Powder Technology 196 (2009) 139–146

Contents lists available at ScienceDirect

Powder Technology

journal homepage: www.elsevier.com/locate/powtec

Effective packing of 3-dimensional voxel-based arbitrarily shaped particles

Thomas Byholm, Martti Toivakka, Jan Westerholm *

Abo Akademi University, Laboratory of Paper Coating and Converting and Center for Functional Materials, Porthansgatan 3, FI-20500 Abo, Finland

ARTICLE INFO

Article history:
Received 8 August 2008
Received in revised form 16 January 2009
Accepted 12 July 2009
Available online 26 July 2009

Keywords:
Particle packing
Digital
Voxel
Optimisation
Memory

ABSTRACT

In many research areas including medicine and paper coating, packing of particles together with numerical simulation is used for understanding important material functionalities such as optical and mass transfer properties. Computational packing of particles allows for analysing those problems not possible or difficult to approach experimentally, e.g., the influence of various shapes and size distributions of particles. In this paper a voxel-based algorithm by Jia et al. [X. Jia, R.A. Williams, A packing algorithm for particles of arbitrary shapes, Powder Technology 2001, vol. 120, pp. 175–186.] enabling the packing of arbitrarily shaped particles, is memory- and speed-optimised to allow for simulating significantly larger problems than before. Algorithmic optimisation is carried out using particle shell area reduction decreasing the amount of time spent on collision detection, fast rotation routines including lookup tables, and a bit packing algorithm to utilise memory effectively. Presently several hundreds of thousands of complex arbitrarily shaped particles can be simulated on a desktop machine in a simulation box consisting of more than 10^9 voxels.

1. Introduction

In many different research areas including pharmaceutical science and paper coating there is a clear need to better understand how particles are packed when forming, e.g., a coating. A more detailed knowledge of the packing process would be beneficial in order to understand and optimize the microstructure of different kinds of porous media.

Traditionally analytical methods have been used, packing mathematically defined objects and applying various chemical and physical interactions in order to assess the relative movements and interactions of the objects. In this paper we will focus on computational methods for improving the speed and memory efficiency of a voxel based particle packing approach first presented in [1]. In voxel based representations, solid objects are discretized using elementary volume units, voxels, typically cubes, the 3-dimensional objects analogous to discretized 2-dimensional objects, pixels. The simulations in [1] are based on moving these voxel objects in different directions in space. For clarity, in Fig. 1 we have visualized two pixel objects in 2D-space. Any subsequent translation or rotation of these two objects will retain the approximate shape and size of each object, always filling or not filling a pixel and at most one particle in any given pixel.

Well done code optimisations can have a dramatic effect on the effectiveness of almost any algorithm [2], and hence we are interested in both representing the voxel objects as effectively in computer memory as possible in order to minimize memory usage, and also simulating the movement and interaction of voxel objects as quickly as possible. We will also look at how to optimise low level operations, deeply nested in the algorithm of [1], as these operations are good candidates for optimisation based on their usage frequency. Although many of these operations are fast, their frequency of use will add up to a considerable total time.

Furthermore, since it is often desired to use high resolutions when discretizing objects into voxels, there is a need for considering different kinds of real time data compression for the packing matrix, that is, the 3-dimensional space where the voxels are situated. Since voxel based packings operate on a large 3-dimensional array for representing the packing matrix, memory requirements tend to escalate very quickly. High resolutions are needed for accurately representing a wide particle size distribution as well as extreme aspect ratios. A straightforward way of representing the packing matrix is to use an array containing id-numbers for the respective particles as in Fig. 1. While this enhances speed by not having to erase a particle before moving it, it consumes large amounts of memory. To represent a 4000^3 array with 32-bit elements we would need 256 GB of memory. Furthermore, the spatial locality when accessing voxels in the packing matrix will be very poor and the deepest nested operations of the algorithm consist of intensive reading and writing of data. To overcome this problem, real time bit-packing is applied representing particle positions with 1 and void space with 0. Although a little bit more work need to be done like storing particle metadata separately and always removing them before trying a new position, the increased spatial locality and 32-fold reduced need for memory outweigh the extra steps. These techniques allow for packing of complex sets with arbitrarily shaped particles on normal 64-bit desktop machines. As the amount of computer memory increases steadily future desktops will be able to simulate even higher resolutions and larger

* Corresponding author.
E-mail address: jan.westerholm@abo.fi (J. Westerholm).

0032-5910/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.powtec.2009.07.013

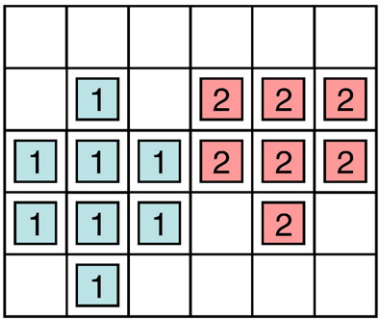

Fig. 1. 2-dimensional illustration of a packing matrix containing two distinct particles. Each particle is described by a set of pixels identified by the id of the particle.

packing matrices. Basic statistic output was added for the packed material, including porosity and rotational distributions of packed particles and RMS surface roughness including a graphical user interface for easy particle definition and batch processing.

We start by introducing the basics of the packing algorithm considered in this paper in Section 2. In Section 3, we consider in detail the different optimisations done to improve the time and memory efficiency of the algorithm. The main results are presented in Section 4 and the conclusions in Section 5.

2. Packing algorithm

The general structure of a particle simulation process can be divided into three logically separated tasks: geometric definition, particle movement rules and collision detection. In the geometric definition we define how particles are represented geometrically and which data structures are used to represent the particles. These auxiliary structures will have a significant impact on memory consumption and the efficiency of particle movements and collision detection. Particle movement rules should capture the essential physics of the packing process while keeping the logic of the rules reasonably simple. Thirdly, the collision detection is a central feature of the program inducing restrictions of particle movements. We will be particularly interested in developing fast collision detection methods by defining the particle geometry in such a way as to support this.

2.1. Particle and voxel data representation

In our particle packing simulations the volume within which particle movements are simulated, the packing matrix, is divided into 3-D unit cubes, voxels. A voxel is the 3D analogue to 2D pixels representing the finest granularity of volume available. Hence e.g. all particles consist of a collection of voxels, and any voxel within the simulation box is either empty or occupied by precisely one particle.

The main idea behind voxel based particle packing is to avoid the time consuming collision detection associated with analytical methods [3], where particles are represented as mathematical objects, a sphere for example being represented by its centre point and radius. Collision detection, or intersection test, is used to determine if a particle due to its movement will be overlapping any other particle, thus enabling us to decide if the particle movement will be allowed or not. In a simple implementation an intersection test has to be done for all voxels of every object in the simulation box. If an intersection with any other object is found, the two are said to collide and the move cannot be accepted. The intersection tests for analytically defined particles are very time consuming and depend largely on the type of object being checked for collision. Although this kind of approach can be optimised in many different ways including neighbour lists [4], octree representations [5] and bounding boxes, it is still very complex for large numbers of particles. The basic complexity is $0(n)$ meaning that n (number of particles) intersection tests have to be performed for every single particle move. Furthermore the complexity increases if we want to pack arbitrarily shaped objects using analytical methods since these are often represented using large numbers of polygons, meaning that the number of intersection tests needed is further increased, depending on the number of polygons used to describe every particle.

In Ref. [1] objects are not represented analytically, instead they are built up from voxels [5]. Every object can be approximated by a set of voxels arranged on a regular grid so that it represents the analytical shape as closely as possible. The desired accuracy can be varied by increasing the resolution, that is decreasing the voxel volume. All the particles simulated are inserted in a large packing matrix. As we are working with voxel data, all coordinates are represented using integers. The packing matrix is a simple 3-dimensional data array containing a particle id or a special void id at every element. When a particle is moved, it is removed from its previous voxel position and written into its new position in the array. The collision detection can then be done by checking all the new positions that the voxels of the particle in question will overlap. If any of these voxels already contains an id belonging to another particle, the move is denied and the particle is reset to its original position. This way we eliminate the analytical intersection tests and we do not need to scan through all particles as the collision detection is based on the packing matrix only.

To enable some of the optimisations that we will deal with in Section 3, all particles are also represented as metadata. For every particle we know the position of its centre point as defined by the user, its current rotation and the original voxel map, so that we can reconstruct it without the simulation box.

2.2. Particle movement

To create a particle packing, particles need to be introduced in one way or another into a simulation box. A simple approach would be to try random (x,y,z) positions until a position is found where the particle does not overlap with any other particle. This procedure is repeated until the desired packing density is reached. The problem with this method is that it is not possible to achieve high packing densities, as the particles are static after they have been added to the packing. This means that large gaps will be formed between particles which are difficult to fill, resulting in low packing densities.

To deal with this problem, various packing algorithms that move the particles according to predefined rules can be implemented, usually involving collision detection to ensure that particles are not moving into the domain of another particle. Since the goal of particle packing is often to make a densely packed structure, we have to find a set of particle movement rules that optimises the particle movement to achieve this goal. We are using the same fast statistical approach as in [1] to achieve this effect. The underlying principle is to introduce one or more particles every nth time step. They can be introduced using many different methods, resulting in slightly different packing structures, but the most straightforward way is to introduce new particles at random (x,y) positions at the top of the simulation box. For every time step, all particles are moved 1 voxel position along any axis (x,y,z) and rotated freely within a preset maximum value, which is set individually for all three axes relative to their initial rotation. The packing property of the algorithm is achieved by assuring that the particle movement is generally downwards according to the following rule. New particle positions are found by assigning a 50% probability to move in the negative directions and 50% probability for moving in the positive directions on the x-, y- or z-axis, allowing moves along every axis simultaneously. The maximum movement is 1 voxel in each direction. To simulate the downwards movement a rebounding probability between 0% and 100% is used, describing how high the probability to accept an upwards movement is. The upwards component of the move is accepted if and only if the rebounding

probability is also true. This means that a lower rebounding probability will result in particles moving more rapidly downwards. The reason for allowing upwards components at all is to enable the particles to escape a local minimum to find a more optimal position thus increasing packing density. This is in one way a simplification of the Monte-Carlo deposition method implementation developed by Vidal et al [6] that strives to find a global minimum by the use of an energy coefficient. Furthermore, a random rotation is also applied to the particle. The rebounding mechanism and rotations are the only basic rules that are used for this packing approach. To further simplify the algorithm, no real physical or chemical interactions are modelled. In the present approach we avoid looking at the packing process itself and instead we focus on the result it generates. Many particle packing algorithms simulate the actual chemical and physical interactions between particles and perhaps a fluid phase, trying to understand how the actual packing process works. We aim only at analysing the stabilised packed structure.

3. Optimisations

Before looking at specific optimisations we consider the general structure of the packing algorithm and try to identify the bottle necks. It is evident that the main efforts of the code optimisation should be focused on those parts of the packing algorithm where most of the time is spent, typically in a relatively limited section of the algorithm. It turns out that most of the time is spent rotating particles and accessing the packing matrix. When moving a particle we start by rotating the particle and then perform the collision detection. To do this, we need to rotate every voxel in the particle to their new positions and check these positions in the packing matrix to make sure that there is no voxel belonging to another particle at the same position. If the voxel is occupied, the particle will remain in its old position, otherwise it will be inscribed at the new position, resulting in additional access to the packing matrix. In Fig. 2 a flow diagram of the basic structure of the interesting part of the algorithm is presented and it is here that most of the time is spent. Moreover, the time to calculate particle translation and rotation values can be totally neglected; they are very fast operations compared to the operations needed for the rotation of all points in the particle and data access. The operation consists of generating random numbers and using those to decide rotation and translation. The actual moving process, on the other hand, normally consists of executing hundreds or even thousands of rotations and the same amount of reading and writing in the particle matrix. Although the latter are simple operations, it is the considerable number of times that they are executed for every move that make them interesting targets for optimisation. Furthermore, an analysis of packings of monodisperse spheres indicates that the percentage of the collision detections that results in a move can be as low as 30%, which is an important fact to take into account when doing optimisations. It is also preferable to avoid removing a particle from the simulation box before it is known if it can be moved or not. Moreover, it is essential to study how the number of moves can be reduced without affecting the end result. It might not, for example, be necessary to introduce new particles at the top of the packing matrix, since the particle movements are random anyway, and other particles will not have any effect before the new particles reach them. To compare speeds a test packing was used consisting of monodisperse spheres with 1 unit length diameters packed in a cubic packing box with the size of 10 unit lengths. Our various optimisation techniques are presented in Sections 3.1–3.7 and the overall results in Section 4.

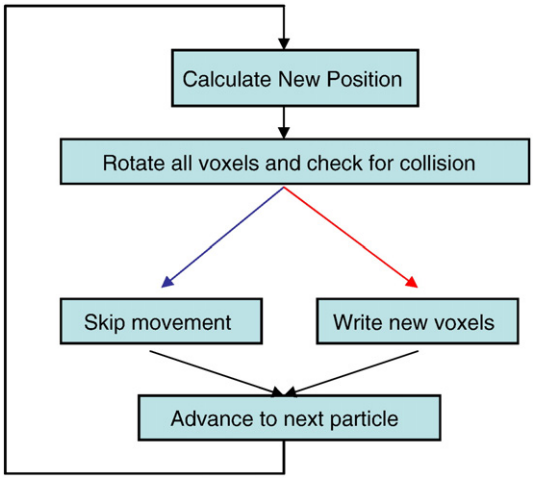

Fig. 2. The flow diagram of the most time consuming part of the algorithm. The calculation of new positions using random numbers can be disregarded from an optimisation point of view, instead it is the particle voxel manipulations and packing matrix accesses that need to be addressed.

3.1. Particle rotation

For every particle move a new rotation has to be calculated for every voxel that defines the particle. The standard way of rotating objects is to use rotation matrices [5]. The matrix, once set up, can be applied to all voxel positions in.

In the original paper shearing transformations are used for rotation [8]. The reason for using this method is that it is more exact, and we avoid creating holes in the particle structure. When rotating a set of voxels we will usually get several voxels rounded off to the same integer position as seen in Fig. 3 This is due to the rounding of floats to integer values and it is difficult to avoid using normal rotations. The shearing will in fact avoid these holes.

Another solution to the problem with holes is to reverse the rotation operation, looking at all possible result positions and backwards rotating those positions into the simulation box containing the original particle voxel data. Also this method eliminates holes in the structure. To accomplish this we define a parallelepiped around every particle before rotating it. This parallelepiped is calculated by rotating the outer corner coordinates of the bounding box of the particle. In this case, the bounding box will simply be the corners of the voxel matrix that defines the particle. This procedure ensures that all positions in the simulation box that might contain a voxel belonging to the particle in question will be checked. All positions inside this box relative to the packing matrix can then be rotated to original particle data, using the origin as reference point. If a particle voxel is found in the original data, the corresponding position in the simulation box can be assigned a voxel. This method is very slow since it requires rotations of all voxels contained in the bounding box, and even though we could optimise it by using a bounding sphere or similar, it would still be very time consuming.

Other optimisation methods identifying edges and rotating line segments between them have been reported by Chien et al. [9]. Here a

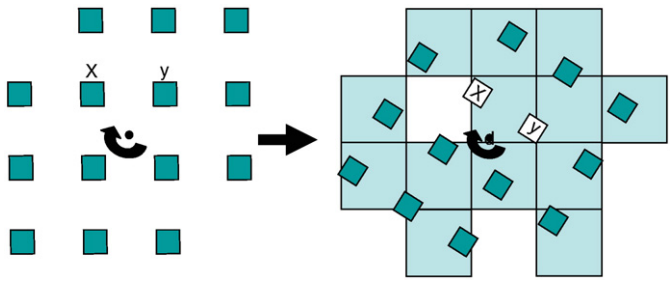

Fig. 3. Voxels, exemplified by the two neighbouring voxels named *x* and *y*, are rotated into their new positions, resulting in a hole in the structure denoted by the white square. The smaller squares illustrate the voxels being rotated and the larger squares the resulting positions.

faster way to do the manipulation is applied where we allow holes to form within the particle. The reasoning behind this decision is that particles are generally much larger than 1 voxel, and even if there are random 1 voxel holes in the surface, the particle would not be able to penetrate into another particle. When relaxing a constraint this way we need to represent only the voxels that belong to a particle in an array, interleaving (x,y,z) coordinates. Now we only need to rotate the coordinates in the array and check if those positions result in a collision.

As we are working with voxel data, all coordinates are represented by integers. This means that for every single coordinate rotated, we have to execute three float to integer calculations. The standard way of doing this internally is to set the correct rounding mode and then do the rounding operation. However, the setting of the rounding mode is a surprisingly slow operation which we avoid by using an inline assembler which omits the setting of the rounding word [10].

Instead of forming a composite rotation matrix in 3D with three independent rotation angles we use a fast and simple rotation routine which operates on coordinates centred at the origin and performs three consecutive rotations, first around the x-axis, then around the y-axis and finally around the z-axis. Basically we are using three 2D rotations [7] to calculate the result as seen in Fig. 4. The most important reason for doing this is that we can represent the particles easily so that the voxels are centred at the origin of the coordinate system of the particle, and we do not need to do any other transformations that would be more straightforward to include in a composite matrix. When using this method we are actually performing 12 multiplications as opposed to 9 in an optimised matrix rotation. Nevertheless, a difference in speed was not measurable, the most probable reason being the fact that memory accesses are dominating the process. Both methods can use pre-calculated sine and cosine values, meaning that it's only the multiplications and additions that will differ. These operations are very fast compared to memory access operations, especially when cache misses are involved.

There is an essential and easy optimisation that can be done at this stage. An arbitrary rotation in 3 dimensions contains three angles and hence there are 6 unique trigonometric calculations to be done, but these calculations can be avoided. They will slow down the rotations by a fair amount when taking into consideration that there are a large number of coordinates that have to be rotated for every move. This can easily be overcome by limiting the rotations degrees to integers.

```
x = Xo
y = Yo
z = Zo

//Rotate y  and z around the x-axis
ty = cos(AngleX) * y - sin(AngleX) * z
tz = sin(AngleX) * y + cos(AngleX) * z

y= ty
z= tz

//Rotate x and z around the y-axis
Zr = cos(AngleY) * z - sin(AngleY) * x
x  = sin(AngleY) * z + cos(AngleY) * x

//Rotate x and y around the z-axis
Xr = cos(AngleZ) * x - sin(AngleZ) * y
Yr = sin(AngleZ) * x + cos(AngleZ) * y
```

Fig. 4. Rotation routine. Using three consecutive 1-dimensional rotations, a point (X_o,Y_o, Z_o) in 3D is rotated around the origin to the resulting position (X_r, Y_r, Z_r).

Pre-calculated tables with the sine and cosine values can be used meaning that for every trigonometric value, we will only need one table lookup. It would also be possible to calculate the 6 values before entering the loop providing a comparable speedup, but the lookup tables needed are small making them easy to justify for eliminating all unnecessary calculations. The pre-calculated values are stored in two arrays, one containing sine values and the other containing cosine values.

3.2. Particle representation

The easiest way of representing a particle made of voxels is to make a solid object. As mentioned in [1], this is very inefficient, since the particles are only moved the distance of 1 voxel for every timestep. Therefore it is not possible for a particle to venture into another although they are hollow. Using this fact the algorithm in [1] moves only the outer shell of the particles eliminating all voxels in the interior, resulting in a drastic decrease in time spent in collision detection. We present here a method that we will call "shell area reduction" which will take this method a bit further. We begin by simplifying the representation of a solid object by a hollow one, retaining only a thin solid shell defining the borders of the particle. It is then easy to construct an algorithm that produces a chess-board like configuration on a flat voxel surface using modulus operations. This way half of the voxels can be defined as empty and half as real voxels. Similarly we can treat arbitrarily shaped surfaces by counting the number of neighbouring voxels at every position of the shell. If the number is above a predefined threshold value, the voxel is removed. This way we will achieve similar hole structures on arbitrary surfaces. By adjusting the threshold value we can easily achieve a result where the number of voxels on a random surface is reduced to roughly one half, in other words we get about the same amount of reduction as in the chessboard case. Tests show that this idea is feasible and no problems have been encountered for objects with reasonable resolution, and especially not for roughly spherical objects. While trying out different packing simulations using different shapes of particles, the result was examined to see if particles had penetrated into larger particles. The only situation where we were able to produce such penetrations was when thin cylindrical particles were packed and some of the particles had a diameter very close to 1 voxel. This allowed the smallest particles to penetrate through the shell of larger particles. Though it is possible to create such a problem, it is very unlikely to occur if we use a little bit of caution when defining our simulation. If the size of the holes created using the shell area reduction is around 1–2 voxels in size, any particle larger than this will not cause problems.

When packing using this shell area reduction method there is always a risk for a slight overlap from rotations where some voxels that were not there while packing, will be written into the set when finalising it using the exact rotation method. These voxels might collide with the voxels of another particle. To check the amount of overlap, one set of simulations with spherical particles with a diameter of 1 unit length was simulated and one with hexagonal plates with a diameter of 1 unit length and a thickness of 0.4 unit length. Each set contained 7 simulations with the same particle definitions but varying resolutions from 20 to 80 voxels/unit length. These sets where simulated both with the shell area optimisation enabled and disabled. The results are shown in Fig. 5. The rotation method allowing holes to form in the data results in an overlap of less than 0.1% on all resolutions. For spheres it is the rotation technique that produces almost all of the overlap, while the importance of the shell area reduction is slightly elevated when packing plate-like particles. The reason for this is the flat structure of the plate-like particles. When two particles align their flat surfaces against each other the possibility for overlapping is increased when using a particle shell with lots of holes in it. Furthermore it was seen that the number of voxels overlapping was almost constant on all resolutions, the reason being that there are roughly the same number of contact

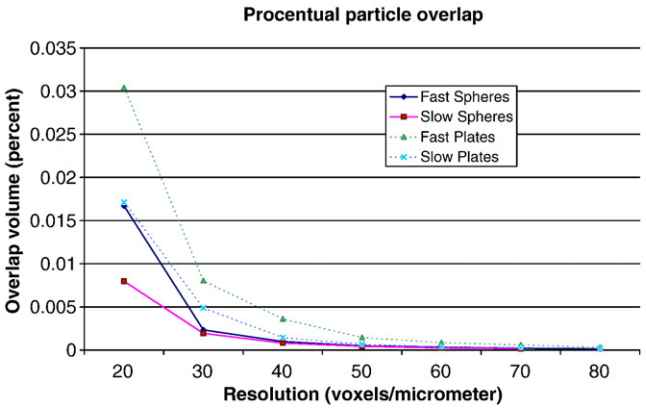

Fig. 5. Packings with resolutions from 20 to 80 voxels/unit length using spheres and plates. The degree of overlap between particles was analysed to see how the particle description optimisations affect the end result. The "Fast" keyword in the graph indicates that shell area reduction was used.

points between particles on all resolutions with the same packing parameters. It is only at the contact point that overlaps can be found.

The shell area reduction method is added as an optional feature to the algorithm. It can be turned off to ensure more exact results, but since the voxels needed for rotation are cut down to one half, the algorithm benefits considerable speed improvements.

When the number of voxels in a particle has been reduced, the coordinates are entered into an array, one array for every unique shape of particle present. The coordinates in the array are normalised so that they are centred around the origin and interleaved in the (x,y,z) form. The pointer to this array is added to the metadata of a particle. This metadata enables us to quickly scan through the coordinates one by one, rotating them internally, and then by the use of the particle's position in the packing matrix, check the positions for collision. This method is used instead of representing the particle in a 3D array, meaning that we need to scan through the whole array rotating the points that define a particle voxel. Moreover, for every active particle a rotation array is provided where the calculated rotations for every voxel are stored. The values for this array are stored when a new particle position is accepted and written into the packing matrix. Then the next time the particle is moved, this data is used to remove the particle eliminating the need for calculating the rotations. The use of this extra array would be a memory problem for simulations with very large numbers of particles, but as we will show later particles are freezed using a predefined rule leaving only a small part of the particles active. Another reason for using this representation is that the set of rotated coordinates can be sent directly to the packing matrix which internally checks all coordinates and sends the result back. This avoids lots of function calls while keeping the algorithm conceptually clear.

3.3. Modulus operations

When particles are packed into a simulation box, the boundaries of the box will introduce a disturbance in the packing structure. Generally it has been reported that the ratio between the size of the largest particle and the side length of the box has to be at least 1:10 [11]. Particles close to the edges are not able to move as freely as those close to the centre of the box where particles are dynamically interacting with each other. A solution is to introduce periodic boundary conditions in the x and y directions allowing the particles to move through the boundaries. Periodic boundary conditions are implemented by calculating the modulus of the x and y coordinate with respect to the side length of the simulation box. This solution introduces some performance issues that need to be addressed.

The most straightforward way is to execute the modulus operation on every coordinate when reading from or writing to the packing matrix. Though this works perfectly well, it is not efficient because most of the particles are far away from the border and need no modulus operation. The solution to this problem is to use the bounding box for the particle. When a particle is moved we start by normalising its reference corner (x,y) coordinates. The reference corner is the (x_o, y_o, z_o) corner closest to the packing matrix origin. The normalisation process modifies negative x and y coordinates by adding the side length of the box. This process ensures that the reference corner is inside the packing matrix. After this the collision detection can start, one operation using modulus and one without modulus being provided. The modulus operation is only used for those particles with a reference corner extending beyond the upper boundary of the x or y packing matrix. This way the number of modulus operations needed is minimised, showing around 10% speed increase on our standard set and 20% speedup when doing a similar set with polydisperse spheres of comparable simulation size. The reason behind this is that in the later case the number of spheres not requiring modulus operations is dramatically increased. In our standard test set with spheres which have diameters 1/10th of the simulation box side, this means that they will use a considerable part of the simulation box x and y lengths when including the bounding boxes as illustrated in Fig. 6. The amount of particles in the box is close to one thousand. Only a small portion of the particles outside the modulus region will benefit from the optimisation. This explains why the polydisperse set where a large part of the particle sizes is much less than 1/10th of the packing box size benefits from the optimisation to a larger extent.

The modulus operation is quite slow, being comparable to a division operation and it would be desirable to avoid it altogether. As reference coordinates are already normalised, we can actually be sure that they are always positive, and it is now easy to wrap them only if needed as seen in Fig. 7. Only one subtraction and an if-statement will be executed for most of the operations and since this check is executed on voxel level there will only be a negligible part of the total particle voxels that need wrapping. The result of this optimisation on our reference packing is roughly a doubled packing speed. Since the operation is very fast, no effect of the border optimisation was seen and memory access operations are now dominating the collision detection process.

3.4. Collision detection

Our tests indicate that a collision actually occurs in more than 50% of the collision tests. This implies that it would be good idea to detect overlapping voxels as quickly as possible, exiting the collision detection procedure at an early stage. As mentioned in Section 3.2 we save the rotated coordinates when accepting a new move. The reason for not doing this while executing the actual collision detection is that we would have to revert this action if a collision was found. The

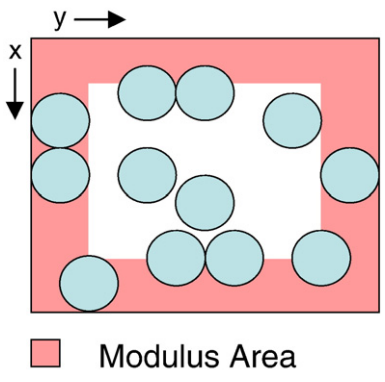

Fig. 6. Cross section of the packing matrix illustrating the size of the modulus zone.

```
// subtract box side length from the coordinate x
result = x - boxSize
//if the result is negative : return the unchanged coordinate
if (result < 0) result = x
```

Fig. 7. As coordinates might extend beyond the upper bound of the simulation box the box length is subtracted from the coordinate. The coordinate will be negative if it is within bounds, meaning that we should use the original coordinate. On the other hand, if the result is positive, the coordinate must be outside the box and we can use the new result value.

array containing the rotated coordinates is used for removing the particle the next time we try to move it and as such it has to reflect the current state of the particle. This means that we can exit the collision detection as soon as the first voxel overlapping with another particle is found.

It would also be beneficial to optimise the collision detection in such a way that we can locate a collision quicker. One might think that a collision is more likely to occur on the bottom side of a particle (the side turned downwards when starting the simulation), since particles are generally moving downwards and settling on a layer of already packed particles. This proves to be a false assumption, since the particles are rotated up to 360°. Theoretically the bottom of a particle can be anywhere, and we have not been able to create a way to easily keep track of which part of the particle is presently facing downwards. An easier approach is to spread the collision detection among the octants of the particle randomly meaning that surface locations spread around the particles are tried for collision at an early stage. Especially when we are simulating plate-like objects which are allowed to rotate many degrees in one step, it is possible that a reasonably large part of the particle might collide with a neighbouring particle. Spreading the collision detection will be particularly effective if the number of colliding voxels in an octant is significant as this will mean that the collision could be detected at an early stage. The sequence of voxels to be tested for collision can be done on a pre-calculated set of voxel positions containing the coordinates of the voxels to be tested in a particle, so that it will not affect the packing speed. Although the array is accessed sequentially, the corresponding positions will be random. If a collision is about to happen in one of the octants, we are now likely to find it earlier compared to if we are scanning one quadrant at a time. In practise it was very difficult to see any clear advantage. Our pre-calculated set was packed with random positions and ordered positions using different degrees of maximal rotations of the particles. As can be seen in Fig. 8 there was a slight tendency for increased speed but it was not clear enough for us to keep the optimisation. For further research an algorithm which uses the rotation angle to try to optimise the collision could be tried. This method would need to keep track of

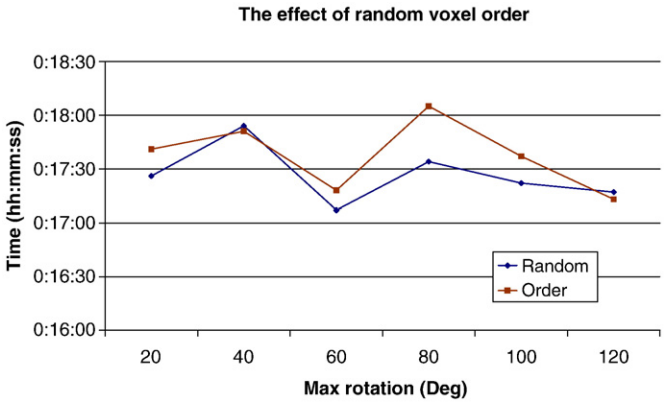

Fig. 8. The effect of random surface voxel order in collision detection on packing speed as a function of maximum rotation allowed from the initial particle rotation.

where the last collision was detected implying that there is a higher possibility for collision in that general direction. Although it might mean more calculations it could prove worth while, since it could reduce the number of memory accesses needed.

3.5. Particle freezing

As was mentioned in [1], particles which will not be moved anymore can be freezed, so that we do not need to consider them anymore. This is crucial both considering memory consumption and speed. If we store the metadata for rotations for hundreds of thousands of particles it will add to the already problematic memory consumption of the packing matrix, and moving hundreds of thousands of particles simultaneously is simply not feasible. The solution is to freeze particles that are deemed inactive. We have implemented this by keeping a particle coordinate history and a particle freezing interval t_f specified in time steps. For every t_f time steps, the particle position history is updated for all active particles. If they have not moved since the last time, they will be freezed. By increasing the freezing interval, particles are allowed to move for a longer time before freezing. Typically this limits the number of active particles to between 100 and 2000 particles which is feasible to handle. It is important though to note that the number of active particles will also be a function of the mean particle size in relation to the packing box surface area; moving particles will sooner or later be blocked by new particles introduced into the packing matrix.

3.6. Rising roof

In all simulations, analytical or voxel based, new particles are introduced at the top and subsequently translated and rotated towards the bottom of the particle simulation box. It is actually not necessary to always start at the top since the *x,y* position and the direction of the movement of the particle are random. Instead we start by introducing a new particle close to the top level of the particles already present in the simulation box. We call this the *z*-position of the roof. When a collision is detected while inserting a new particle, the roof is slightly elevated, to help inserting new particles. This way the insertion point of new particles is slowly risen until it reaches the maximum height of the simulation box, hence the name rising roof. This way the total number of moves needed for a particle to reach other particles at the bottom can be greatly reduced.

3.7. Memory considerations

In the original paper [1] it was stated that separate identification numbers need to be used for every particle to achieve high speeds of simulation. Although this assumption has good grounds, there are more factors that need to be considered. The main reason for using separate identification numbers for every particle is that we do not need to remove them before doing the collision detection. The only thing we have to do is to check if the new position results in a collision with a voxel with another identification number than the currently processed particle. This means that if there is a collision we can just go straight to the next particle. If, on the other hand, we use 1 bit per voxel in the packing matrix to identify if the voxel is taken or not, we cannot separate between different particles. This raises the need for removing a particle before doing the collision detection since we do not consider the collision between the old and the new position of the same particle. Technically it is easy to remove the particle, since we store all metadata for the particle, but it results in extra work that needs to be done if there is a collision. If there is a collision we have to write the particle back to its original position, meaning that the removal of the particle was useless in the end.

However there is a great need to reduce the memory consumption for the packing matrix. When making polydisperse particle packings

where sizes between particles can vary from 10 to 50 times the smallest particle size, the resolution needed is very high, a worst case scenario for typical paper coating applications seeing up to 5000^3 voxels in order to make sure that the smallest particles are detailed enough. This would result in a 466 GB packing matrix if we use 32 bit values for every voxel position. This however can be reduced to less than 15 GB by using a one bit representation, which is possible to handle on modern desktops.

Another way of dealing with the memory problem would be to use the same 32 bit identification number and introduce an effective real time compression algorithm on that map. For instance an octree representation typically provides a good compression ratio on the 3D data. The problem with this approach is that the data access is very dynamic with low spatial locality. A very large part of the data set will be accessed in cycles over and over again for every time step taken, since every active particle is moved one step. An effective bit packing algorithm on the other hand would make access and modification very fast, since bitwise operations can be used.

The scheme that we have used consists of a 2D map of y and z coordinates, referencing to arrays of bit-packed x-rows. This means that we can find the correct x row that we want to access very rapidly. Fetch and bit set methods can be seen in Fig. 9. A reference map called data is created, it contains the x rows for every y and z combination and can be used to fetch the right row rapidly. After this, the correct word containing the voxel is found by dividing the x position by 32 since 32 voxels fit into a single 32 bit word. Since 32 is 2^6, it is easy to execute the division buy shifting the integer value for x by 5 steps to the left using the bitwise shift operation "≫". After fetching the right 32 bit word we need to either modify or read the exact bit corresponding to the voxel we wish to study. To achieve this goal we need to create a word with only 1 bit set, the bit we want to read or modify. This is accomplished by shifting the integer value 1 until it reaches the right position in a word using the "≪" operation. The number of shifts needed can also be found using the bitwise and operation "&", by executing "x & 31". The integer value 31 is represented by the word where the rightmost 6 bits are set, which means that combining it with the value of x will leave us with only the 6 lowest bits of the value x, or the reminder of the division with 32. This is the correct position of x within its 32 bit word and can be used to shift the 1 to its correct position. This results in a 32 bit word in which the value for our voxel exists and a 32 bit word where only the bit in the same position as our voxel is set. This can now be used to either modify or read the data. Fetching is easily done using the logical AND operation "&" leaving us with a value greater than 0 if the voxel is set and 0 if the voxel is not set. If we want to set the bit at that position, the bitwise OR operation "|" is used. The result is a word were all the bits in the original word is set plus the bit at the position that we want to modify.

The results in Section 4 show that the bit packed data set actually outperforms the original voxel representation. The reason for this is the memory access frequency of the algorithm. Deepest in the algorithm is the memory access that modifies 1 voxel at a time. When profiling the algorithm using 32 bit identification numbers it was seen that around 70% of the time was spent on memory operations. This problem is to be expected in a memory intense application like particle packing. Memory access is quite often the bottleneck in modern computer applications [12]. Memory consumption is reduced by a factor 32 using the bit packing and spatial locality is increased. Fetching a single word to extract a bit might well mean that the same word will be used again soon for checking another voxel in the same particle.

```
Data Fetch
(data[y] [z]) [x >> 5] & ( 1 << (x & 31)) ;

Set Bit
tempValue = &((data[y] [z]) [x >> 5]) ;
*tempValue = (*tempValue | (1 << (x & 31))) ;
```

Fig. 9. Bitwise operations for fetching and setting a given voxel.

4. Results

In Fig. 10 the results of the different optimisations are represented. As can be seen, the sine and cosine optimisations are the most effective optimisations, though improved rounding operations further increase the speed by roughly the double. When adding the reuse of saved rotation coordinates, improved modulus operations and rising roof, we see moderate improvements but the combined result is significant. The last optimisation, being the shell area reduction gives a final triple increase in speed, though it is worth noting that a little bit of caution has to be exercised when choosing particle shapes and resolution to ensure that no particle penetration will occur. The random spreading of collision detection on the different parts of the surface of the particles did not, on the other hand, provide any improved speed.

In our simulations a 64 bit AMD machine with 4 GB of primary memory was used. It is important though to keep in mind that we set out without knowing the exact implementation details used by the authors of the original article [1]. Nevertheless the same types of optimisations were used for the base case A. What this graph does not show is how our fast rotation fares compared to the shearing rotation implemented in [1]. In Fig. 11 we show how bit packing compares to the use of a 32 bit array. Bit packing actually outperforms the unpacked set, and the effect proves stronger as the resolution increases. Probably the most important reason for this is that spatial locality increases using bit packing; when fetching a single voxel the cache line will be filled with a large number of neighbouring voxels, compared to the 32-bit case.

It is worth noting that our optimisations did not change the complexity of the algorithm. The complexity for updating the simulation one timestep is $O(n)$, where n is the amount of voxels in the mean number of particles that are active during the simulation. The optimisations done focused on reducing the number of voxels and the number of timesteps. Furthermore, memory optimisations were applied since it was seen that memory access operations dominate the packing process. Since the number of voxel access operations during a single timestep is significant, further optimisations were successfully done by focusing on individual time consuming operations done during voxel manipulation, such as floating point rounding, modulus operation and trigonometric functions.

To demonstrate how the algorithm scales we present the graph in Fig. 12. Generally, we expect the algorithm to scale in proportion to

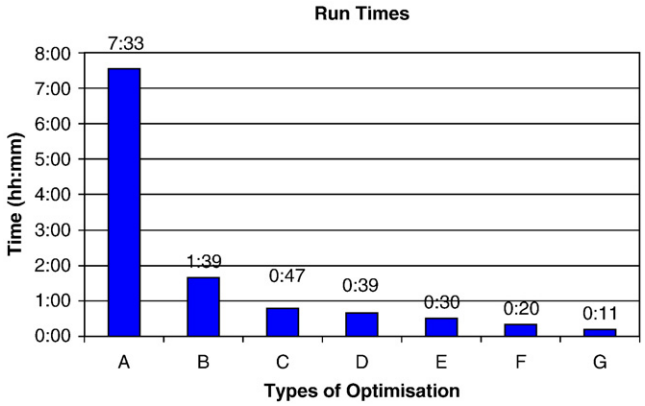

Fig. 10. Time consumed on a 400^3 packing matrix with our mono-disperse test sphere packing (A) Basic optimizations (B) Sine/Cosine (C) Rounding operation (D) Saved rotation coordinates (E) Rising roof (F) Improved modulus (G) Shell area reduction.

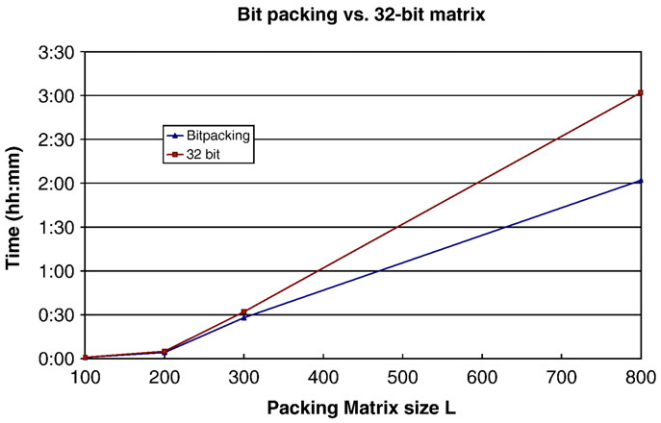

Fig. 11. The effect of bitpacking on speed for different resolutions.

the number of voxels in the packing matrix, but not to depend much on the shape of the discretized particles. Two cubic simulation matrices with a unit size side length of 10 mm and resolutions varying from 20 to 160 voxels/mm were simulated. One matrix consisted of a monodisperse packing of spheres with a diameter of 1 mm and the other was a polydisperse packing containing 50% per volume of spheres with the diameter 1 mm and 50% per volume of spheres with the diameter 0.4 mm. This results in a packing where the area of the spheres is roughly doubled in the polydisperse case. Hence we would also expect that the simulation time would be roughly doubled, since only the voxels on the surface of the particles are used for collision detection. The graph shows the running time of the polydisperse packing for 20, 40, 80 and 160 voxels/mm resolution is 4.4, 3.1, 2.7 and 2.2 times slower respectively. Hence, when the resolution is very small, other parts of the algorithm like updating particle metadata affect the run time to a greater extent. Higher resolutions tend to even out the difference slowly approaching the area difference ratio between the packings. If we, for example, would be using an analytical worst case approach with n^2 collision detections per particle move, the result would be drastically different, since there are around eight times more particles in the polydisperse case.

As an example of complex arbitrarily shaped particles we have simulated the packing of cow-shaped particles in Fig. 13.

5. Conclusions

Simple optimisations targeting deeply nested methods were successfully applied to a particle packing simulation algorithm, providing an algorithm which is roughly 40 times faster than the one

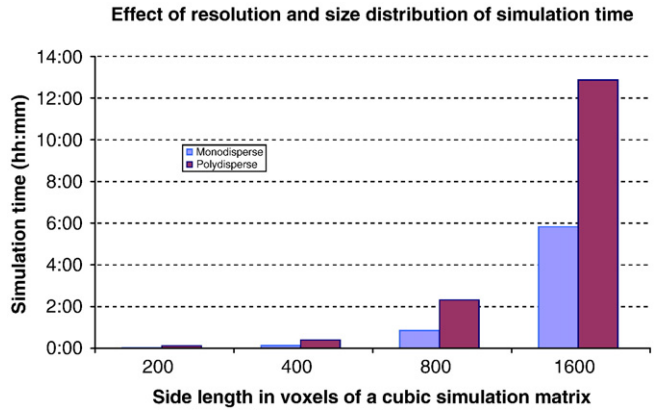

Fig. 12. Comparison of simulation time of a monodisperse and polydisperse cubic simulation box with a side length of 10 microns and different resolutions.

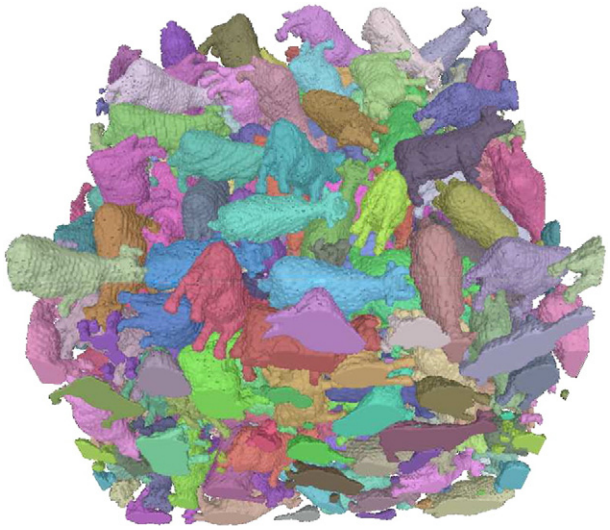

Fig. 13. An example of complex particle shape, "cow" in a packing.

that we set out with. Judiciously chosen methods allowed us to both decrease memory consumption and increase simulation speed without sacrificing the quality of the result. The improved algorithm is able to perform large simulations of more than 4000^3 voxels containing hundreds of thousands of particles on a modern desktop computer.

Acknowledgements

Financial support from the Finnish National Agency for Technology and Innovation — TEKES (grants 40514/03 and 40466/05) and the C-Coat consortium are gratefully acknowledged.

References

[1] X. Jia, R.A. Williams, A packing algorithm for particles of arbitrary shapes, Powder Technology 120 (2001) 175–186.
[2] K.R. Wadleigh, I.L. Crawford, Software Optimization for High Performance Computing, Prentice Hall PTR, 2000.
[3] E. Haines Akeinine-Möller, Real-time Rendering, Second edition, A K Peters, Ltd, 2002, pp. 631–667.
[4] A. Donev, S. Torquato, F.H. Stillinger, Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles. I. Algorithmic details, Journal of Computational Physics 202 (2005) 737–764.
[5] A. Watt, 3D Computer Graphics, Third edition, Addison-Wesley, 2000, pp. 51–52.
[6] D. Vidal, X. Zou, T. Uesaka, Modelling coating structure development using a Monte-Carlo deposition method, Tappi Journal 2 (2003) 16–20.
[7] P. Shirley, Fundamentals of Computer Graphics, A K Peters Ltd, 2002.
[8] B. Chen, A. Kaufman, 3D volume rotation using shear transformations, Graphical Models 62 (4) (2000) 308–322.
[9] S.-I. Chien, Y.-M. Baek, A fast black run rotation algorithm for binary images, Pattern Recognition Letters 19 (5–6) (1998) 455–459.
[10] H. Brönnimann, G. Melquiond, S. Pion, The design of the boost interval arithmetic library, Theoretical Computer Science Archive 351 (1) (2006) 111–118.
[11] D.J. Cumberland, R.J. Crawford, The packing of particle, Handbook of Powder Technology, Elsevier, 1987, pp. 96–97.
[12] D. Burger, J.R. Goodman, A. Kägi, Memory bandwidth limitations of future microprocessors, Proceedings of the 23rd annual international symposium on Computer architecture, 1996, pp. 78–89.