision of the large cuboid (blue lines), we observe the initial process where the cuboid is rendered (projected) onto the framebuffer as seen from the camera viewpoint. Thus, "coloured" pixels (the intersection of the red vector with the framebuffer) indicate that a ray (solid red vector) cast from the camera viewpoint through that pixel would strike the cuboid. A white pixel indicates background, implying its associated ray (dashed red vector) does not intersect the cuboid. The path of the intersecting ray, however, is followed voxel-by-voxel (yellow cubes) until the ray exits the cuboid or strikes a non-empty voxel. In the latter case, the pixel corresponding to the ray can now be coloured based on the voxel content.

Figure 6.13: Visualisation of the sampling process running on a trefoil model. The right image shows the signed-distance field alone and the left image shows the sampling process acting on the field.

6.5 Summary

In this chapter, we demonstrated construction of a signed-distance field in a single rendering pass, that is, where the entire triangulated mesh is sent *once* to the GPU. In particular, we use the new geometry processing stage (found on modern GPUs) to produce intersection triangles *within* the rendering pipeline, thereby avoiding having to resend triangle data multiple times. We further support this processing by employing a *flattened* 3D texture. This enables the storage of plane-intersection data (fragments) from *any* z-plane, immediately, rather than on separate CPU-issued passes (as would be required with a 3D texture). Finally, we demonstrated a real-time technique for visualising simultaneously both the particle sampling process and the underlying signed-distance field. In addition, optimisations for mitigating unnecessary shader computation when producing the visualisation were discussed.

Part III

Results and Discussion

Chapter 7

Simulation framework

The central claim of this thesis is that our framework for GPU-based sand simulation produces *valid* sand behaviour *faster* than previous CPU-based approaches, while matching or outperforming related simulation on the GPU. In this chapter, we provide support for this claim.

We begin by corroborating *physical validity*, which entails analysing, under varying conditions, the measurable static and dynamic features of the simulated material. The features we address and the test cases we perform are selected because of their ubiquity in the granular material science literature. The results of our testing are in line with previous DEM simulations. In one instance, the sand framework outperforms previous single and dual-particle models, in providing a superior match to physical experiment (see Section 7.2.1).

Next, we analyse framework *performance*, focusing on the *quantity* of sand simulated and the simulation *speed*. Results are compared to previous DEM simulations, including multiparticle models, where a multicore CPU-based machine or the same GPU has been used. We address differences in previous experimental conditions by normalising results to reflect per-granule performance. Lastly, we assess the effect of high-quality lighting on simulation performance, for which we find only a marginal decrease in rendering time.

7.1 Testing environment

All algorithms and methods described in this thesis have been implemented in C++. The sand simulation framework compiles with GNU GCC 4.1 and executes under GNU/Linux (Ubuntu 7.04). However, the *instrumented* driver for the graphics card, which provides extensive real-time information on GPU performance, failed to work under GNU/Linux. Therefore, testing has been performed under Windows Vista^{TM} (32-bit), using the Microsoft Visual Studio 2005 IDE for compilation.

Single-precision floating-point values are used throughout the framework, in shaders, textures,

and the CPU-based orchestration. This restriction ensures that numerical data remains consistent between the CPU and the single-precision floating-point processors on the GPU. While methods exist to emulate double-precision on GPUs lacking native support [GST05], the expected computational throughput is about half that achieved when running in single precision mode [GST07, GS08].

Testing hardware consisted of an Intel(R) Core(TM)2 6600 (clock speed of 2.40GHz) with 2048 MB of RAM and an EVGA GeForce 8800 GTX/PCI/SSE2 graphics card with 768 MB of dedicated memory. To support the latter, we installed the NVIDIA display driver, version 181.20. For shader orchestration and rendering, we used OpenGL 2.1, along with the OpenGL Utility Toolkit (GLUT) 3.7, to handle windowing and user input. The OpenGL Extension Wrangler Library (GLEW) 1.5.1, provided access to special features of the graphics card that are not natively supported by OpenGL 2.1. Finally, we used the NVIDIA PerfKit 6.5, to gain card-specific access to the real-time GPU performance counters.

In the next section, we explain our approach to physically validating the simulated sand. We present details on the testing methodology used and discuss quantitative findings. Simulation performance is addressed similarly in Section 7.3.

7.2 Granular physics validation

Confirming physically valid behaviour in the sand simulation framework requires that we first identify characteristic features of static and dynamic granular assemblies (sandpiles). In particular, we seek features that admit comparison to previous DEM simulations and empirical work on real-world sand. In the former case, we must distinguish between simulations that use *monodisperse* particles, where all particles have the same shape, size, and mass, as opposed to *polydisperse* particles, which vary in these properties. Furthermore, we are interested in the distinction between multiparticle and single particle models of sand.

As characteristic behaviours of a sandpile arise from mesoscopic (or microscopic) granular mechanics [JNB96], much previous work in granular science has focused on granule-granule contacts. Experiments with photoelastic disks and particles [MB05], under various packing conditions [VHC+99] and with local force perturbations [GHL+01], have suggested the presence of *heterogeneous force-bearing networks* (or chains) within dry granular material [OR97]. Understanding the behaviour of these force chains, as well as their spatial correlations, in the absence of a load or in the presence of applied forces, is a fundamental goal of granular mechanics [MB05]. In particular, the distribution of *normal contact forces* in these networks has been addressed extensively in the literature [RJM+96, MJN98, TVC98, LMF99, TW00, OLL+02, BEH+03, RBP+04, MB05, ZLW+06, vEEvH+07].

By convention, the distribution of normal contact forces, denoted $P(f)$, is calculated from *normalised* forces, so that $f \equiv f_n/\bar{f}_n$, where $\bar{f}_n$ is the average normal force. For notational

convenience, we shall assume that forces f_n and f_t refer to their normalised analogues, allowing us to distinguish between the normal contact force distribution $P(f_n)$ and the tangential contact force distribution $P(f_t)$.

Experimental measurements of $P(f_n)$ are typically conducted at particle-wall contacts because of the relative simplicity of boundary force measurements, as opposed to bulk force measurements. Physical experiments have used carbon paper at the base and the sides of a cylinder packed with glass beads to derive $P(f_n)$ [MJN98]. In contrast, DEM simulation readily permits measurement of forces, and many other properties, both *within* the bulk and at the *boundaries* of the simulated sand pile [SGL02]. In these measurements, independent of whether particles are monodisperse, amorphous, or vary in their packing order, two important features arise. First, the distribution of normal contact forces is indiscriminate with respect to particle friction [BMM+01] and packing structure. Second, the distribution exhibits an *exponential tail* at large f_n, by which we mean $f_n > 1$, and a plateau or peak near $f_n = 1$ [MJN98]. Indeed, an empirical functional form for the distribution has been suggested [MJN98], namely,

$$P(f_n) = a(1 - be^{-{f_n}^2})e^{-\beta f_n}, \tag{7.1}$$

where a, b, and β are fit parameters. Previous work has shown that Equation 7.1 provides a good fit for the entire range of forces recorded in both experimental work [MJN98] and DEM studies [SGL02]. In addition, previous work in DEM simulation, using the exponential fit, has suggested the tail of the tangential contact force distribution $P(f_t)$ *decreases slower* than the tail of $P(f_n)$ [SGL02].

More recent experiments [BEH+03, MB05, ZLW+06] and DEM simulations [RJM+96, TW00, OLL+02, vEEvH+07] have found that the tail of $P(f_n)$ *decreases faster* than predicted by an exponential fit. In particular, DEM simulation [vEEvH+07], as well as experimental techniques using photoelastic particles [MB05] and droplet emulsions [BEH+03, ZLW+06], has suggested a *Gaussian* tail. Thus, for large particle (or granule) counts, where sufficient statistical information can be accumulated, previous work [RJM+96, MJN98, vEEvH+07] gives

$$P(f_n) \sim exp(-c{f_n}^{\alpha}), \tag{7.2}$$

as a fit to the tail of the distribution ($f_n > 1$). Values for an exponent have varied between $\alpha = 1.5 \pm 0.4$ [RJM+96, MJN98] and $\alpha = 1.7 \pm 0.1$ [vEEvH+07]. The question of whether the tangential contact force distribution $P(f_t)$ has a similar tail is not answered in previous work.

For a sand pile subjected to *uniaxial compression*, previous experimental work, DEM simulation, and theoretical analysis have all found peak formation near $f_n = 1$. Radeke et al. [RBP+04] discuss the appropriateness of the *gamma distribution* as a fit to the distribution of force magnitudes for granular media in a *loaded dense* state. For comparison to Equations 7.1 and 7.2,

we restate the gamma probability distribution suitably as

$$P(f_n) = f_n^{\alpha-1} \frac{1}{\Gamma(\alpha)\, b^{\alpha}} e^{(-f_n/b)}, \tag{7.3}$$

where $a, b, f_n > 0$ and $\Gamma(\alpha) = (\alpha - 1)!$ for α a positive integer.

Figure 7.1 provides a comparison of the exponential, Gaussian, and gamma fit functions. For the case of uniaxial compression, the gamma fit evidences a peak near $f_n = 1$, while under standard conditions, we expect a flattening off in this region, as suggested by both the exponential and Gaussian fit functions. Moreover, the Gaussian is likely to produce a better fit to the tail of the distribution, where it decreases faster than the exponential fit.

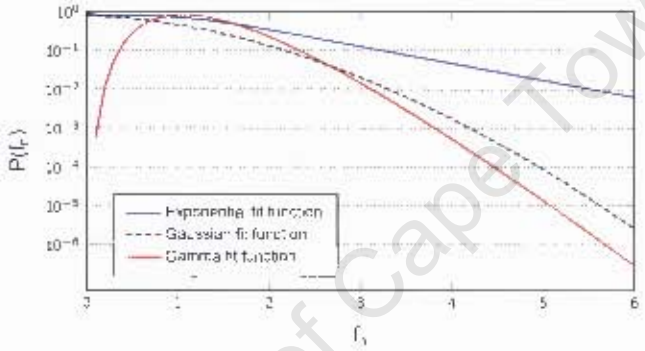


Figure 7.1: Fit functions for normal contact force distribution of a sandpile. The exponential fit has parameters $a = 2.55, b = 0.65, \beta = 1.35$; the Gaussian function uses $\alpha = 1.8$; and the Gamma function uses $a = 6, b = 0.21$.

Given the importance of the normal contact force distribution in proving physical validity, as well as the availability of good empirical functional fits, we view similar testing performed on our simulated sand as mandatory.

Apart from force distribution, previous work has examined the *spatial correlation* among forces [MJN98, SGL02]. Thus, based on previous work [MJN98, LMF99, SGL02], we define the spatial force-force correlation function $F(r)$ as

$$F(r) = \frac{\sum_i \sum_{j>i} \delta(r_{ij} - r) f_i f_j}{\sum_i \sum_{j>i} \delta(r_{ij} - r)}, \tag{7.4}$$

where r_{ij} is the fixed-distance between the centres of particles i and j, while f_i is the magnitude of the normalised *normal* contact force acting on particle i. Formally, the function δ is zero everywhere, excepting that $\delta(0) = 1$. However, we allow for a small tolerance ϵ, less than a particle's radius, to account for imprecision in particle positions. That is, $\delta(x) = 1$ for $x \in [-\epsilon, \epsilon]$. Importantly, since the calculation of force-force correlation is performed at the particle-level, we assume particles i and j are *not* part of the same grain.

Results have varied, with some previous experimental work seeing no spatial correlation among forces [MJN98], while others find only weak correlations at the base of the pile, extending

out approximately five particle diameters [LMF99]. Previous DEM simulation has found force correlations over shorter distances, within the *bulk* of the material, extending no more than three particle diameters [SGL02].

These findings provide an additional test case for physically validating the simulated sand. The most favourable outcome for such a test would be finding no spatial correlation among forces, in agreement with previous experimental work. If spatial correlation is found, we expect this to be highly localised, extending a few particle diameters, at most.

Another important property of static sandpiles is the *contact geometry* among its granular constituents. In particular, we refer to the geometric arrangement of grain-grain contacts within the sandpile. Granule contact geometry has been studied under various packing conditions [BMM+01, SGL02], with some work looking at the distribution of contact angles $P(\theta)$ among granules and the influence of friction on this distribution [SGL02]. In the latter study, contact angles between *particles* were measured relative to a local spherical system, as shown in Figure 7.2. Two key results from the study were that friction has only a weak influence on the distribution of contact angles. In addition, large forces are concentrated at smaller angles to the vertical, suggesting a *vertical* support structure.

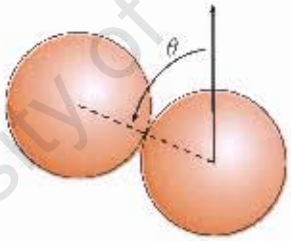


Figure 7.2: Measurement of contact angles using a local spherical coordinate system. The contact angle *theta* for a pair of bordering particles is measured relative to the vertical ($\theta = 0$). A horizontal contact produces a contact angle of $\theta = 90$ degrees.

The value of performing such a test is in the geometric information it extracts from the simulated sandpile. This information serves as a metric, allowing *structural* comparisons to other simulations, while accounting for differences in the underlying grain representation. The specific advantage is that we need only calculate the distribution of contact angles based on the *centroids* of each tetrahedral grain in the sand simulation framework. Such a comparison, based on particle positions alone, may be complicated by the rigid coupling between bordering particles that form part of the same tetrahedral grain.

Continuing with geometric contact among granules, measurement of forces in granular materials have found inhomogeneity in force magnitudes among granules in mutual contact [SGL02]. In particular, heterogeneous force *networks* have been elicited by experimental work with photoelastic particle packings [VHC+99], piles of photoelastic particles subject to local point-forces [GHL+01], and frictionless water droplets [ZLW+06]. Furthermore, computer simulation

has suggested the presence of mechanically complementary subnetworks, distinguishable by the magnitude of the forces they propagate through the granular material [RWJ+98]. A "strong" force network appears to be formed by particle-particle contacts propagating forces greater than the average normal contact force. Similarly, a "weak" force network appears to conduct smaller forces through the material.

A natural hypothesis is that the strong force network supports *all* the stress in the system, while the weak force network acts merely to support the latter framework [SGL02]. However, Silbert et al. [SGL02] found no clear evidence of a distinction between weak and strong force phases in the force-bearing structures. They found that half of the contacts in the strong force network contribute approximately 80% to the average contact force, which is not as large as expected [SGL02].

Performing a similar test case, to quantify the contribution of these supposed subnetworks to the average normal contact force in the bulk of the simulated sand, would allow comparison to previous work. In achieving similar results, we may provide further validation for the presence of expected granular behaviour in the simulation framework.

The contact structures and behaviours mentioned thus far also affect *global* features, such as a sandpile's response to stress. In an *oedometer* test, a stress is applied to a specimen along its vertical axis, while the corresponding strain in the horizontal direction is impeded [CE09]. In this instance, shear stresses and strains, as well as compressive stresses and volume changes might occur. However, the material is prevented from failing in shear, implying that compression is the dominant source of strain [CE09].

In such an experimental test, Coetzee et al. [CE09] compressed corn granules in a loaded cylinder, using a low compression rate to prevent dynamic effects. In addition, they replicated their experimental setup using numerical simulation of two-dimensional grains. Of comparative relevance to our simulation, they model each grain as a rigid arrangement of two partially interpenetrating spheres, thus emulating the two-dimensional silhouette of the experimental corn grains. While the simulated grains lack three-dimensional correspondence to the real-world material, comparison between their experiment and simulation is justified by the one dimensional action of the strain.

The experimental and DEM results indicate *elastic hysteresis* at the start of unloading, however, only the experimental results show *two-thirds* recovery of strain at the end of unloading phase. These behaviours may be explained by interlocking among granules preventing full elastic reversion to a previous state.

A similar test performed on our simulated sand would produce, at best, both elastic hysteresis and roughly two-thirds strain recovery in the unloading phase, thus reproducing a global feature of sand in the simulation framework.

Another, perhaps better known, global feature of sandpiles is the *angle of repose* (AOR): The *maximum angle* at which sand will remain stationary, once it has been dumped in a pile, as

shown in Figure 7.3. Measurement of this angle assumes the sandpile has reached an equilibrium configuration, involving the cessation of all grain movement, including sliding and rolling.

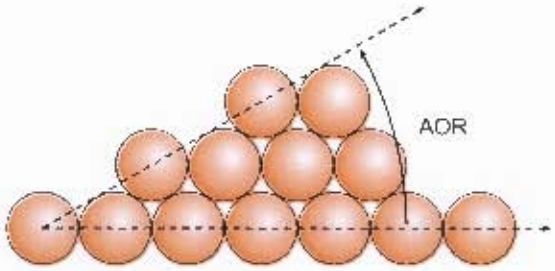


Figure 7.3: Angle of repose for a pile of spherical grains. An analogous measurement works for multiparticle grains, where elevation is calculated at the particle-level as well.

The AOR of a sandpile is known to vary according to global material properties, such as density; local properties, such as granule shape; and boundary conditions, which are present during and at the end of pile construction [ZXY+01]. For comparison to previous work, we limit our focus to the *coefficient of friction* (a local property) and its affect on the AOR. In particular, we look at *sliding friction* and ignore the affects of rolling friction and static friction.[1]

The coefficient of sliding friction controls the force impeding the translational motion among *particles* in contact [ZXY+01]. In particular, a sufficiently large coefficient, supports larger elastic deformation tangentially, at the contact point between particles. Thus, by increasing particle contact time and facilitating the dissipation of kinetic energy, sliding friction supports interlocking among granules. A reasonable prediction, therefore, is that larger coefficients should produce piles with larger angles of repose. Indeed, Zhou et al. [ZXY+02] find a power law relationship, that is, $a\mu^b + c$ where μ is the coefficient of sliding friction.

Thus, an additional test case for physical validation is to check whether piling simulated sand produces an AOR that is in a power law relationship to the coefficient of sliding friction.

A final important and *dynamic* feature of sand is the *weight-independent constant* mean flow rate of sand through an orifice. A typical hourglass (consisting of upper and lower "bulbs"), for example, relies on this interesting feature of granular material: Granules pass through the central opening between the bulbs at a constant mean flow rate that is *independent* of the amount of material remaining in the upper bulb. Thus, if an hourglass is turned over before the upper bulb is drained, the time taken for the material to return to the initial bulb is equal to the time elapsed so far. Other substances, including most fluids, do not demonstrate this behaviour. Indeed, many historical water clocks boast intricate designs that attempt to overcome a decreasing (that is, height-dependent) fluid flow rate [LMV92].

[1] The simulation framework does not account for *particle-level* rolling or static friction, since the *granule* itself ultimately produces the necessary rotational response or stationary behaviour. For rolling friction, viscous damping is applied to the torques acting on each granule. While for static friction, we assume that sliding friction provides enough "roughness" to particles so that stable interlocking among geometrically irregular granules can occur.

As a recognised property of (mesoscopic) granular material, constant mean flow rate is an important test case for physically validating the simulation. In particular, we expect to find flow decreasing linearly with time. Moreover, the diameter of the aperture has an equally important influence on flow behaviour. A widely accepted law proposed by Beverloo et al. [BLvdV61] predicts that grain flow rate through an orifice depends on the aperture diameter D_0 via

$$W = C\rho_g\sqrt{g}(D_0 - kd_g)^{\frac{5}{2}}, \tag{7.5}$$

where ρ_g and g are the apparent density and gravitational acceleration, respectively, while C and k are the empirical discharge and shape coefficients, respectively [MJA+07]. The dependence of C on the friction coefficient is implicitly assumed here. The validity of the law has been tested for monodisperse granules with diameter d_g larger than 0.5mm [MJA+07]. Comparison to this work would provide physical validation of the dynamic behaviour of the simulated sand.

In summary, we have looked briefly at previous research in granular material science that addresses important physical features of sand. The physical features requiring validation, as identified in this section, are summarised in Table 7.1. In all cases, the literature provides quantitative data for simulated or real-world sand. Confirming the physical correctness of the simulation framework, amounts to careful comparison of our results to this previous data. The testing methodology for extracting these results from the simulation framework is the topic of the next section.

Feature	Expected findings
Force distribution (standard)*	Exponential fit, as well as a Gaussian fit to the tail
Force distribution (compression)†	Gamma function fits the entire range
Force distribution (tangential)‡	Exponential fit; and possible Gaussian fit to the tail; with the tail decreasing slower than for normal contact forces
Force-force spatial correlation	Only a small local spatial correlation is seen, less than a few particle diameters
Contact angle distribution	Friction has only a weak influence on the distribution
Contact network	Weak distinction found between "weak" and "strong" force-bearing structures
Stress-strain behaviour	Elastic hysteresis and two-thirds strain recovery observed during unloading
AOR	Friction-dependent behaviour obeys a power law
Orifice flow (fixed D)§	Constant mean flow rate, with slower throughput for increasing interparticle friction
Orifice flow (variable D)¶	Relationship between flow rate and orifice diameter D obeys Beverloo's law

Table 7.1: Sand features to test for in the sand simulation and the expected result of each test.

* Normal contact force distribution for sandpile at rest in cylinder
† Normal contact force distribution for sandpile under uniaxial compression
‡ Tangential contact force distribution for sandpile at rest in cylinder
§ Orifice diameter is fixed across all simulations
¶ Orifice diameter is fixed for the duration of a simulation, but varied across simulations

7.2.1 Testing Methodology

Physically validating the sand simulation requires assaying five aspects of the sandpile: force distributions and force-force correlation; contact structures; stress-strain behaviour; AOR with respect to friction; and granule flow rate through an orifice. In this section, we construct test cases able to measure these features and thereby give an indication of whether the simulated sand meets the expectations summarised in Table 7.1.

Forces

To analyse the force characteristics of our sand simulation, we adopt the experimental setup of Silbert et al. [SGL02]. This choice is justified by our framework's *particle-level* similarity to their granular model, where they use uniformly-sized spherical particles. They differ in allocating each particle as a *separate* 3D grain. Nevertheless, their force model similarly assumes cohesionless, inelastic particles, which interact via either the Hookean (linear) spring or Hertzian contact laws, where our simulation employs the latter at the particle level.

The multiparticle framework presented in this thesis, however, stands in contrast to much of the previous work in DEM simulation, including Silbert et al., where single-particle granules are used. In particular, the latter simulations usually need explicit static-friction modelling, which may influence measured physical properties, such as the distribution of force. Therefore, we need to collect physical data on granules *and* their particle constituents, thus ensuring that any deviation from physically valid behaviour, at the grain- or particle-level, is detectable.

Similar to Silbert et al. [SGL02], we use a cylindrical container, with surfaces set to the same frictional and elastic properties as the grains themselves. As illustrated in Figure 7.4, the cylinder is constructed from a "rough" (particle-sampled) circular base, while a *smooth* implicit surface is used to model the cylindrical walls. While both rough and smooth boundaries are addressed in Silbert et al., the rough case requires depth-average normalisation of forces. The latter is needed to account for periodic packings due to "granular" (particle-sampled) sidewalls, in contrast to the smooth case. Nevertheless, they find no significant differences between the two approaches.

For simulation of a stationary sandpile, under previously stated conditions, we record both the normal and tangential contact force *magnitudes* experienced by 64K grains and 256K particles, during a single simulation time step. Since particle forces are stored in the force texture, we simply copy particle force data directly from GPU texture memory to CPU memory and save the data to disk. For grains, however, the total force is calculated and used only during grain dynamics, but not stored in a texture. Thus, we use an *ad hoc* texture image in the simulation and set this as an additional render target in the fragment shader handling grain property updates.

In each case, the probability distributions for the recorded forces are computed by a general procedure that we apply to any input data for which a probability distribution must be derived. We begin by calculating the Kaplan-Meier estimate [KM58] of the cumulative distribution

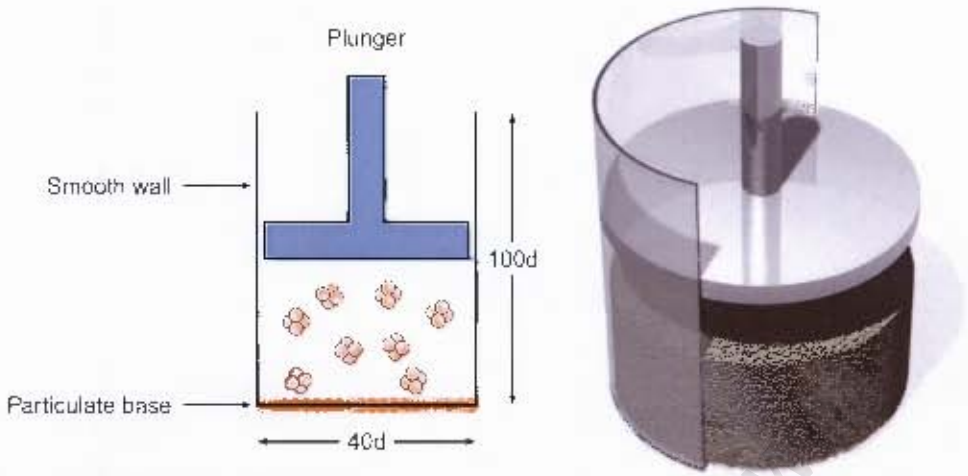


Figure 7.4: The cylinder setup used to hold the simulated sand during testing. Cylinder surfaces are represented *implicitly*, with a particle-sampled base adding "roughness" and smooth walls preventing periodic effects. The plunger is used in later tests to apply an axial stress to the material. The suffix 'd' indicates units of particle diameter. The image on the right shows the actual simulation arrangement, including 65,536 grains, with part of the cylinder removed for clarity.

function (cdf) from input values (either grain or particle forces, in the present case), to produce an *empirical cdf*. The resulting cdf values, calculated at discrete points, are binned according to the Freedman-Diaconis rule [FD81]. The result is "differentiated" by applying finite differences to the cdf, to produce values for the matching *probability density function*, that is, the probability distribution of the input data.

Measuring forces in a sandpile under uniaxial compression, requires using a plunger in the simulation (see Figure 7.4), to add a compressive stress at the top of pile. Instead of dynamic compression, however, we apply a vertical force of fixed magnitude and wait for the pile to stabilise. The latter means ensuring all grains have near zero linear and rotational velocity. When this condition is met, all force magnitudes are recorded in a single simulation time step. Producing the normal contact force distributions from this data follows the same procedure as described above for a sandpile under standard conditions.

For *force-force spatial correlations*, we need to know not only force magnitudes, but granule and particle positions as well (see Equation 7.4). The use of position data in the correlation function F, adds significant computational complexity to the calculation. In the case of particles, for example, computing a *single* value of F, say at r, requires that we locate for *each* particle, *all* other particles lying a distance r away. As seen in Figure 7.5, the maximum number of uniformly sized spheres able touch an equivalent sphere, without producing intersections, is 12 [Ans04]. Therefore, for distances of only one particle diameter, the calculation already entails significant computational cost. An analogous argument holds for granules as well.

The problem can be mitigated to a manageable extent by using a neighbourhood data structure, such as an octree. This allows searching around an arbitrary point in space, for example, a particle's centre, for those particles that lie within a specified distance of the centre point. The

Figure 7.5: One possible arrangement of spheres that achieves the maximum adjacency count of 12. The central sphere has been removed, making visible one sphere at the back. Another sphere is concealed, accepting a fractional part visible between the two rightmost spheres.

advantage is that the smaller particle neighbourhoods greatly reduces the number of candidate particles, thus increasing computational speed. However, many of these data structures return particles in a *cubical*, as opposed to *spherical* neighbourhood, centred around the query point. This requires further validation of the particle positions, to ensure they lie at the correct distance.

While the spatial correlation function requires finding particles lying at a distance r from another particle, we can rearrange the summations (see Equation 7.4) to make use of neighbourhood querying. For each particle i, we calculate distances r_{ij} and force products $f_i f_j$, relative to *all* particles j lying within the spherical neighbourhood of radius $r_{\max}$. The force products are accrued to matching F_r, initially set to zero, while matching a_r, initially zero, are incremented by one. When all particles have been processed in this way, we merely divide the accrued forces by their matching counts to find $F(r) = F_r/a_r$.

While the previous enhancements work for particles, grains present two important complications: Selecting an appropriate diameter for a tetrahedral grain is nontrivial, as is calculating the distance between two grains. The diameter is used to normalise the distances r, over which the correlation is computed, thereby allowing meaningful comparison to previous work. The distance between grains would need a heuristic, such as the smallest distance between their particle constituents. This requires ten comparisons between particles on different grains in the worst case. Along with neighbourhood searching, grain-level correlation is computationally expensive. Thus, we calculate only *particle-based* force-force spatial correlation.

Contact geometry

A similar methodology to spatial correlation is followed to produce the *distribution of contact angles*, which provides information on the contact geometry within the pile. Here, we are most interested in the distribution of *grain* contact angles, as grains are the smallest mobile rigid units comprising the sandpile. Nevertheless, for comparison, we compute the distribution of particle contact angles as well. The latter entails using neighbourhood queries to locate particle-level contacts among *distinct* grains, while ignoring same-grain particle contacts, as these do not inform about the interaction between moving components. Thus, in deriving both distributions, we need to resolve particle IDs into their parent granules' IDs. The required mapping between

these ID spaces is given in Section 4.3.1.

Global features of intergranule contact, namely, the *contact network* can be examined indirectly. For comparison to the result of Silbert et al. [SGL02], we quantify the contribution of the "weak" and "strong" subnetworks to the average normal contact force in the bulk of the material. In particular, using a variable *threshold force*, f_{cut}, we may calculate the *fraction of contacts* remaining in the force network whose contact force is greater than f_{cut} (for strong forces) or less than f_{cut} (for weak forces). Similarly, we are able to calculate the *percentage contribution* each network makes to the average force.

Stress-strain

The global response of a material to a large uniform stress is an important mechanical consideration, since this determines, in part, how the simulated material will interact with arbitrary objects in the environment. Using the same test setup as before (see Figure 7.4), including the plunger for compression, we simulate the stress-strain experiment of Coetzee et al. [CE09]. Sand is compressed initially by a small force to produce a flattened top, after which the plunger is eased upwards to the lowest height at which zero compression is recorded. The experiment then begins by recording the deformation of sand in response to increasing force applied downwards by the piston. The compression rate used is small to prevent dynamic effects and allow the sandpile adequate time to respond.

AOR

Another global feature of the sandpile is its AOR. Numerous methods exist to measure the AOR [Car70], at least two of which employ direct measurement of a stationary pile, formed by either grain injection [GH97] or grain discharge [ZXY+01]. Alternatively, indirect measurement is achieved by tilting a leveled heap, typically in a rotating drum, until avalanche occurs [PSO+06].

We use an injection scheme, as shown in Figure 7.6, modeled on the discharge scheme of Zhou et al. [ZXY+02], but differing in two crucial ways. Firstly, our test case discharges sand from a reservoir, through a single scupper, into the sump below. Since the AOR is measured for the pile in the sump, our approach is technically an injection scheme. Zhou et al., in contrast, measures the AOR of sand released into the *reservoir*. This measurement is made once excess runoff has discharged via two scuppers located on either side of the reservoir. We do not use this approach in our case, since friction and interlocking between non-spherical granules may hasten scupper discharge by "pulling" otherwise stationary granules into the outflow. The resulting AOR, therefore, fails to accurately model pile formation under conditions of more gradual runoff and, instead, estimates the resulting AOR after avalanching. Secondly, Zhou et al. use a container with an initial depth of four particle-diameters, which is varied across experiments. Instead, we choose to use a much larger container depth to reduce sidewall effects.

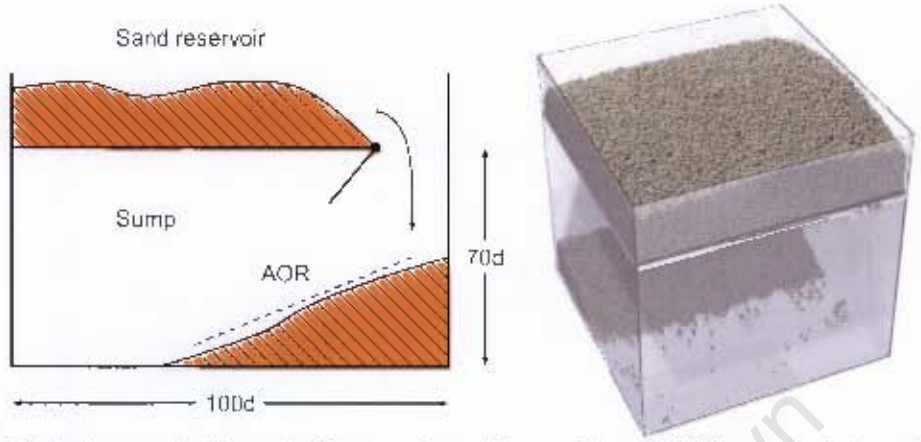


Figure 7.6: Test case setup for angle of repose. A sandpile consisting of 64000 granules is placed into the reservoir. The outlet or scupper, running the entire 100d depth of the container, is then opened, thus releasing granules to flow into the sump below. After approximately 24000 granules have passed into the sump, the outlet is closed and the AOR of the material in the sump is measured. The result of an actual simulation run is shown on the right.

Granular flow

The final simulation property to measure is the flow rate of sand through an orifice, such as present between the "bulbs" of an hourglass. While arbitrary hourglass designs are not guaranteed to produce constant mean flow rate, specific boundary conditions admitting this behaviour have been found empirically [NTS+82]. In particular, an inverted cone (hourglass) exhibits constant mean flow rate while the material it holds stays at a height of two-and-half times its width (W) or more. In addition, the width of the material needs to conform to the inequality $W > D - 30d$, where D is the diameter of the outlet [HR01]. Finally, the flow rate itself is relatively smooth provided $4d < D < 6d$, where values of D smaller than the lower bound often produce blockage [HR01].

Considering these conditions, we construct the test case as shown in Figure 7.7. In particular, we examine two determinants of flow rate, namely, *interparticle friction* and the *outlet diameter* D. In the latter case, the outlet diameter takes on values in the interval $[8d, 24d]$.

7.2.2 Results and analysis

In the previous section, testing procedures for five aspects of the sandpile were outlined. In this section, we give the results of those tests. For forces, we give the distributions for normal and tangential contact under standard conditions and under uniaxial compression, as well as the results for force-force spatial correlation. The contact geometry among granules is quantified indirectly, through contact angle distributions and changes in intergranule contact count over different force thresholds. We show the strain in a sandpile relative to an applied stress. Finally, we quantify the effect friction has on sandpile AOR and granule flow rate. In the latter case, we also observe flow rate across different outlet diameters.

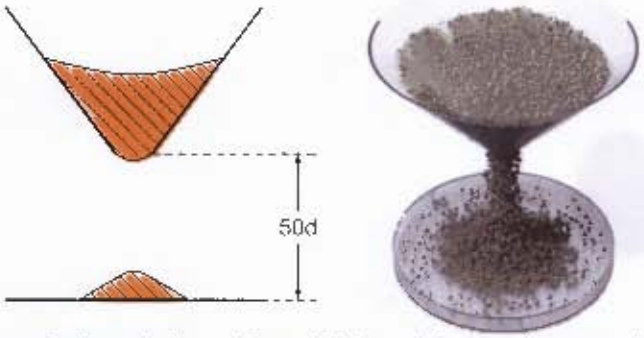


Figure 7.7: Test case for flow rate through funnel. An inverted cone with an aperture of 45 degrees is sliced orthogonally to its central axis, thereby removing its apex and producing a "funnel" with a hole of radius 8d. The funnel is made sufficiently large to comfortably contain 64K granules and is placed with its hole at a height of 50d above the floor. The funnel itself is modeled implicitly, as a smooth surface.

Forces

Figure 7.8(a) shows the distributions of normal contact force magnitudes for particles and granules, in an unloaded sandpile, using a coefficient of friction $\mu = 0.5$. For both particles and granules, the matching exponential fits (described by Equation 7.1) show good agreement with the exponential fit of Silbert et al. [SGL02]. Interestingly, while the fit parameters for particles and granules are numerically close to one another (see Table 7.2), their graphs show a marked deviation from one another, beginning with forces two to three times the mean force, that is, $f_n > 2$. In addition, while the exponential fit appears to give a closer approximation to the particle distribution, the latter distribution also starts deviating from the fit, near $f_n = 4$.

	Test case	Exponential			Gaussian	Gamma	
		a	b	β	α	a	b
Particles	Friction	2.10	0.61	1.29	1.51		
	No friction	2.78	0.72	1.46			
	Compression					3.36	0.30
Granules	Friction	2.14	0.66	1.24	2.45		
	No friction	10.05	1.01	2.13			
	Compression					6.14	0.16
Silbert et al.	Friction	2.55	0.65	1.35			

Table 7.2: Comparison of the parameters for fitting functions across test cases. The values for particles and granules are from the simulated material. Test cases involving friction use $\mu = 0.5$.

The deviation between particle and granule distribution is not unexpected, as some of the large forces that act on particles, act tangentially to the parent grain's centre. In this way, they may neutralise one another through conversion into opposing angular accelerations. Of greater interest is the finding of a disagreement between the exponential fit and the distributions around $f_n = 3$. For comparison, we have included in Figure 7.8(a) the Gaussian approximations to the *tails* of

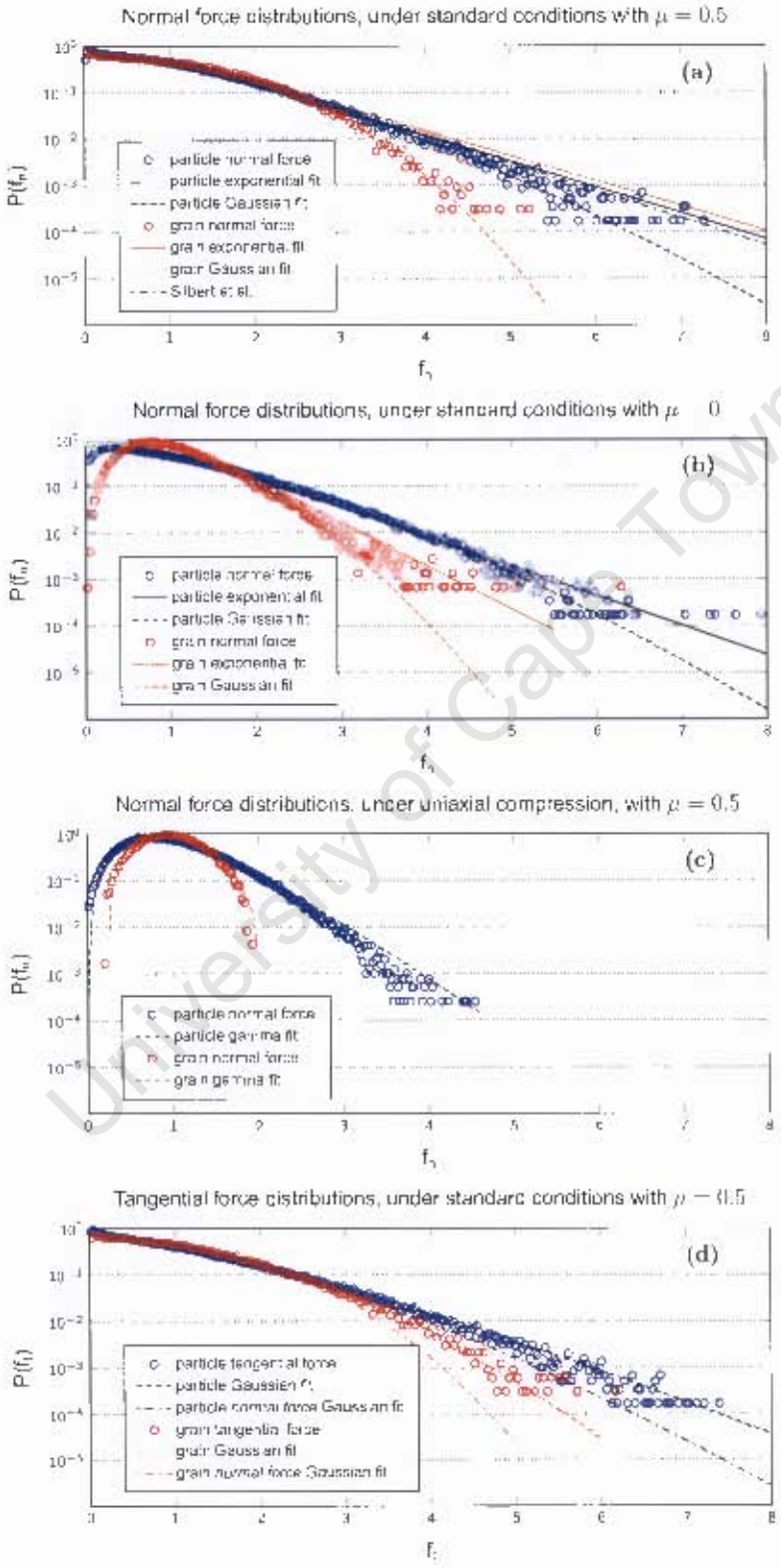


Figure 7.8: Results for normal and tangential force distributions, under standard conditions, with normal forces recorded under uniaxial compression as well.

the distributions. Surprisingly, in both cases, the Gaussian fit, with exponents in the expected ranges (see Table 7.2), provides a visibly better approximation over the *entire* distribution, not just the tails. This result suggests that the distribution of small forces ($f_n < 1$), which are *not* included in the Gaussian fit calculation, are predicted by the distribution of larger forces.

While agreeing strongly in the tails of the $P(f_n)$, the smaller force regions appear to lack an obvious peak or plateau near $f_n = 1$. However, we found disagreement among previous experimental work and DEM simulation concerning the behaviour of $P(f_n)$ as f_n approaches 0 [SGL02]. In addition, we face the burden of a single-precision floating point arithmetic on the GPU, where rounding or numerical imprecision may invalidate comparisons based on *small* force magnitudes.

Nevertheless, previous work in DEM simulation finds, for forces less than the mean, the distribution bends *downwards* when $\mu = 0$ and *upwards* when friction is present in the simulation [SGL02]. We find the same result, as seen in Figure 7.8(b), where both distributions evidence a peak near $f_n = 1$. In addition, the Gaussian fits to the tail data provide better approximations to these distributions, as before. This result again shows that smaller forces can be estimated from tail data alone. It also suggests the Gaussian fit is useful across different coefficients of friction.

Our results for uniaxial compression are shown in Figure 7.8(c), along with a gamma distribution fit in each case. The gamma fit for the particle and grain force distributions appear to closely match the data (see also Table 7.2). We also observe clear peak formation near $f_n = 1$ and a much smaller force range than seen in previous distributions. The latter two observations may be explained by the presence of force bearing structures in the sandpile, each conducting forces of different magnitudes. When these structures are overwhelmed by a much larger applied force, a more uniform force structure is produced in the pile. Thus, a *larger* mean force *dominates* among the recorded forces, producing peak behaviour at the mean and a smaller normalised force range.

For completeness, we show the distribution of *tangential forces* $P(f_t)$ in Figure 7.8(d), which includes the Gaussian fits to the *normal* contact force distributions for comparison. As before, normalised forces have been used. The results show that $P(f_t)$ decays more slowly than $P(f_n)$, which is in agreement with previous DEM simulation [SGL02].

Finally, all force distributions in Figure 7.8 appear to flatten out at the end of their tails. This is partly a result of the velocity and penetration constraints employed by the simulation, which prevents particles and grains from disobeying the CFL condition [CFL67], thus guaranteeing stable simulation behaviour. This amounts to preventing excessive penetration during a simulation time step, which would otherwise produce larger forces than expected and numerical instability. Thus, large forces arise only sporadically in the simulation and correspond to unit-sized bins at the far end of the force histogram. Conversion of the latter into a distribution of unit area results in the apparent flatness at the lower end of the logarithmic scale.

In Figure 7.9, we give the spatial force-force correlation derived from the bulk of the simulated

material. Distances in the diagram are normalised by particle diameter and are found to be in the range $[(1 - \frac{1}{10})d, 4d]$. The upper bound was chosen beforehand for comparison with the range used in previous work. However, the lower bound results from a physical constraint set in the simulation that limits particle penetration depth to one tenth of a particle diameter. This prevents the production of very large forces and consequent numerical instability.

The latter penetration constraint is reflected as a large increase in F for r near $(1 - \frac{1}{10})d$, where forces of similar magnitude direct particles away from one another. In addition, we see a negligible effect of friction on the correlation function, where the presence of friction resulted in only a very slight increase in *local* correlation in previous DEM simulation [SGL02]. For all three cases in the figure, localised correlation is observed extending roughly two particle diameters into the bulk, suggesting an unlocalised structure to force transmission network.

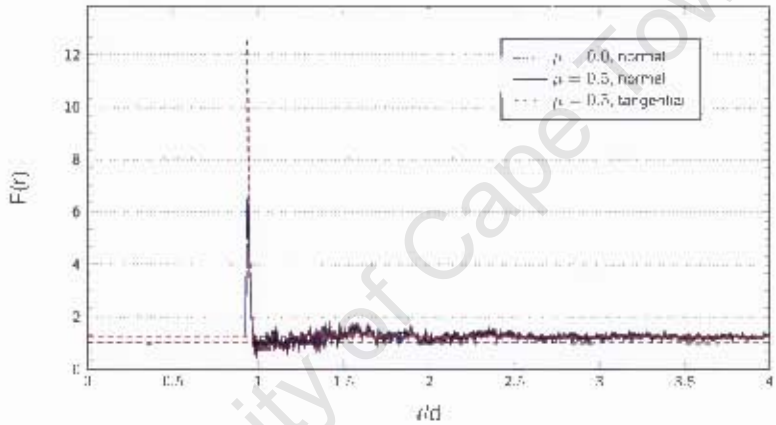


Figure 7.9: Spatial force-pair correlation function for normal and tangential contact forces. The results for $\mu = 0$ and $\mu = 0.5$ are plotted against a distance r normalised by the fixed particle diameter d. The dashed black lines indicates $F = 1$, while the red dashed line is at $F = 1.269$.

We derive the previous observations based on the reference line $F = 1.269$, which we found through a linear fit, constrained by a zero gradient. This produces the expected tapering off of F beyond $r = 2.5d$. Previous works finds the same behaviour, but relative to the line $F = 1$ [SGL02, MJN98]. One possible cause for this anomaly, is that we base our calculations on particles. The interlocking between granules might produce long range networks of contact, leading to a slight uniform increase in F.

Contact geometry

The contact angle distributions are given in Figure 7.10, where we give results for the presence and absence of friction in the simulation. For the frictional case, we include a second distribution of contact angles, but only among grains experiencing force magnitudes larger than twice the mean force. Importantly, these distributions are derived from *intergranular* contacts, thereby helping to discern geometric structure between mobile elements in the sandpile. The results show that friction does not play a significant role in the relative orientations of grains. In the presence of

friction, previous work has found large forces concentrated at lower angles, where we observe a small increase in concentration of large forces at large angles, that is, in the horizontal plane. This suggests interlocking among irregular granules, which results in contact "networks" that dissipate forces to the sides of the container.

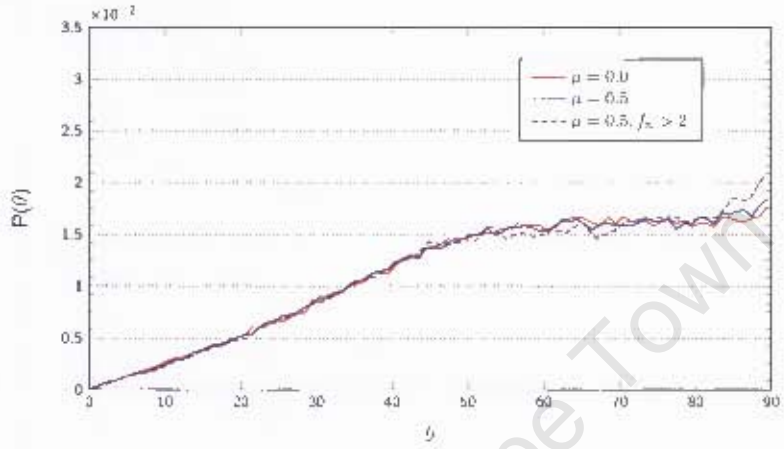


Figure 7.10: Probability distribution $P(\theta)$ for grain-grain contact angles. Here, θ is given relative to the local spherical system and defined as the angle the contact pair makes with the vertical, $\theta = 0$. Thus, horizontal contacts are given by $\theta = 90$. The dotted line indicates the distribution of forces twice the mean force.

In Figure 7.11, we show the fraction of contacts remaining in the force network, for contact forces greater than or less than the cutoff force f_{cut}. This indicates the size of the "strong" and "weak" force bearing parts of the sandpile, respectively, supporting the cutoff force. Also shown is the percentage contribution these structures make to average force, given a specific cutoff force.

The main result, indicated by the arrow in the figure, is that half of the contacts in the "strong" force network (where $f_n > f_{cut}$) contribute approximately 81% to the average contact force. The latter distinction is not as large as one might expect. However, this result is in agreement with Silbert et al. [SGL02], who found no clear evidence of a distinction between "weak" and "strong" force phases in the force-bearing structures.

Stress-strain

In Figure 7.12, we show the change in stress and strain in a confined sandpile produced by serial loading and unloading. The results show *elastic hysteresis* at the start of the unloading phase, as expected by elasticity theory and corroborated by experiment [CE09]. We also see roughly *two-thirds* of the strain recovered at the end of the unloading phase, which agrees with the *experimental* findings of Coetzee et al. [CE09]. Surprisingly, their *numerical simulation*, which used dual-particle granules, failed to show this behaviour. This suggests that the advantage of our model arises from interlocking among complex granules, possessing pronounced convex *and* concave surface features. Such geometrical "roughness" may facilitate irreversible sliding, when

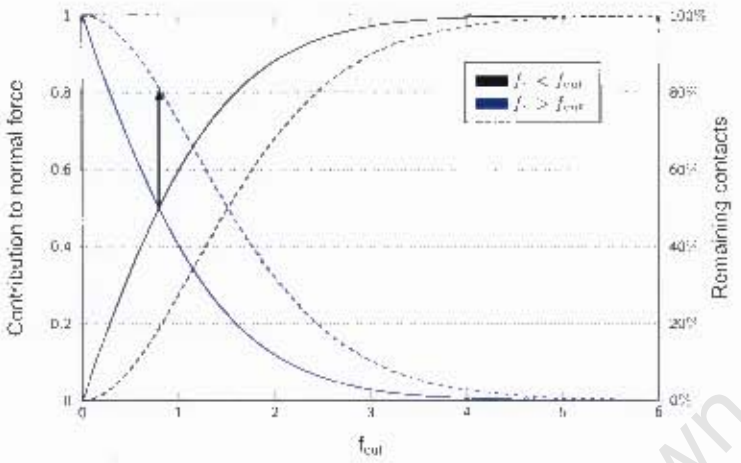


Figure 7.11: Fractional contribution to the bulk normal contact force (solid lines) and the percentage of particle contacts that make up that contribution (dashed lines), as a function of imposed contact force threshold f_{cut} for a sand pile with $\mu = 0.5$. The blue lines (dotted and solid) indicate contributions from normal forces larger than the cutoff force, while black lines are for forces smaller than the cutoff. The arrow indicates that 50% of particle contacts contribute to 81% of the bulk average contact force.

grains experience enough compression to rearrange into a pattern that prevents particle-concavity disengagement.

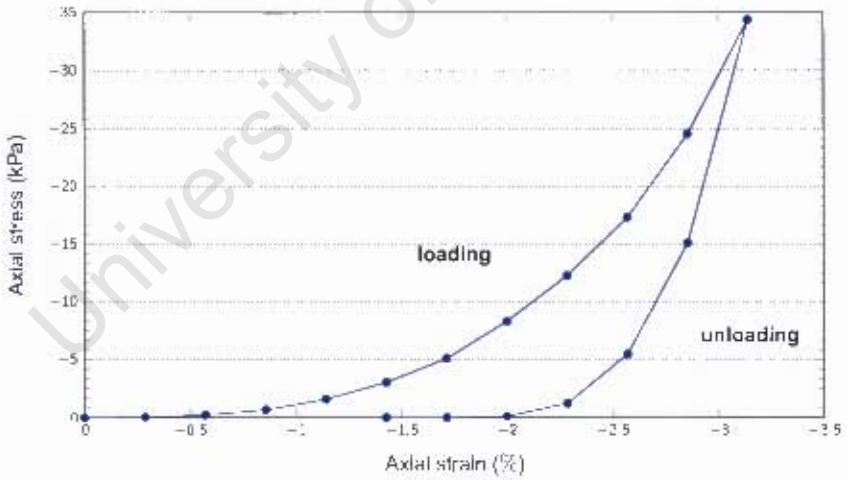


Figure 7.12: Uniaxial stress-strain behaviour for a sandpile in serial loading and unloading phases. Hysteresis is seen as the unloading phase returning along a different path. In addition, unloading does not return to the origin, showing an incomplete recovery of the initial strain.

AOR

In Figure 7.13, the results from the AOR measurements across different coefficient of sliding friction are shown. Importantly, the AOR measurement involves a large degree of uncertainty (with 95% confidence interval of ten degrees), caused by nonlinear slope features. We attempt

to mitigate some of the difficulty in computing the slope by using a robust linear least-squares regression with bisquare weighting on surface granules. This minimises the effect of outliers, while providing a good fit the entire range of surface granules. As evidenced in Figure 7.13, the resulting AORs appear to follow a power law, which is in agreement with findings of Zhou et al. [ZXY+02].

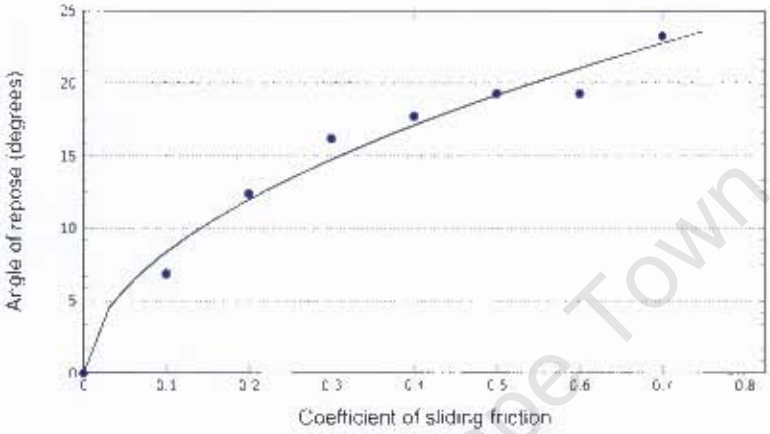


Figure 7.13: Angle of repose versus the coefficient of sliding friction. Particle, wall and floor restitution is set to 0.5. Friction at the walls and floor is fixed at 0.5. The line represents a power function fit to the data with an exponent of 0.51.

Granular flow

In Figure 7.14, the number of granules remaining in the cone (upper bulb) is given as a function of time, for different values of the particle-particle friction coefficient. We observe a *constant* mean flow rate in the outflow from the inverted funnel, visualised as a strong *linear* relationship in grain quantity and simulation time. The latter finding and the observation that increased interparticle friction leads to longer drainage time, are in accordance with experimental results [MJA+07].

In Figure 7.15, we provide simulation results for mass flow rate in relation to aperture diameter. We see good agreement with Beverloo et al. [BLvdV61] and previous numerical simulation [MJA+07]. Importantly, this test was performed with 64K particles, which conforms to the height requirement for material above the orifice, as discussed in Section 7.2.1.

7.2.3 Discussion

The main aim of testing was to corroborate valid granular behaviour in the simulation. In line with this goal, we quantified key physical properties of the simulated sand and compared our results to previous work.

Testing began with measurement of force distributions in the sandpile. Comparison of the parameter values of the Gaussian and exponential fits, as well as our subjective interpretation of

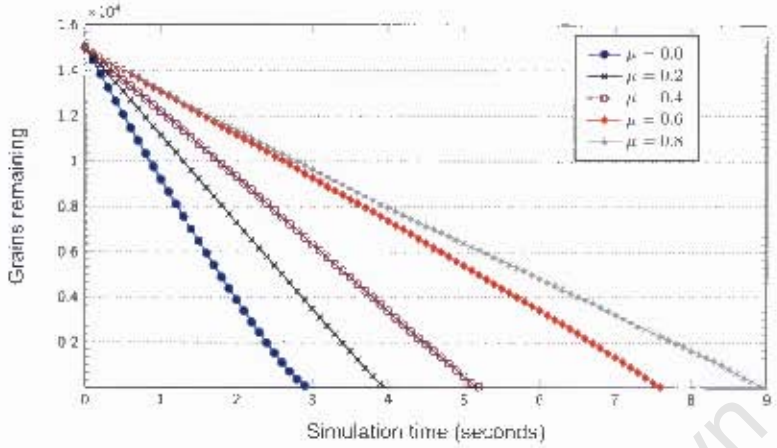


Figure 7.14: Flow rate through an "hour glass" (inverted cone). 16K grains, each weighing 0.0143kg, pass through a hole of diameter 8d in the bottom of an inverted cone having an aperture of 45 degrees and a particle-wall friction coefficient $\mu = 0.2$. Since grains are dropped into the cone at the start of simulation, we discount initial dynamic effects by collected data once 1K grains have passed through the hole. The resolution of the recorded data is five times denser than indicated by the plot marks.

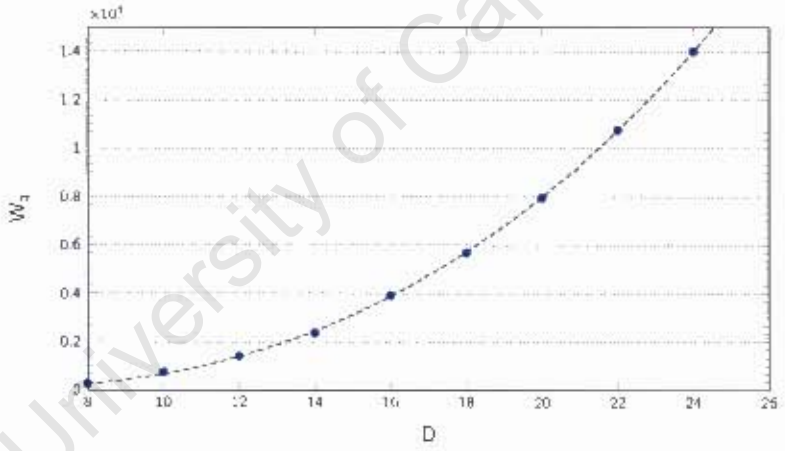


Figure 7.15: Flow rate W_g of grains versus the aperture diameter D of the inverted cone. W_g is the number of grains fallen per unit time. The dashed line indicates the best best fit using Equation 7.5, which is given by $W_g = 7.94(D - 4.107)^{\frac{5}{2}}$, where the first constant incorporates the density and gravitational coefficients

the distribution shapes, indicated a close match to previous work [SGL02]. Moreover, our results corroborated recent granular science research that advocates the superiority of the Gaussian fit, relative to the exponential fit, for a sandpile under standard conditions [OLL+02, BEH+03, MB05, ZLW+06, vEEvH+07]. We saw the Gaussian matching closely to and *interpolating* the empirical distributions over most of their force ranges, where the exponential fits failed beyond forces twice to three times the mean. For a sandpile under uniaxial compression, we found the Gamma fit interpolated most of the data, supporting its use in modelling compressed granular heaps, as suggested by previous work [RBP+04].

Digging deeper into sandpile morphology, we tested force-force spatial correlations. The results revealed localised correlation, which was not unexpected [SGL02]. The unexpected, however, did occur in the distribution of intergranule contact angles. This useful indirect measure of contact geometry showed that our granules appeared to distribute forces horizontally, to the sides of the container, rather than vertically, as with spherical granules [SGL02]. However, both localisation and supporting sidewalls are supported by the idea of localised stress tracts in sandpiles, which are implicated in the arching behaviour of heaped granules [JNB96]. The same behaviour was mentioned in the introduction to this thesis, where we referred to the depth-independent pressure found in full grain silos. This is thought to arise when the arching behaviour and the friction along the walls of a silo are sufficient to withstand the extra weight lying above [JN92].

Frictional behaviour *among* granule contacts also influences the angle of repose (AOR) of a sandpile, as our testing has shown. This relationship is expected, since sliding friction among particles and between particles and container walls plays a key role in controlling the linear and rotational motions of heaped particles [ZXY+02]. In the case of multiparticle granules, our results indicate a possible power law between sliding friction and AOR, agreeing with uniparticle granule simulation of Zhou et al. [ZXY+02]. However, we stress that while these conditions are necessary, they are not sufficient to recognise a *power law relationship*. Full corroboration would require extensive testing, using varied grain sizes and other orthogonal features. Nevertheless, without further testing, the simulation evidences compelling AOR-dependent behaviour, such as dune formation, as seen in Figure 7.16. Here, sand is dropped in two columns into a container, resulting in a typical dune-like surface.

In another test, pushing down on the sandpile and then removing this compression, produced an initial elastic hysteresis, followed by two-thirds recovery of the strain by the end of unloading. This simulation result reproduces the *physical experiment* of Coetzee et al. [CE09], where their numerical simulation did not. The advantage of our sand simulation framework is likely due to the irregular grain geometry, which may model interlocking among granules, and thus better inhibit full recovery of the initial strain.

Finally, we examined another well-known feature of sand and similar granular materials: its constant mean flow rate through an outlet, which makes simple hourglasses possible. Our experiments reproduced constant mean flow rate, across different sliding friction coefficients, as well as in accordance with Beverloo's law, across different outlet diameters. An example of the hourglass simulation is given in Figure 7.17, where the sand in the funnel evidences a *meniscus* (central depression) due to faster outflow at the centre.

Thus, for modelling the physically valid behaviour of real sand, lying in a container or interacting with its environment, including being compressed from above or flowing out from below, the simulated sand has demonstrated its applicability in all cases. Moreover, our findings show that a *multiparticle model* of irregular grain geometry is able to mimic many interesting and characteristic behaviours of sand. The only question left to answer is whether the computational cost involved in producing physically compelling behaviour admits real-time performance.

Figure 7.16: Stable dune formation with an uneven surface.

Figure 7.17: Sand flowing out of an inverted cone. Sand is initially dropped into the top of the inverted funnel, which mimics an hourglass in supporting constant mean flow rate through the bottom opening. In addition, pile formation is demonstrated underneath the opening.

In these examples, we show rigid bodies interacting with sand. In Figure 7.18, an avalanche causes a small structure to topple. The two-block structure experiences a force of its bottom, causing it to fall slightly towards the oncoming deluge. When the sand settles, the blocks sway backwards, with momentum causing the structure to topple. In Figure 7.25, we give an example of rigid body interaction without sand.

Figure 7.18: An avalanche toppling a structure.

7.3 Granular simulation performance

In this section, we answer two key questions about simulation performance:

1. How much sand can the framework simulate on a single GPU?
2. How quickly does this sand evolve?

We begin answering these questions by analysing how changes in sand volume affect simulation speed. In addition, we address various design limits that define how well the framework will scale with future developments in hardware.

7.3.1 Testing methodology

Similar to the tests performed for physical validation, we use a cylinder of radius 50d and height 100d. The walls of the cylinder are smooth, while the floor is "rough" (particle-sampled). Here sand is *dropped* into the cylinder in batches of 16,384 grains, where each batch consists

of 32x32x16 grains (that is, 16 grains in height). Grains in each batch are spaced with their centres approximately two particle diameters apart, allowing for a small random perturbation in position, as well as randomised orientations. Each batch is centred above the cylinder's floor and dropped from roughly 15d above the pile's top, excepting the first batch, which starts 50d above the floor. For each batch dropped, we wait for both the *mean collision count* and the *mean frame time* to stabilise for 30 seconds wall-clock time, after which the mean values are recorded and the next batch is dropped.

Importantly, since grains are added progressively to the simulation, reallocating storage and copying data into new textures would needlessly waste GPU compute time. Instead, we preallocate sufficient space to contain all granules, up to and including those in the last batch. Furthermore, since the *implied* particle and granule ids for empty texture space are numerically zero, they are ignored in physically-based calculations and in particle and granule updates.[2]

Finally, most of our tests quantify independent variable performance relative to either *elapsed frame time* or the variable's *percentage contribution* to total frame time. The former measure aids comparison across different trials, while the latter measure allows in-trial comparisons. Furthermore, isolating and quantifying computational performance requires that we turn off particle rendering in Section 7.3.2. Later, in Section 7.3.3, which addresses visualisation and lighting performance, we turn rendering on again.

7.3.2 Speed and volume

The effect of sand volume on simulation time is a key determinant in framework performance. In Figure 7.19, we quantify this volume-time relationship and observe that increases in total simulation time (as measured by frame time) are in direct proportion to increases in grain quantity. That is, adding sand grains into the simulation produces a predictable *linear* increase in the total time spent simulating the entire sand volume.

The question remains as to what causes the increases in total simulation time seen with larger grain counts. Given that grain-based processing alone determines this relationship, we argue *a priori* for four dominant causes, viz. greater cost maintaining particle and grain properties; longer grid construction time; more grid queries; and more physically-based calculation.

Firstly, for particles and grains, we have already eliminated allocation costs by preallocating the total texture space (as discussed previously in Section 7.3.1). This leaves the cost of updating the property textures. We measure this by disabling force calculation and grid operations. The result, as revealed in Figure 7.19, shows that property updates contribute as little as 4 milliseconds (2.6%) to the total cost of simulating 256K grains. Moreover, this contribution is manifestly less than other variables in the figure.

[2]A fragment shader is spawned for *all* texel positions, empty or otherwise. However, each shader first checks to see if it is associated with an empty id. If so, it discards its output immediately, thus ceasing activity before any physics calculations run.

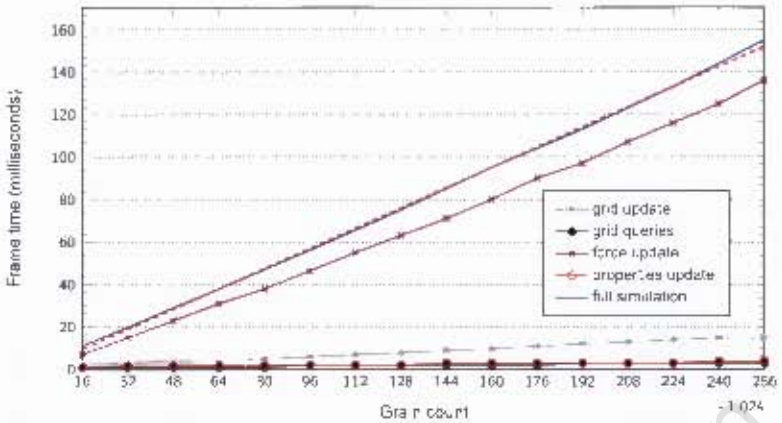


Figure 7.19: Simulation performance measured by comparing computational time versus sand volume. We quantify variables by the frame time they consume. Individually measured variables include the time spent updating the grid, performing grid queries, calculating particle forces, and updating particle and granule properties. The total simulation time (blue line) is shown along with its linear fit (red dashed line).

Next, the computational cost of grid construction is found by disabling all physically-based calculation and skipping particle and granule updates. We ignore the once-off cost of grid allocation and instead measure only grid update performance. The result, given in Figure 7.19, shows grid update time increases linearly with increasing sand volume. However, the contribution it makes to total simulation time remains below 20 milliseconds, for up to 256K grains (one million particles).

A related influence on simulation time is the cost of querying a fully-constructed and updated grid. Computing this cost during a simulation is problematic, as it calls for eliminating collision calculations that otherwise prevent particle interpenetration. Such unrestrained interpenetration soon results in an invalid grid, which affords little meaningful *dynamic* information on runtime performance. Instead, we settle for recording *static* performance by disabling not only force calculations, but particle and granule updates as well. This freezes particle and granule positions and results in a typical (pile-like) particle-id distribution in the grid. Constructing the grid once and reusing it for queries produces the result shown in Figure 7.19. For 256K grains, we find a 3 millisecond query cost, representing a 1.9% contribution to total simulation time.[3] This reinforces the negligible contribution of grid operations, including both queries and updates, to the total simulation cost, especially as sand volume increases.

Finally, since sand (partially) fills the cylinder, additional grains are expected to produce more collision events and more physically-based force calculations. The relationship between grain count and collision count is visualised in Figure 7.20, where we observe *linear* scaling. This result may be explained, in part, by a geometric limit in the maximum number of spheres that can be in contact with one another at any given time, since most collisions occur in the bulk of the

[3] This result is surprising, as every shader instance, for each of the one million particles, queries its own particle's voxel and twenty-six surrounding voxels. The low cost here may follow from our query algorithm, which reads the grid in layers. Specifically, nine voxels are read from a single layer of the grid, which match to nine bordering texels on the grid texture. This may facilitate prefetching from the texture units, making local queries fast.

material, where this limiting behaviour plays a role. In turn, this suggests a possible causal role of increasing collision count on increasing simulation time. This is supported in Figure 7.19, where we find that force calculation is the *dominant* contributor to total simulation cost.

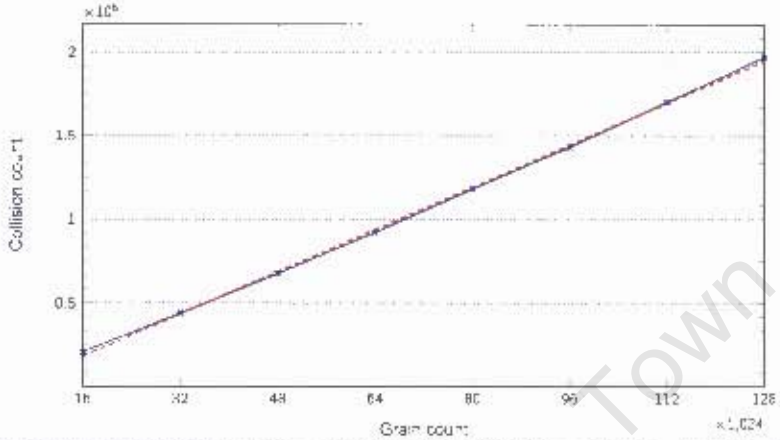


Figure 7.20: Collision count versus grain count for grains added to a cylinder. The red dashed line is a linear fit to the data.

We find further support for the importance of force calculation by directly analysing hardware performance, viz. fragment shader, vertex shader, geometry shader, and texture unit use, for increasing collision counts. The results of this analysis carried out for stationary sand piles of varying size, is given in Table 7.3.

	Fragment shader	Texture busy	Texture waits	Frame time
Collisions	0.8562 (0.05)	0.8118 (0.05)		0.9999 (0.0001)
Fragment shader	n/a	0.8883 (0.005)		0.8614 (0.01)
Texture busy		n/a	0.7766 (0.05)	0.8143 (0.05)

Table 7.3: The significant correlations found between GPU hardware counters. Correlations are shown with their matching p-values in brackets. For example, collisions and fragment shader activity is read as $0.8562\ (p < 0.05)$. We measured GPU performance across sixteen *stationary* piles with grain quantities from 16K to 128K inclusive, at 16K increments. For each pile, we recorded GPU data for 20 seconds wall-clock time and averaged the results to obtain the final readings. The recorded data included frame time; fragment, vertex, and fragment shader activity; and the percentage of GPU compute time shaders spent waiting for texture units. We also included collision counts as measured previously. The table shows only the significant correlations we found.

We note the strong correlations in collision count versus fragment shader activity and collision count versus texture unit use. These are explained by fragment shaders containing all the physically-based calculations and the majority of texture read instructions, which together result in lengthy compute times and resource use. Indeed, we find pixel shader activity strongly correlates with texture unit activity.

In comparison, most vertex shaders perform simple coordinate transformations. However, the most important of these is selecting the positions to write particles ids to in the grid texture. Yet, our results show no significant correlations relating to vertex shaders, which again confirms

the negligible contribution grid operations make to simulation time.

The correlations found across different collision counts, support the argument that increasing sandpile volume (in the cylinder), lengthens frame time by increasing the amount of physically-based collision calculation performed. We further validate this assertion by recording the activity of an *ad hoc* geometry shader added to the grid construction program.[4] The activity of the geometry shader in Figure 7.21 corroborates the lack of influence of grid construction, though it does not discount the effect of grid queries. However, we observe pixel shader activity and frame time both increasing and leveling off at similar times during the test, which again supports the relationship between simulation time and physically-based calculation.

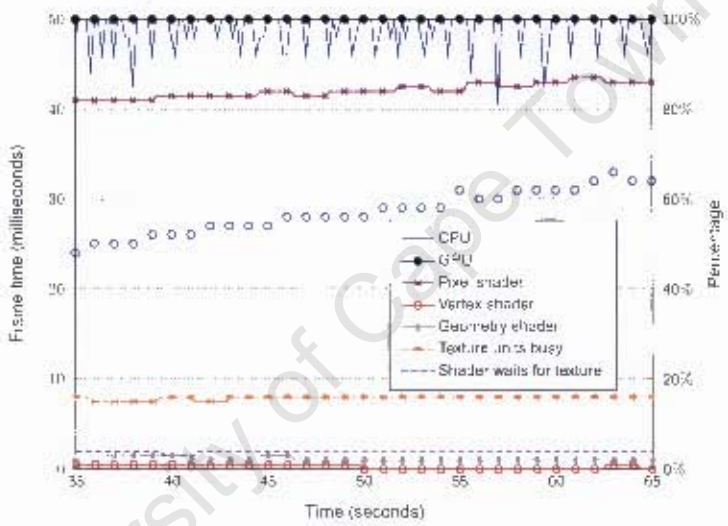


Figure 7.21: Hardware performance measured by frame time and resource use versus wall-clock time. Frame time, indicated by o, is plotted with respect to the left vertical axis. All other variables (indicated in the legend) are measured against the right vertical axis, as percentage use of their respective resources.

In addition, Figure 7.21 also maximal use of the CPU, with erratic behaviour due to both averaging of dual-CPU activity and CPU-GPU communication time. However, the GPU is used maximally as well, demonstrating efficient resource use and CPU-GPU transfer. In the case of inefficient transfer, we would see less GPU activity as resources are stalled by data transfer.

The results of this section together suggest specific hardware limits affecting framework performance. We speculate, if future hardware trends continue to include more shader cores, more sand could be simulated, at similar frame times. However, simulating the same volume of sand at a higher rate, would need larger texture memory bandwidth; faster texture access; improvement to the underlying simulation algorithm; or some combination of these.

[4] We replaced the vertex shader, used in grid construction, with a trivial (pass-through) shader and employed a geometry shader to handle the coordinate transform work. Importantly, the geometry shader replicates all the functionality of the vertex shader, with little modification to the underlying code. This allows us to distinguish grid construction performance from other influences by observing geometry shader activity relative to vertex and pixel shader activity.

Comparison to previous work

Providing meaningful comparison to previous work is not straightforward. In particular, different simulations may use different time step lengths or lack results for specific grain counts, complicating frame time and frame rate comparisons.

We resolve these difficulties by basing performance comparisons on the *Cundall number*, which normalises differences in computation time, grain quantity, and iteration count.[5] Specifically, we compute $C = N_t N/T$, where N_t is the number of simulation time steps, N is the number of rigid bodies, and T is the computational time taken by the CPU or GPU [Cle09]. Therefore, the Cundall number C can be thought of as the number of *granule time steps* per computational second. In Table 7.4, Cundall numbers are used to compare performance results with previous work. Note that we have used our worst Cundall number over the range of 16K to 256K grains.

	Framework	**Harada et al.***	**Venetilo et al.†**	**Ferrez‡**
Parallelism§	GPU	GPU	GPU	SMP
Particle sizes¶	Uniform	Uniform	Uniform	Multiple
Cundall number	1.490×10^6	6.652×10^5 ‖	4.736×10^6 **	2.0234×10^4 ††

Table 7.4: Performance and feature comparison of the sand framework to other DEM simulations.
* [HTK+07] † [VC07] ‡ [Fer01]
§ SMP PC with four Pentium III Xeon processors or an NVIDIA 8800 GTX (GPU) in *all* other cases.
¶ Refers to support for monodisperse (uniform) or polydisperse (multiple) particles.
‖ Based on 16K *four-particle* chess pieces simulated at 40.6 fps
** Based on 128K *particles* simulated at 37 fps
†† Based on 12,000 *particles* simulated for 1000 iterations in 593 seconds.

The Cundall numbers demonstrate the sand simulation performs two orders of magnitude more granule time steps compared to a CPU-based DEM simulation, while matching the performance of previous GPU-based simulations. The discrepancy in Cundall numbers between our GPU-based multiparticle rigid-body simulation, and the monodisperse single-particle simulation described by Venetilo et al. [VC07], indicates monodisperse simulation performs better. The latter's advantage arises from ignoring three degrees of rotational freedom, which means less calculation, but at the cost of less diverse behaviour. Finally, while the GPU outperforms the CPU for monodisperse simulation, we note that polydisperse granular simulation has only been addressed on the CPU. Nevertheless, most of the behavioural diversity associated with polydisperse simulation is handled adequately by our multiparticle sand framework, as corroborated by the physical validation carried out in Section 7.2.

[5] Comparing *different* simulations, based on their (fixed) Cundall numbers, assumes an ideal situation, namely, these simulations demonstrate linear scaling between computational time and grain count. If a simulation exceeds linear scaling, such as with larger grain counts, then its Cundall number decreases for these values. Therefore, favouring previous work, we calculate their Cundall numbers based on available data for the *smallest* grain count (usually implying the smallest computational time). This scheme ensures that any comparison to previous work represents the strongest criticism of our results.

7.3.3 Visualisation and Lighting

In all previous analyses, we have measured simulation performance without rendering cost. However, when rendering sand with simple Phong illumination, as discussed in Chapter 5, we record the performance results illustrated in Figure 7.22. This indicates that simple visualisation adds an almost negligible overhead to the simulation framework. Moreover, the total cost of simulation and rendering is linear in the grain count.

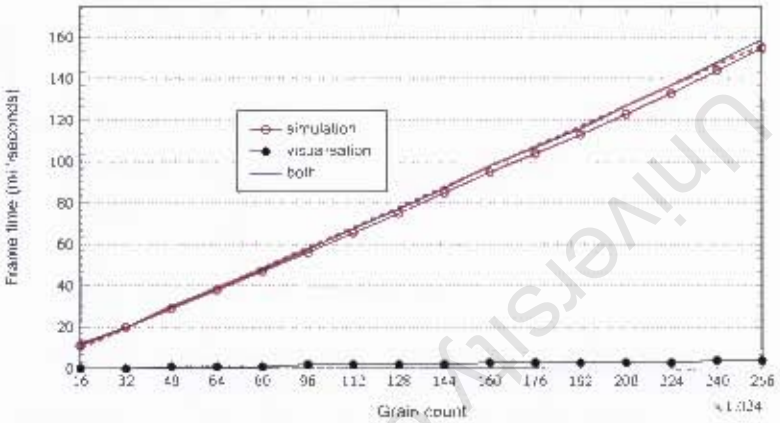


Figure 7.22: Simulation performance with visualisation active. The linear fit (red dashed line) is given for the cost of simulation and rendering running together.

However, the Phong illumination model lacks important effects, such as granule and sand self-shadowing, as well as environmental shadowing. What cost does our high-quality shadowing model exact on simulation performance?

We collect data on a stationary sandpile of 128K grains using the same setup as before. A single light source is used, with a view of the entire scene, but ignoring box sidewalls (that is, treating these effectively as nonreflective nonrefractive glass). In Figure 7.23, we provide hardware performance data collected from two simulation cases, one using lighting alone and the other including shadow mapping. The rendering performance, however, reveals no significant differences between the two. In particular, we find a difference of only 2 frame per second between the two cases.

Nevertheless, the difference in the frame rate may be explained, in part, by greater oscillating behaviour in CPU use under shadowing conditions. Under simple lighting conditions, the simulation framework issues five rendering batches (passes), namely, grid construction, interparticle force calculation, rigid body property updates, particle property updates, and particle rendering. This necessitates the allocation of 1,310,720 pixel primitives (or texels) and a matching number of vertices. Shadowing, however, adds one more pass, namely, shadow map construction, which requires 1,572,864 pixel primitives, where the additional primitives store the shadow map. Thus, we are able to explain the decrease in frame rate by increased CPU-GPU coupling. That is, we infer that real-time interactivity remains dependent on grain quantity and independent of illumination quality. However, the density of the shadow map, that is, the precision of the depth

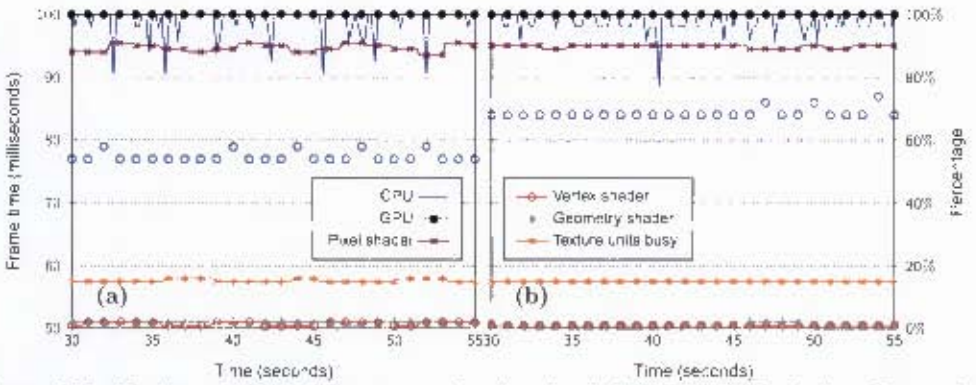


Figure 7.23: Hardware performance measured under simple lighting and shadowing. Frame time, indicated by a, is plotted with respect to the leftmost vertical axis. All other variables (indicated in the legend) are measured against the rightmost vertical axis, as percentage use of their respective resources. The graph (a) indicates simple lighting conditions, while (b) shows the same variables measured with shadow mapping enabled.

texture, determines both shadow quality as well as shadow map construction rate.

An example of the visual results achievable with real-time shadow mapping is shown in Figure 7.24. The images in the figure are screen captures taken during the funnel flow experiment. To aid visibility the funnel itself is not visualised, but remains physically present, as in the experimental setup (see Figure 7.7). The example demonstrates sand shadowing both itself and its environment.

Figure 7.24: Hourglass with real-time lighting and shadows. The slight "spattering" of granules, seen in the top left image, resulted from dropping the sand into the funnel at the start of simulation.

7.4 General rigid body simulation

While the DEM model for sand (see Chapter 3) is able to reproduce complex granular behaviour, it is also applicable to rigid body simulation *sans* sand. Indeed, we handle the mechanics of rigid bodies and granules similarly: through particle-particle collision processing.

Granular physics aside, the difference between our framework and a general rigid body simulation is our superior performance when dealing with a very large number of small particulate structures (like tetrahedral granules). In addition, we have described a mechanism for accelerated lighting of these structures (see Chapter 5).

Of course, the end result is an unattractively bumpy surface, but this is remedied by rendering, for each rigid body, its triangulated mesh, instead of its particle-sampled counterpart. This requires a trivial extension, where we rotate and translate the triangulated geometry of each rigid body, when it is rendered, according to the body's current orientation and position, respectively.

In Figure 7.25, we show an example of this type of simulation, where a metal ball is shot into a stack of boxes. Inside the framework, this scene consists entirely of particulate geometry, where all interactions, among the floor, glass walls, metal ball, and boxes, are handled through interparticle collisions.

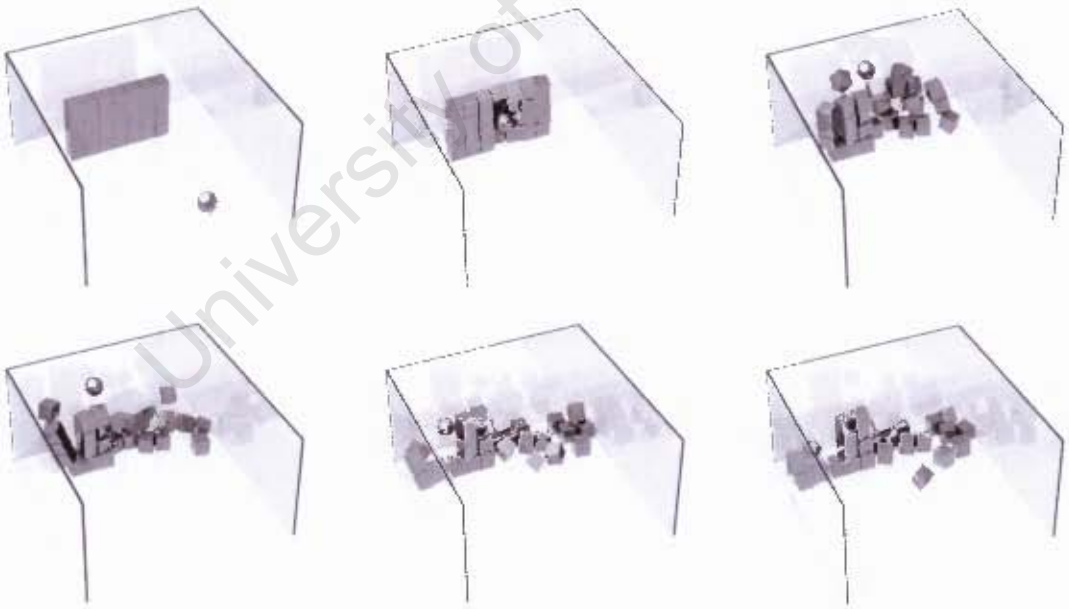


Figure 7.25: Bowling

7.5 Summary

In this chapter, we have seen that the sand simulation framework is able to produce physically valid granular behaviour, at interactive frame rates, using high-quality real-time lighting with

shadows. Testing has shown that the simulated sand reproduces many important properties of its real-world counterpart. Our results also support previous work in granular material science, where we have corroborated recent work on force distributions, as well as reproduced stress-strain behaviour in line with experiment, where previous DEM simulation has failed. Along with the reproduction of well-known behaviours like constant mean flow rate, our results support the use of multiparticle models for modelling the characteristic behaviours of granular materials, like sand.

We have demonstrated that adding more sand into the simulation leads to a predictable *linear* increase in simulation and rendering time. The correlation between grain quantity and simulation time was explained by corresponding increases in the amount of physically-based calculation, due to increases in the number of pixel shader instances computing forces between particles. Thus, if future trends in hardware continue to add more shader cores, as well as maintain texture unit bandwidth at a sufficient size to serve these cores, we can expect to simulate much larger volumes of sand at the same speeds. Finally, producing high-quality sand visualisation, with lighting and shadows, exacts an almost negligible cost on total simulation time.

Chapter 8

Conclusion and future work

Many real-time computer games contain virtual worlds built upon terrestrial landscapes, in particular, sandy terrains, such as deserts and beaches. Akin to real-world landscapes, these terrains often contain grass, trees, lakes, and rivers, and share their borders with oceans. These terrain elements contribute to enhancing realism, particularly when sophisticated material simulation is used, allowing plants to sway with the wind and water to ripple when disturbed. Unfortunately, the same is not true of the underlying terrain itself, which remains inert, uninteresting, and nongranular.

The problem is that present techniques employ low level-of-detail (LOD) models that represent sandy terrain as a flat textured surface, producing computationally cheap, but poor quality sand. In contrast, high LOD techniques, such as the distinct element methods (DEM), model sand at a much finer granularity, thereby reproducing interesting and expected granular behaviour. The catch is that DEM exacts a large computational cost from having to update and store the mechanical properties of millions of individual granules, many times each second.

Our approach to this problem has been inspired by many current techniques that exploit the processing power of the modern GPU, to make possible the real-time in-game simulation of fluid, fire, and cloth, on a single PC. While other techniques have addressed simulation of grain assemblies on the GPU, this thesis goes further in creating a complete framework for physically-accurate simulation of granular material inside a general environment.

8.1 GPU-accelerated granular physics

A major problem facing single particle models of granular material is the computationally expensive static friction handling needed to produce pile formation. A major theoretical contribution of this thesis is a multiparticle model, based on the work of Bell et al. [BYM05],

tailored to address this problem. For grains, we choose the smallest possible *multi*particle arrangement having three-dimensional symmetry: a regular tetrahedron comprised of four particles. Its symmetry, in particular, allows us to accelerate orientation and direction dependent physically-based calculation. Based on this grain structure, we have described a DEM-based multiparticle simulation model that reproduces static friction *intrinsically*, through locking behaviour among irregular grain geometry. In addition, we have presented a force model tailored to simulating complex (nonlinear) sand behaviour, yet simple enough that it is expressed easily, through pairwise (particle-particle) contacts among grains.

As a practical contribution, we have achieved real-time sand simulation by performing expensive physics calculations (needed by our physically-based DEM model) entirely on the GPU. These improvements include collision detection and grain property updates, which are performed in the rendering pipeline. In addition, we have described a novel approach for representing simple surfaces implicitly, as particles. The advantage of this approach is decreased texture memory consumption, which frees up space for storing explicitly represented bodies and granules.

8.2 GPU-accelerated rigid body physics

Another practical contribution is our GPU-accelerated mesh-sampling process, which enables the rapid transformation of an arbitrary closed mesh into its particle-based representation. The particulate surfaces that result can interact both among themselves and with sand, through pairwise contact handling on the GPU. Our specific contributions in this area are the following:

- We modified an existing GPU-based sampling technique [ED06], so that computationally expensive triangle processing is now offloaded to the geometry stage of the modern GPU. This includes handling bounding box construction and plane intersection, inside a geometry program. The resulting computation is less CPU-bound, leading to rapid generation of the signed-distance field (SDF).

- We implemented a GPU-based particle sampler, which simulates the constrained motion of particles over an implicit surface (namely, the zero subfield) within the SDF. This allows for an even sampling of the mesh, given sufficient particle density, as particles repel one another, thus diffusing evenly over the surface.

- We implemented a hybrid visualisation technique that combines ray-casting into the SDF volume with rendering of the evolving sampling process. Since these are run simultaneously, we have accelerated the visualisation by depth-based ray culling. Specifically, we ignore rays that intersect the implicit surface in areas overlapped by particles. Our visualisation allows a user to see whether particle density is unnecessarily high or low, without having to run rigid body simulation.

The result is a GPU-accelerated framework equipped with an easy mechanism for introducing arbitrary closed meshes into a simulation. This leads to a framework well-suited to general rigid body simulation, with or without the presence of granular material. Furthermore, when a large quantity of identical rigid bodies need to be simulated, the framework is highly attractive, both in its GPU-accelerated handling of these structures and for its real-time lighting performance.

8.3 Real-time lighting

Another major practical contribution is an illumination model that mitigates the computationally expensive lighting needed with large grain counts. We have described a method to accelerate granule rendition by exploiting its underlying particle representation. Specifically, we have used an implicit sphere representation to produce real-time surface reflection and shadowing, where previously, lighting of complex explicit geometry was needed.

8.4 Sand simulation

Our framework reproduces many important physical properties of real sand, including dynamic flow behaviour and response to extrinsic forces, as results from compression or projectile collision.

For real-time visual effects involving granular materials, the framework's appeal lies in the linear scaling between grain quantity and frame time. In addition, being able to handle large sand volumes of up to one million granules, suggests larger offline applications, such as visual effects for film. Provided GPU hardware development continues to add more processing (shader) cores and larger supporting texture memory interfaces, we expect the framework will handle and evolve much larger sand volumes at a similar rate.

Further, many useful scientific applications exist in the granular sciences, where these materials are studied for their static and rheological properties. In particular, the simulation framework has been shown to agree with previous scientific research dealing with force distributions, spatial force-force correlations, and internal sandpile geometry. The simulation has also demonstrated important dynamic behaviours, including both elastic hysteresis and strain-loss in the unloading phase of compression; constant mean flow rate of grains in an hourglass; and pile formation with an angle of repose predictably dependent on intergranule friction. Our results support the use of the sand simulation framework as a means to produce physically valid granular behaviour, which conceivably admits real-world applications. These include diverse modelling applications, from settling of corn grains during transport and storage to the toning process in electro-photographic copiers.

8.5 Future work

The long-term view of this thesis is towards incorporating the high level-of-detail granular simulation, as presented here, into future research encompassing a full level-of-detail (LOD) model of sandy terrain. The major impediment remains a limit in grain quantity. The sand volume represented in the simulation amounts to only a small section of sandy terrain. Thus, the challenge will be to integrate multiple terrain divots into a seamless whole, managed according to a LOD model. In particular, we expect that GPU simulation would produce localised granular simulation, under high LOD conditions, while coarser representations would substitute in low LOD conditions, such as for sand seen at a distance. In the latter case, height fields might be used to supplant large sand areas with a textured, grossly modifiable surface that LOD management might disassemble dynamically into a particulate representation, when grain-level simulation is required.

A more ambitious project is to bring together large-scale simulation, achieved in continuum (fluid) mechanics models, with finer models, such as DEM. On one hand, there is no complete continuum theory accurately predicting macroscopic granular features and only a few microscopic models. On the other hand, the DEM approach provides grain-level accuracy, at a high computational cost. The goal then is to develop a model that predicts macroscopic flow behaviour, while providing micro- or macroscopic (grain-level) property estimation. However, special considerations are needed at the grain-level, as a continuum-based solution is not well posed to handle nonaffine granular motion.

Bibliography

[ABC+07] C. Allen, D. Bloom, J. Cohen, and L. Treweek. Rendering tons of sand. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2007.

[Ang06] E. Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley, 4 edition, 2006.

[Ans04] K. Anstreicher. The thirteen spheres: A new proof. *Discrete and Computational Geometry*, 31(4):613–625, 2004.

[ASA07] N. Anh, A. Sourin, and P. Aswani. Physically based hydraulic erosion simulation on graphics processing unit. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, pages 257–264. ACM New York, NY, USA, 2007.

[B+07] P. Brown et al. Vertex program extension specification. *Revision 46*, 25 July 2007. `http://www.opengl.org/registry/specs/ARB/vertex_program.txt`.

[BA05] J. Bærentzen and H. Aanæs. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, pages 243–253, 2005.

[Bar97] D. Baraff. An introduction to physically based modeling: Rigid body simulation II: Nonpenetration constraints. *SIGGRAPH Course Notes*, 1997.

[Bat00] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

[BB90] G. W. Baxter and R. P. Behringer. Cellular automata models of granular flow. *Physical Review A*, 42(2):1017–1020, Jul 1990.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[BD98] L. Brendel and S. Dippel. Lasting contacts in molecular dynamics simulations. *Physics of Dry Granular Media*, pages 313–318, 1998.

[BEH+03] J. Brujic, S. F. Edwards, I. Hopkinson, and H. A. Makse. Measuring the distribution of interdroplet forces in a compressed emulsion system. *Physica A: Statistical Mechanics and its Applications*, 327(3-4):201–212, 2003.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, pages 777–786. ACM, 2004.

[Bli77] J. Blinn. Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics*, 11(2):192–198, 1977.

[BLvdV61] W. Beverloo, H. Leniger, and J. van de Velde. The flow of granular material through orifices. *Chemical Engineering Science*, 15:260–269, 1961.

[BMF05] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *ACM SIGGRAPH 2005 Courses*, page 3. ACM, 2005.

[BMF07] R. Bridson and M. Müller-Fischer. Fluid simulation: SIGGRAPH 2007 course notes. In *International Conference on Computer Graphics and Interactive Techniques*, pages 1–81. ACM Press New York, NY, USA, 2007.

[BMM+01] D. L. Blair, N. W. Mueggenburg, A. H. Marshall, H. M. Jaeger, and S. R. Nagel. Force distributions in three-dimensional granular assemblies: Effects of packing order and interparticle friction. *Physical Review E*, 63(4):041304, Mar 2001.

[Bro08] P. Brown. Texture buffer object extension specification. *Revision 6*, 30 June 2008. `http://www.opengl.org/registry/specs/ARB/texture_buffer_object.txt`.

[BS08] P. Brown and M. Strauss. NVIDIA Depth buffer extension specification. *Revision 9*, 6 August 2008. `http://www.opengl.org/registry/specs/NV/depth_buffer_float.txt`.

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

[BW98] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 43–54, 1998.

[BYM05] N. Bell, Y. Yu, and P. J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 77–86, July 2005.

[Car70] M. A. Carrigy. Experiments on the angles of repose of granular materials. *Sedimentology*, 14(3-4):147–158, 1970.

[CD98] B. Chopard and M. Droz. *Cellular automata modeling of physical systems.* Cambridge University Press New York, 1998.

[CE09] C. Coetzee and D. Els. Calibration of granular material parameters for DEM modelling and numerical verification by blade-granular material interaction. *Journal of Terramechanics*, 46(1):15–26, 2009.

[CFL67] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215, 1967.

[CHC+06] N. Carr, J. Hoberock, K. Crane, and J. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209. Canadian Information Processing Society Toronto, Canada, 2006.

[CHP+96] A. Carrillo, D. Horner, J. Peters, and J. West. Design of a large scale discrete element soil model for high performance computing systems. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society Washington, DC, USA, 1996.

[Cle09] P. Cleary. Ball motion, axial segregation and power consumption in a full scale two chamber cement mill. *Minerals Engineering*, 2009.

[CLH96] B. Chanclou, A. Luciani, and A. Habibi. Physical models of loose soils dynamically marked by a movingobject. In *Computer Animation'96. Proceedings*, pages 27–35, 1996.

[Cra03] M. Craighead. NVIDIA point sprite specification. *Revision 1.3*, 6 March 2003. `http://www.opengl.org/registry/specs/NV/point_sprite.txt`.

[Cro77] F. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):242–248, 1977.

[CS79] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, 29:47–65, 1979.

[CUD08] *Compute Unified Device Architecture.* 2008. `http://developer.nvidia.com/cuda`.

[DBM07] E. Dorjgotov, B. Benes, and K. Madhavan. An immersive granular material visualization system with haptic feedback. *Theory and Practice of Computer Graphics'07*, 2007.

[dBvKO+00] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, and M. Overmars. *Computational Geometry (2nd Edition).* Springer-Verlag, 2000.

[DCB+04] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng. Real-time voxelization for complex polygonal models. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 43–50, 2004.

[dFdMT+92] L. H. de Figueiredo, J. de Miranda, D. Terzopoulos, and L. Velho. Physically-based methods for polygonization of implicit surfaces. In *Proceedings of the conference on Graphics interface '92*, pages 250–257, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[Dru08] E. Drumwright. A fast and stable penalty method for rigid body simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):231–240, 2008.

[ED06] K. Erleben and H. Dohlmann. Scan conversion of signed distance fields. *Proceedings of DSAGM*, pages 81–91, 2006.

[EHK+04] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, page 29. ACM, 2004.

[Faz07] S. Fazekas. *Distinct Element Simulations of Granular Materials.* PhD thesis, Budapest University of Technology and Economics, Department of Theoretical Physics, 2007.

[FB74] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[FD81] D. Freedman and P. Diaconis. On the histogram as a density estimator: L_2 theory. *Probability Theory and Related Fields*, 57(4):453–476, 1981.

[Feh69] E. Fehlberg. Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems. *NASA Tech. Rep. TR R-315, Marshall Space Flight Center, Atlanta, GA*, 1969.

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

[Fer04] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Pearson Higher Education, 2004.

[FK03] R. Fernando and M. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[FLS63] R. Feynman, R. Leighton, and M. Sands. *The Feynman Lectures on Physics; Vol. I.* 1963.

[GBF03] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics*, 22(3):871–878, July 2003.

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

[GH97] Y. Grasselli and H. J. Herrmann. On the angles of dry granular heaps. *Physica A: Statistical and Theoretical Physics*, 246(3-4):301–312, 1997.

[GHL+01] J. Geng, D. Howell, E. Longhi, R. P. Behringer, G. Reydellet, L. Vanel, E. Clément, and S. Luding. Footprints in sand: The response of a granular material to local perturbations. *Physical Review Letters*, 87(3):035506, Jul 2001.

[GJD05] F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics*, volume 2005, 2005.

[Gla89] A. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[GLE08] *The OpenGL Extension Wrangler Library*. 2008. `http://www.opengl.org/sdk/libs/GLEW`.

[GS08] D. Goddeke and R. Strzodka. Performance and accuracy of hardware-oriented native, emulated and mixed-precision solvers in FEM simulations (Part 2: Double precision GPUs). Technical Report 370, Fakultat fur Mathematik, Technische Universitat Dortmund, 2008.

[GST05] D. Goddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. *Proceedings of ASIM*, 2005.

[GST07] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):221–256, 2007.

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

[Hai06] E. Haines. An Introductory Tour of Interactive Rendering. *IEEE Computer Graphics and Applications*, 26(1):76–87, 2006.

[Hal08] T. Halfhill. Parallel processing with CUDA. *Microprocessor Report*, 1(28):08–01, 2008.

[Har05] M. Harris. Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Courses*, page 220. ACM, 2005.

[HBS+03] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, 2003.

[HCA06] R. Hammerstone, M. Craighead, and K. Akeley. Vertex buffer object extension specification. *Revision 0.93*, 4 November 2006. `http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt`.

[HIKL+99] K. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[HK06] T. Harada and S. Koshizuka. Real-time cloth simulation interacting with deforming high-resolution models. In *ACM SIGGRAPH 2006 Research posters*, page 129. ACM, 2006.

[HKK07] T. Harada, S. Koshizuka, and Y. Kawaguchi. Sliced data structure for particle-based simulations on GPUs. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, pages 55–62. ACM New York, NY, USA, 2007.

[HL98] H. J. Herrmann and S. Luding. Modeling granular media on the computer. *Continuum Mechanics and Thermodynamics*, 10:189–231, 1998.

[HQL01] D. Hong, P. Quinn, and S. Luding. Reverse Brazil nut problem: Competition between percolation and condensation. *Physical Review Letters*, 86(15):3423–3426, 2001.

[HR01] D. Hirshfeld and D. Rapaport. Granular flow from a silo: Discrete-particle simulations in three dimensions. *The European Physical Journal E: Soft Matter and Biological Physics*, 4(2):193–199, 2001.

[HRW93] D. Halliday, R. Resnick, and J. Walker. *Fundamentals of Physics, Extended, with Modern Physics*. Wiley. New York. US, 1993.

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

[HWC60] G. Hardy, E. Wright, and J. Cofman. *An Introduction to the Theory of Numbers*. Clarendon Press Oxford, 1960.

[IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 1985.

[JBS06] M. Jones, J. Bærentzen, and M. Sramek. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, pages 518–599, 2006.

[JN92] H. M. Jaeger and S. R. Nagel. Physics of the granular state. *Science*, 255(5051):1523–1531, 1992.

[JNB96] H. M. Jaeger, S. R. Nagel, and R. P. Behringer. Granular solids, liquids, and gases. *Review of Modern Physics*, 68(4):1259–1273, Oct 1996.

[JS83] J. T. Jenkins and S. B. Savage. A theory for the rapid flow of identical, smooth, nearly elastic, spherical particles. *Journal of Fluid Mechanics*, 130:187–202, 1983.

[JS08] J. Juliano and J. Sandmel. Framebuffer object extension specification. *Revision 120*, 22 April 2008. `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`.

[Kal06] M. Kallay. Computing the moment of inertia of a solid defined by a triangle mesh. *Journal of Graphics Tools*, 11(2):51–57, 2006.

[KBR08] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language. *Language Version 1.30.08*, 1, 7 August 2008. `http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.pdf`.

[KHS04] M. Koster, J. Haber, and H. Seidel. Real-time rendering of human hair using programmable graphics hardware. In *Computer Graphics International (CGI)*, pages 248–256, 2004.

[KK87] G. Kuwabara and K. Kono. Restitution coefficient in a collision between two spheres. *Japanese Journal of Applied Physics*, 26(8):1230–1233, 1987.

[KK95] R. Knecht and G. Kohring. Dynamic load balancing for the simulation of granular materials. In *Proceedings of the 9th international conference on Supercomputing*, pages 164–169. ACM New York, NY, USA, 1995.

[KKK+05] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. *Graphics Hardware*, pages 123–132, 2004.

[KLS+05] J. Kniss, A. Lefohn, R. Strzodka, S. Sengupta, and J. D. Owens. Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches*, page 16. ACM, 2005.

[KM58] E. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, pages 457–481, 1958.

[KN01] T. Kim and U. Neumann. Opacity shadow maps. In *Rendering Techniques 2001: Proceedings of the Eurographics Workshop in London, United Kingdom, June 25-27, 2001*. Springer, 2001.

[KS05] M. J. Kilgard and G. Stahl. Texture rectangle extension specification. *Revision 1.21*, 4 October 2005. `http://www.opengl.org/registry/specs/ARB/texture_rectangle.txt`.

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

[KvdDP03] D. Knott, K. van den Doel, and D. Pai. Particle system collision detection using graphics hardware. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. ACM, 2003.

[L+06] B. Lipchak et al. Fragment program extension specification. *Revision 27*, 4 November 2006. `http://www.opengl.org/registry/specs/ARB/fragment_program.txt`.

[LBW08] B. Lichtenbelt, P. Brown, and E. Werness. Transform feedback extension specification. *Revision 5*, 28 February 2008. `http://www.opengl.org/registry/specs/EXT/transform_feedback.txt`.

[LC87] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–169. ACM New York, NY, USA, 1987.

[LCG+07] K. LaMarche, S. Conway, B. Glasser, and T. Shinbrot. Cellular automata model of gravity-driven granular flows. *Granular Matter*, 9(3):219–229, 2007.

[LH93] J. Lee and H. Herrmann. Angle of repose and angle of marginal stability: molecular dynamics of granular particles. *Journal of Physics A: Mathematical and General*, 26(2):373–383, 1993.

[LL86] L. Landau and E. Lifshitz. *Theory of Elasticity: Volume 7*. 1986.

[Lla07] I. Llamas. Real-time voxelization of triangle meshes on the GPU. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2007.

[LLW04] Y. Liu, X. Liu, and E. Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications*, pages 247–256. IEEE Computer Society, 2004.

[LM93] X. Li and J. M. Moshell. Modeling soil: realtime dynamic models for soil slippage and manipulation. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 361–368, 1993.

[LMF99] G. Lovoll, K. J. Måløy, and E. G. Flekkøy. Force measurements on static granular materials. *Physical Review E*, 60(5):5872–5878, Nov 1999.

[LMV92] A. Lepschy, G. Mian, and U. Viaro. Feedback control in ancient water and mechanical clocks. *IEEE Transactions on Education*, 35(1):3–10, 1992.

[Lou08] W. J. Loudon. *An Elementary Treatise on Rigid Dynamics.* BiblioLife, 2008.

[LS07] F. Labelle and J. Shewchuk. Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. In *International Conference on Computer Graphics and Interactive Techniques.* ACM Press New York, NY, USA, 2007.

[LSK+06] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[Mah02] W. Mahaney. *Atlas of Sand Grain Surface Textures and Applications.* Oxford University Press, 2002.

[MB05] T. McReynolds and D. Blythe. *Advanced Graphics Programming Using OpenGL.* Morgan Kaufmann, 2005.

[MDH07] X. Mei, P. Decaudin, and B. Hu. Fast hydraulic erosion simulation and visualization on GPU. In *15th Pacific Conference on Computer Graphics and Applications*, pages 47–56, 2007.

[Mir96] B. Mirtich. Impulse-based dynamic simulation of rigid body systems. Master's thesis, University of California Berkley, 1996.

[MJA+07] C. Mankoc, A. Janda, R. Arévalo, J. Pastor, I. Zuriguel, A. Garcimartín, and D. Maza. The flow rate of granular materials through an orifice. *Granular Matter*, 9(6):407–414, 2007.

[MJN98] D. Mueth, H. Jaeger, and S. Nagel. Force distribution in a granular medium. *Physical Review E*, 57(3):3164–3169, 1998.

[MLH00] H. Matuttis, S. Luding, and H. Herrmann. Discrete element simulations of dense packings and heaps made of spherical and non-spherical particles. *Powder Technology*, 109(1-3):278–292, 2000.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[MSW+08] H. Meier, M. Schlemmer, C. Wagner, A. Kerren, H. Hagen, E. Kuhl, and P. Steinmann. Visualization of particle interactions in granular media. *IEEE Transactions on Visualization and Computer Graphics*, 14(5):1110–1125, Sept.-Oct. 2008.

[MTP+04] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *ACM SIGGRAPH 2004 Papers*, pages 787–795. ACM, 2004.

[NBG+08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. 2008.

[NDM04] A. Ngan, F. Durand, and W. Matusik. Experimental validation of analytical BRDF models. In *International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2004.

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

[Ngu08] H. Nguyen. *GPU Gems 3*. Addison-Wesley, 2008.

[Nor06] J. Noren. Real-time rendering of granular materials. Technical report, Umeå University, Department of Computer Science, Sweden, 2006. `http://www.cs.umu.se/education/examina/Rapporter/JohanNoren.pdf`.

[NTS+82] R. Nedderman, U. Tüzün, S. Savage, and G. Houlsby. The flow of granular materials I: discharge rates from hoppers. *Chemical Engineering Science*, 37(11):1597–1609, 1982.

[NVI06] *NVIDIA GPU Programming Guide*. Revision 2.50, 2006. `http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf`.

[NVI07] *NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture*. 2007. `http://developer.download.nvidia.com/opengl/specs/g80specs.pdf`.

[NVI08a] *GeForce 8800 GTX specifications*. 2008. `http://www.nvidia.com/page/geforce_8800.html`.

[NVI08b] *NVIDIA GPU Programming Guide: GeForce 8 and 9 Series*. Revision 1.0, 2008. `http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf`.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113, 2007.

[OLL+02] C. S. O'Hern, S. A. Langer, A. J. Liu, and S. R. Nagel. Random packings of frictionless particles. *Physical Review Letters*, 88(7):075507, Jan 2002.

[ON03] K. Onoue and T. Nishita. Virtual sandbox. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 252. IEEE Computer Society, 2003.

[OR97] S. Ouaguenouni and J. Roux. Force distribution in frictionless granular packings at rigidity threshold. *Europhysics Letters*, 39(2):117–122, 1997.

[Owe07] J. Owens. GPU architecture overview. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2007.

[Pau02] B. Paul. Shadow extension specification. 14 February 2002. `http://www.opengl.org/registry/specs/ARB/shadow.txt`.

[PBM+05] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*. ACM Press New York, NY, USA, 2005.

[PF05] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*. Addison-Wesley Professional, 2005.

[Pho75] B. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PSO+06] N. Pohlman, B. Severson, J. Ottino, and R. Lueptow. Surface roughness effects in granular matter: Influence on angle of repose and the absence of segregation. *Physical Review E*, 73:031304, 2006.

[PT07] J. Pilgrim and T. Tornberg. How to build a sixty foot man of moving sand. In *ACM SIGGRAPH 2007 sketches: San Diego, California*, volume 5, 2007.

[PTV+92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge Univ. Press, 1992.

[PZvB+00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 335–342, 2000.

[Rap04] D. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[RBP+04] C. Radeke, K. Bagi, B. Palancz, and D. Stoyan. On probability distributions of contact force magnitudes in loaded dense granular media. *Granular Matter*, 6(1):17–26, 2004.

[RDG+04] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime isosurface extraction with graphics hardware. In *Proceedings of Eurographics*, 2004.

[RJM+96] F. Radjai, M. Jean, J.-J. Moreau, and S. Roux. Force distributions in dense two-dimensional granular systems. *Physical Review Letters*, 77(2):274–277, Jul 1996.

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 343–352, 2000.

[Ros06] R. Rost. *The OpenGL Shading Language (2nd Edition).* Ltd, 2006.

[RSD+97] F. Radjai, J. Schafer, S. Dippel, and D. Wolf. Collective friction of an array of particles: A crucial test for numerical algorithms. *J. Phys. I France*, 7:1053, 1997.

[RSP+87] A. Rosato, K. Strandburg, F. Prinz, and R. Swendsen. Why the Brazil nuts are on top: Size segregation of particulate matter by shaking. *Physical Review Letters*, 58(10):1038–1040, 1987.

[RWJ+98] F. Radjai, D. E. Wolf, M. Jean, and J.-J. Moreau. Bimodal character of stress transmission in granular packings. *Physical Review Letters*, 80(1):61–64, Jan 1998.

[SA07] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[SA08] M. Segal and A. Akeley. The OpenGL Graphics System: A Specification (Version 3.0). 11 August 2008. `http://www.opengl.org/registry/doc/glspec30.20080811.pdf`.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

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

[Sch97] G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques' 97*, pages 151–162, 1997.

[SDW96] J. Schäfer, S. Dippel, and D. Wolf. Force schemes in simulations of granular materials. *J. Phys. I France*, 6(5), 1996.

[SGG+07] A. Sud, N. Govindaraju, R. Gayle, E. Andersen, and D. Manocha. Surface distance maps. In *Proceedings of Graphics Interface 2007*, pages 35–42. ACM Press New York, NY, USA, 2007.

[SGL02] L. E. Silbert, G. S. Grest, and J. W. Landry. Statistics of the contact network in frictional and frictionless granular packings. *Physical Review E*, 66(6):061303, Dec 2002.

[Sho85] K. Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH Computer Graphics*, 19(3):245–254, 1985.

[SKvW+92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast shadows and lighting effects using texture mapping. *ACM SIGGRAPH Computer Graphics*, 26(2):249–252, 1992.

[SOH99] R. Sumner, J. O'Brien, and J. Hodgins. Animating sand, mud, and snow. *Computer Graphics Forum*, 18(1):17–26, 1999.

[TPG99] G. Treece, R. Prager, and A. Gee. Regularised marching tetrahedra: Improved iso-surface extraction. *Computers and Graphics*, 23(4):583–598, 1999.

[TSD07] N. Tatarchuk, J. Shopf, and C. DeCoro. Real-time isosurface extraction using the GPU programmable geometry pipeline. In *International Conference on Computer Graphics and Interactive Techniques*, pages 122–137. ACM New York, NY, USA, 2007.

[Tur91] G. Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 289–298, New York, NY, USA, 1991. ACM.

[TVC98] O. Tsoungui, D. Vallet, and J. Charmet. Use of contact area trace to study the force distributions inside 2D granular systems. *Granular Matter*, 1(2):65–69, 1998.

[TW00] A. V. Tkachenko and T. A. Witten. Stress in frictionless granular material: Adaptive network simulations. *Physical Review E*, 62(2):2510–2516, Aug 2000.

[VC07] J. Venetillo and W. Celes. GPU-based particle simulation with inter-collisions. *The Visual Computer*, 23(9):851–860, 2007.

[vEEvH+07] A. R. T. van Eerd, W. G. Ellenbroek, M. van Hecke, J. H. Snoeijer, and T. J. H. Vlugt. Tail of the contact force distribution in static granular materials. *Physical Review E*, 75(060302):060302(R), 2007.

[Ver67] L. Verlet. Computer "Experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98, Jul 1967.

[VHC+99] L. Vanel, D. Howell, D. Clark, R. P. Behringer, and E. Clément. Memories in sand: Experimental tests of construction history on stress distributions under sandpiles. *Physical Review E*, 60(5):R5040–R5043, Nov 1999.

[WH80] T. Whitted and N. Holmdel. An improved illumination model for shaded display. *Communications*, 1980.

[WH89] B. Werner and P. Haff. Dynamical simulations of granular materials using the Caltech hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 1313–1318. ACM New York, NY, USA, 1989.

[WH94] A. P. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pages 269–277. ACM, 1994.

[WHM87] J. Williams, G. Hocking, and G. Mustoe. The theoretical basis of the discrete element method. *Numerical Methods in Engineering, Theory, and Application*, 1987.

[Wil78] L. Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274, 1978.

[Wit77] J. Wittenburg. *Dynamics of systems of rigid bodies*. Stuttgart, 1977.

[WND97] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 1.1*. Addison-Wesley, 1997.

[WRD98] D. Wolf, F. Radjai, and S. Dippel. Dissipation in granular materials. *Philosophical Magazine Part B*, 77(5):1413–1425, 1998.

[WSB+01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, volume 20, pages 153–165. Blackwell Synergy, 2001.

[YHK08] R. Yasuda, T. Harada, and Y. Kawaguchi. Real-time simulation of granular materials using graphics hardware. *International Conference on Computer Graphics, Imaging and Visualization*, pages 28–31, 2008.

[ZB05] Y. Zhu and R. Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24:965–972, 2005.

[ZLW+06] J. Zhou, S. Long, Q. Wang, and A. D. Dinsmore. Measurement of sources inside a three-dimensional pile of frictionless droplets. *Science*, 312(5780):1631–1633, 2006.

[ZTY+07] Y. Zeng, C. Tan, M. Yang, C. Chiang, C. Chang, and W. Tai. A momentum-based deformation system for granular material. *Computer Animation and Virtual Worlds*, 18(4-5):289–300, 2007.

[ZvBG01] M. Zwicker, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 371–378, 2001.

[ZWF+03] Y. Zhao, X. Wei, Z. Fan, A. Kaufman, and H. Qin. Voxels on Fire. In *Proceedings of the 14th IEEE Visualization 2003*. IEEE Computer Society Washington, DC, USA, 2003.

[ZXY+01] Y. Zhou, B. Xu, A. Yu, and P. Zulli. Numerical investigation of the angle of repose of monosized spheres. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 64(2), 2001.

[ZXY+02] Y. C. Zhou, B. H. Xu, A. B. Yu, and P. Zulli. An experimental and numerical study of the angle of repose of coarse spheres. *Powder Technology*, 125(1):45 – 54, 2002.

Appendix A

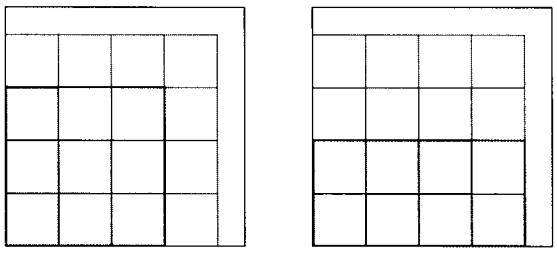
Proofs

A.1 Optimal texture size for flat 3D grids

As mentioned in Section 4.4, we hope to find the optimal texture size for representing a flat 3D grid. Specifically, viewing layers C_i as tiles, we are seeking texture dimensions (T_w, T_h) such that some tiling of that texture, using all k_d tiles, minimizes the number of wasted (unmapped) texels given by $W = T_w \cdot T_h - k_w \cdot k_h \cdot k_d$. Furthermore, since graphics cards support finite-sized textures, we assume that (T_w, T_h) fits within maximum texture dimensions, (M_w, M_h).

We shall also assume from now on that at least one tiling exists, whether optimal or not. If this is the case, then the solution given in this section will find the optimal tiling. Otherwise, no tiling exists, which is an easily identifiable case, as explained at the end of this section.

We define a *valid* texture as the smallest possible enclosing rectangle capable of holding all the tiles of some arrangement while remaining inside maximum texture boundaries. See figure A.1 for an illustration of two possible valid textures for $k_d = 7$ tiles.



(a) Unoptimal (b) Optimal

Figure A.1: Tile arrangements. Grey tiles represent layers C_i and white areas represent possible positions for these tiles. The light grey background square represents the maximum allocatable texture size (M_w, M_h). The blue rectangles represent the texture (T_w, T_h) allocated for enclosed tile arrangement. White space inside a blue rectangle represents "wasted" texture surface area, W.

We are now able to define

$$s = \lfloor \frac{M_w}{k_w} \rfloor, \tag{A.1}$$

$$t = \lfloor \frac{M_h}{k_h} \rfloor, \tag{A.2}$$

where s and t are the maximum quantity of tiles that can fit on the bottom row and left column, respectively, of a texture with dimensions (M_w, M_h). Our assumption that at least one tiling exists implies $s, t \geq 1$. Of course, the valid texture enclosing the resulting arrangement may be smaller than the maximum possible texture size, depending on whether the tiles fit exactly or not.

Now suppose b is the number of tiles on the bottom row of a tiling matched to some valid texture, since at least one such texture exists. Thus, b is an integer with $0 < b \leq s$. Furthermore, the division algorithm [HWC60] guarantees the existence of integers q and r such that $k_d = bq + r$ for $0 \leq r < b$. Since the tiling is a valid one, we must have $q \leq t$, where equality implies $r = 0$.

Visually, the previous paragraph says there exists some tiled rectangle of width bk_w and height qk_h, which fits inside the bounds (M_w, M_h). If $r = 0$, then all tiles were used to construct the rectangle. Thus the valid texture is allocated using the same dimensions as the rectangle it encloses, without any texel wastage, since $(W = 0)$. If $r > 0$, then we allocate a texture of the same width as before, but use a height of $(q + 1)k_h$ instead, since a qth row partially filled by r tiles is included. The resulting texture fits within the maximum bounds, since $q < t$.

A.1.1 The solution

Finding an optimal tiling, therefore, means checking for each b if $q \leq t$. If $q > t$, then b is not a possible solution and we try another. Otherwise, we must check make sure $r = 0$ when $q = t$. If the latter is in fact the case, then we have an optimal solution given by a valid texture (bk_w, qk_h) where no texels are wasted. Otherwise, we know that this particle value of b wastes $b - r$ tiles worth of texel space. We remember this value and continue checking other values of b until either we find a no-wastage situation or until no other possible values for b remain. We can then pick any b with a minimum value for $r - n$ and use a texture with dimensions $(bk_w, (q + 1)k_h)$. However, if we have found no b with a matching valid q, then no solution exists.

Finally, we mention that computing q and r is quick:

$$q = \lfloor \frac{k_d}{b} \rfloor, \tag{A.3}$$

$$r = k_d - qb. \tag{A.4}$$

A.2 Gaussian support size

If we use a Gaussian function as a kernel, then we must find the correct parameters so that the Gaussian is less than some floating-point value $f > 0$ for all points lying outside some cut-off distance $r > 0$. In particular, we wish to choose r so that the repulsion velocity, as discussed in Section 6.3, is calculated from those neighbour particles that lie within the particle's own voxel as well as the 26 voxels that surround it.

The Gaussian requires the setting of a constant scaling factor, central axis, and standard deviation. Theorem A.2.1 explains how to set these parameters to meet the constraints imposed by f and r. Here we are considering a three-dimensional Gaussian centred at the origin with same standard deviation for each coordinate axis. This ensures an unbiased weighting to particles in the surrounding space.

Theorem A.2.1. *Let $f, r > 0$ for some values $f, r \in R$ and let g be a three-dimensional Gaussian centred at the origin with standard deviations $\sigma = \sigma_x = \sigma_y = \sigma_z$ and a constant scaling factor $A > 0$. If $\sigma < \frac{r}{\sqrt{2\ln(\frac{A}{f})}}$ and $||(x, y, z)|| > r$ for some vector $(x, y, z) \in R^3$, then $g(x, y, z) < f$.*

Proof. Consider the three-dimensional Gaussian g given by

$$g(x, y, z) = Ae^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2} + \frac{(z-z_0)^2}{2\sigma_z^2}\right)}, \tag{A.5}$$

where $x_0 = y_0 = z_0 = 0$ so that g is centered at the origin. Let $\sigma = \sigma_x = \sigma_y = \sigma_z$ and $(x, y, z) \in R^3$ such that $||(x, y, z)|| > r$. Thus, we have

$$\begin{aligned} g(x, y, z) &= Ae^{-\left(\frac{x^2+y^2+z^2}{2\sigma^2}\right)}, \\ &= Ae^{-\left(\frac{\ln(A/f)||(x,y,z)||^2}{r^2}\right)}, \\ &< Ae^{-\ln(\frac{A}{f})}, \end{aligned} \tag{A.6}$$

which follows from $\frac{||(x,y,z)||^2}{r^2} > 1$. Equation A.6 now simplifies to $g < f$ as required.

□

Appendix B

Quaternions

The quaternions are a number system, technically termed a normed division algebra over the real numbers. They were introduced in computer graphics by Shoemaker [Sho85].

A quaternion $q = [\mathbf{v}, s]$ is usually described by vector $\mathbf{v}$ and scalar s. The conjugate of a quaternion is defined to be

$$q^* = [-\mathbf{v}, s]. \tag{B.1}$$

Addition of quaternions is defined as

$$q_1 + q_2 = [\mathbf{v}_1, s_1] + [\mathbf{v}_2, s_2] = [\mathbf{v}_1 + \mathbf{v}_2, s_1 + s_2], \tag{B.2}$$

while multiplcation is defines as

$$q_1 q_2 = [\mathbf{v}_1, s_1][\mathbf{v}_2, s_2] = [\mathbf{v}_1 \times \mathbf{v}_2 + s_1\mathbf{v}_1 + s_2\mathbf{v}_2, s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2], \tag{B.3}$$

where '$\times$' and '$\cdot$' refer to the usual 3-vector cross product and dot product, respectively. The norm or "length" of a quaternion is defined as

$$|q| = \sqrt{qq*} = \sqrt{||v||^2 + s^2}. \tag{B.4}$$

Thus, a unit quaternion is any non-zero quaternion q with $|q| = 1$. Any non-zero quaternion q can be made into a unit quaternion by dividing it by its norm.

An important property of unit quaternions is that they describe a three-dimensional rotation. Specifically, given a quaternion q such that $|q| = 1$, then we get the rotation matrix

$$\mathbf{R} = \begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2sz & 2xz + 2sy \\ 2xy + 2sz & 1 - 2x^2 - 2z^2 & 2yz - 2sx \\ 2xz - 2sy & 2yz + 2sx & 1 - 2x^2 - 2y^2 \end{pmatrix} \tag{B.5}$$

In many graphics API's such as OpenGL, a rotation is usually given in the form of an angle[1] θ and the rotation axis specified by a unit vector $\mathbf{n}$. The formula that relates this axis and angle to the quaternion is

$$q = [sin(\frac{\theta}{2})\mathbf{n}, cos(\frac{\theta}{2})]. \tag{B.6}$$

Another useful properties of the unit quaternion is that we can combine rotations directly, rather than having to convert to and from rotation matrix representation. That is, given unit quaternions q_1 and q_2 representing two rotations as described above, the unit quaternion q_c representing a rotation first by q_2, then by q_1, is

$$q_c = q_1 q_2. \tag{B.7}$$

This rotation can be applied to a vector without having to convert to a rotation matrix representation. That is, given a quaternion q representing a rotation we wish to apply to a vector v to get the rotated vector v_r, we can produce this rotation in quaternion notation by calculating qwq^*, where $w = [v, 0]$. Written another way, we get

$$[\mathbf{v}_r, 0] = [v_q, s_q][v, 0][-v_q, s_q]. \tag{B.8}$$

[1] We use radians.