Hardware Accelerated Point-based Rendering of Granular Matter for Interactive Applications

David Jacobsson

March 15, 2007
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Anders Backman
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Keywords

point-based rendering, computer graphics, GPU programming, real-time rendering, granular matter, isosurface extraction, point-based refinement, graphics hardware, point-based graphics, procedural shading, surface splatting, texture splatting

Abstract

In this master's thesis, we propose a new method for shading surfaces of mesh-free, particle-based, real-time simulations. The shading algorithm includes high-resolution texture details that flow with the granular matter simulation. Detail texturing techniques, together with a hardware accelerated surface splatting algorithm, forms the basis for the method.

The thesis also provides an overview of methods that can be used for a point-based rendering pipeline, from isosurface extraction to surface shading. The methods are compared with respect to ease of implementation, performance and shading quality.

With the introduced tools, visualization of granular matter such as sand, mud, salt or snow can be performed in real-time, on the latest graphics hardware.

For simulation, a Smoothed Particle Hydrodynamics method tailored for granular matter, is used.

Contents

Chapter 1

Introduction

Real-time simulations in interactive applications has become very common during the last couple of years. The main driving force in this development is the ever increasing demand on more realistic environments for games and simulations.

Techniques for physical simulations have been well known for a long time. Most of these techniques have been developed for simulation of engineering problems and not for interactive applications. The limited calculation resources that are available in real-time applications put new demands on algorithms. Trade-offs between physical accuracy and calculation speed is at many points needed.

Rigid body simulations have become a standard feature of many interactive applications. Simulations of this kind can frequently be seen in games where simulations of for example falling boxes can enhance the realism of the game environment.

During the last couple of years, another type of real-time physical simulation have been brought to daylight, deformable matter. One example is simulation of fluids such as water and blood. Granular matter is a special collection of deformable materials that includes sand, mud and soil. These materials are complex, in that they sometime behaves as a rigid material, for example when a vehicle drives up onto them, and sometimes as fluids. These materials are also very deformable and the topology of the materials can change dramatically in a short time period.

To simulate deformable materials one can use Finite Element Methods (FEM) which use a mesh to describe the geometry of the material. Another method that have become popular is Smoothed Particle Hydrodynamics (SPH) [MCG03], which is a meshless method. The basic building part in these simulations is particles without explicit connectivity information.

Visualization

Visualization of 3D models in computer graphics is normally done with polygon-based methods. Here, the basic building block are faces, which are formed from vertices together with connectivity information. Graphics hardware is also highly optimized for this kind of rendering.

Since SPH is a meshless method, the connectivity information that is needed for polygon rendering is not directly available. This makes it harder to render these simulations.

One way to solve this problem is to use some kind of mesh construction algorithm such as Marching Cubes [LC87]. This results in a polygon mesh that could be rendered

with a normal polygon based graphics pipeline, the only problem here is the overhead of the mesh construction. This makes other rendering methods interesting.

Point rendering

A more straightforward approach is to use point rendering methods for the visualization. This means that no mesh construction have to be made and polygons never will be introduced into the graphics pipeline.

Point rendering algorithms comes in several different flavors. Direct point rendering methods that use aliased points for rendering are among the simplest. On the other side, rendering methods such as surface splatting uses mathematically complex reconstruction filters that makes high quality texturing possible. These algorithms operates on points on the surface of the simulated matter to produce an occluding surface in screen space.

Shading

The last problem is to shade the occluding surface produced in the point rendering algorithm. Shading the surface can be separated into two problems.

First, the shading method should increase the resolution of the simulation. A typical frame to be rendered may contain ten thousand (10000) particles but these particles normally covers an area of far more than ten thousand (10000) pixels. This means that the shading algorithm should add information to get a high resolution result.

Second, the shading should have some kind of temporal and spatial coherence. When the granular matter deforms and moves, details in the shading have to follow the simulation.

How to solve these two problems, with shading techniques that can be made to work together with the point rendering methods, are unknown and also the main motivation for this thesis work.

1.1 Origins of point-based graphics

Using point-based methods to solve a visualization problem may seem like the wrong way to go since polygonal methods is well known and understood. Todays graphics hardware is also designed and highly optimized for polygons.

The main historical reason for the development of point-rendering was the introduction of laser scanners that makes it possible to produce point sampled meshes with very high density from real world objects. If these meshes are converted to polygons and then rendered, the high density will make the polygons so small that they cover less than one pixel in the final rendering. This makes it unsuitable to create polygons from the point samples since it does not add any information to the final rendering.

The origins of point-rendering dates back to the original point rendering article by Marc Levoy and Turner Whitted [Lev88]. For a more recent overview of point-based techniques for computer graphics, [KB04] is a good starting point. For an introduction to point-based rendering techniques, see [Kri03]. An summary of the most common point-rendering technique, Surface Splatting, is done in [Räs02].

1.1.1 Why points?

When it comes to granular matter simulations, or any particle-based simulation, the simulation itself is point-based which makes point-based methods directly applicable. But the low resolution of todays real-time simulations does not suit all point-based rendering methods directly. Still, point-based methods is attractive since the overhead from converting points to polygons is not present. In the long run when the resolution of the simulations can be increased, point-based methods will be even more attractive.

1.2 Previous work

At VRlab, Umeå University, one of the research areas is real-time simulation of granular materials. The main method used for simulation at VRlab is SPH, and several master thesis projects have been written on the subject.

Among others is the predecessor to this master's thesis [Ves04], which is basically an implementation of [MCG03] together with a new surface extraction and refinement algorithm. This work focus on the simulation of fluids, such as water and blood, and does not deal with the visualization problems that are specific for granular materials.

Another master's thesis [Nor06], performed for Oryx Simulations, basically deals with the same topic as our work, rendering of granular materials. Their work focus on a Marching Cubes type of implementation into an existing simulation platform.

1.3 Objective

> *"The aim of this thesis project is to investigate methods for how to, in real-time, render granular material such as sand, mud or soil, based on purely point-based methods."*

The objective of the thesis states a problem, rendering granular matter in real-time. This problem is no detailed specification on what to do, which means that the thesis work have been explorative since no real idea for how to solve the problem is known. The aim is to present a summary of technologies and methods that can be used to solve the problem, and also to give a more detailed view of what the problem actually is.

1.4 Prerequisites

The intended audience of this report is mainly people in the graphics community interested in point-based rendering techniques for interactive simulation applications. Basic computer graphics knowledge is therefore assumed. Knowledge of university level linear algebra and calculus is helpful when reading the mathematical parts of the thesis.

1.5 Thesis outline

Chapter 1, Introduction
Background of the thesis and introduction of topics covered in the thesis.

Chapter 2, Problem description
Description of the general problem: real-time point-based rendering of a granular matter simulation, and the different steps needed to do this.

Chapter 3, Point-based methods
Overview of the methods explored to solve the different parts of the problem described in chapter 2.

Chapter 4, Our contribution: Point-based texture splatting
Description of our newly developed method, a point-based shading system for granular matter simulations.

Chapter 5, Implementation notes
How the methods in chapter 3 can be put together to a complete system and what tools you need to implement the methods on modern graphics hardware.

Chapter 6, Results and discussion
How our solution looks and what performance that can be expected from the different methods.

Chapter 7, Summary and conclusions
What have been achieved and what research that can be performed in future developments.

Chapter 2

Problem description

To get an interactive real-time simulation of granular matter, several different problems need to be solved. These problems can be divided into two parts. First, a simulation that describes how the granular matter moves and behaves under different circumstances and second, a visualization method that defines the apperance of the material.

The method used for the simulation in our work is based on a previous master's thesis [Ves04], which is basically an implementation of the method used in [MCG03] called *Smoothed Particle Hydrodynamics, SPH*. Since the visualization is the main focus of this thesis no further research have gone into SPH and other simulation methods.

The visualization problem can be divided into four parts (figure 2.1). First, a surface is extracted from the simulation. Second, the surface is refined to give a smooth and dense representation. Third, the refined point cloud surface is rendered with a point rendering scheme. Finally, the surface is shaded in a way which gives the user a granular matter "look-and-feel" of the material. In all four steps the main restriction is the real-time requirement of interactive graphics.

The full system should work in real-time on modern graphics hardware. Extra care have to be put into solving most of the visualization problem on the GPU since the granular matter simulation is driven by the CPU. This will influence the choices made throughout the thesis.

First, a brief overview of the four subproblems will be given and after that each problem is analyzed one at a time in chapter three, *Point-based methods*.

2.1 Surface extraction

The first step in visualizing the granular matter is to decide which particles in the simulation that should contribute to the rendering. The main visible feature of a granular material is the surface. Therefore it makes sense to decide which particles in the simulation that lies on the surface and only let those particles contribute to the visualization of the simulation.

2.2 Surface refinement

Depending on the extraction algorithm, the surface produced can be noisy and contain discontinuities. The surface refinement step ensures that the surface is smooth and dense

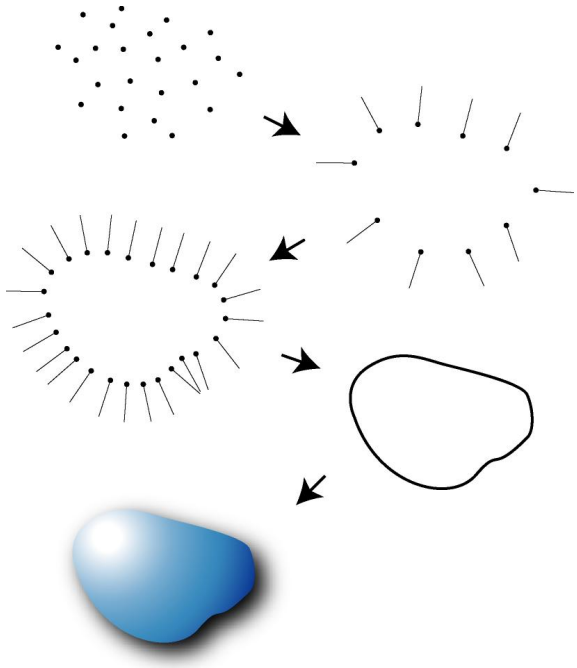

Figure 2.1: The four parts of the problem. Each arrow represents one of the four steps.

enough to be rendered, i.e. the step increase the resolution of the surface. This step can be skipped if the surface extraction algorithm produces a good enough surface to be used for the rendering. One issue here, is that the geometry of the simulation may need to be upsampled to give a visually pleasing result. Furthermore, one should consider if particles should belong to one single surface, or be part of different surfaces.

2.3 Surface rendering

Since a meshless simulation method is used, the result of the simulation can not be visualized directly with polygons that are normally used in computer graphics. The result of the surface extraction and refinement is a point-cloud with normals describing the orientation of each point on the surface. Therefore, algorithms that can render point-clouds directly, without explicit connectivity information, are examined.

2.4 Surface shading

The granular matter simulation is quite sparse and therefore the surface has few simulation points that describe the look of the simulated area. Therefore the shading algorithm should increase the detail of the surface in a way that ensures temporal coherence and results in a high resolution view of the surface from all angles. In short, the problem is to increase the detail to give something that looks like more simulated particles than it is.

Chapter 3

Point-based methods

In this chapter, several different methods that could be part of a point-based rendering pipeline will be evaluated. Those suited for granular matter visualization will be given extra attention. The methods will be presented in accordance to the four different parts presented in the previous chapter, *Problem description*. Methods that run in real-time on modern graphics hardware will be in focus.

3.1 Methods for surface extraction

There exists several methods for extracting surface particles from a particle simulation. The basic problem consists of deciding which of the particles that belongs to the surface of the particle cloud (fig 3.1, left and middle picture). Furthermore the normals that describe the orientation of these particles (fig 3.1, right picture) has to be extracted. In this section a few methods for surface extraction will be explained and examined more carefully.

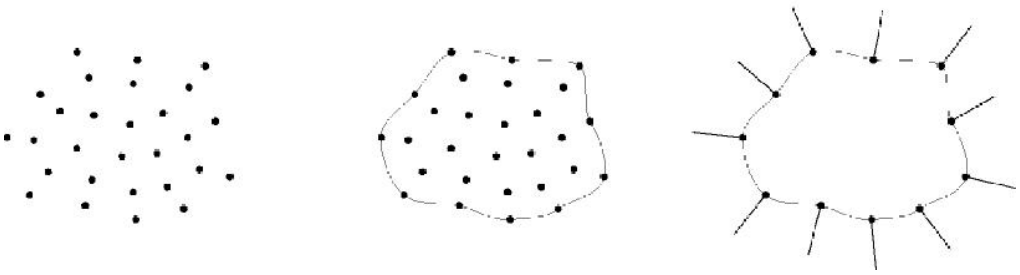

Figure 3.1: Surface extraction

3.1.1 Implicit surfaces

Extraction of a surface from point clouds is normally performed with help of implicit surfaces. An implicit surface is defined as an isosurface (contour) of a function, i.e. a surface that satisfy the equation

$$f(\mathbf{p}) = f(x, y, z) = constant \tag{3.1}$$

for some point $\mathbf{p} = (x, y, z)$ in 3D space. The function $f(\mathbf{p})$ describes a continuous scalar field throughout 3D space. In 2D space the corresponding contour of a function $g(x, y) = constant$ would describe a level curve to a 2D surface (such as height curves on a map). The 2D function $g(x, y)$ describes a continuous height field.

If all particles i in the simulation has a value a_i, the function $f(\mathbf{p})$ can be defined as a weighted mean value depending on the distance from $\mathbf{p}$ to $\mathbf{p}_i$,

$$f(\mathbf{p}) = \sum_i \frac{1}{|\mathbf{p} - \mathbf{p}_i|} a_i. \tag{3.2}$$

This function, that specifies how a particle neighborhood influence the scalar field, is normally denoted *"Smoothing kernel"*.

The problem then boils down to choosing at which points $\mathbf{p}$ the function should be evaluated and for which constant the isosurface should be extracted.

3.1.2 The SPH color field

To extract a surface from a SPH simulation, a feature of the SPH simulation called the color field is used. The color field, c_S, is a scalar field defined as being 1 at all particle locations and 0 everywhere else. Together with a smoothing kernel,

$$W(\mathbf{p}, h) \tag{3.3}$$

the color field can be defined,

$$c_S(\mathbf{p}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{p} - \mathbf{p}_j, h) \tag{3.4}$$

where m_j is the mass, ρ_j is the pressure and $\mathbf{p} - \mathbf{p_j}$ is the distance vector for particle j. Together with h, which is the size of the smoothing kernel, this gives a continuous scalar field throughout space. The color field can then be used to define an implicit surface tracking the simulated particles. The gradient of the smoothed color field, $\mathbf{n} = \nabla c_S$, in turn gives the normals of the implicit surface [MCG03].

To define a surface from the color field the gradient is used to identify which of the simulated particles that belongs to the surface by the expression

$$|\mathbf{n}(\mathbf{p}_i)| > l \tag{3.5}$$

where $\mathbf{n}(\mathbf{p_i})$ is the the value of the gradient of the color field at particle position $\mathbf{p_i}$ and l is a threshold value. The normalized surface normal at particle $\mathbf{p_i}$ is then defined as

$$\frac{-\mathbf{n}(\mathbf{p}_i)}{|\mathbf{n}(\mathbf{p}_i)|} \tag{3.6}$$

The SPH color field surface extraction method is view independent, therefore all surface particles will be extracted, even though they wont be part of the final rendering.

See [MCG03] for further notes on color fields and smoothing kernels.

3.1.3 Iso-splatting

In [CHJ03], a point-based surface extraction algorithm, that can be used for point-based real-time rendering of isosurfaces, is presented. This algorithm generates point samples near the isosurface that in a later step are projected onto the real isosurface. The projected point samples can then be used in a point-based rendering algorithm.

The generation of point samples use a grid procedure that is similar to Marching Cubes [LC87]. A grid is constructed to cover the complete area for the isosurface. For each corner of each cell in the grid, the function defining the isosurface (equation 3.1) is evaluated. With these values one can identify if the isosurface crosses the cell. If the isosurface passes through the cell a point sample is generated in the center of that cell.

To get a better approximation of the isosurface the point samples generated is projected towards the real isosurface. An exact projection can be made but is too expensive to calculate to be feasible in a real-time implementation. An approximate projection $\mathbf{x}_0^{'}$, of a point sample $\mathbf{x_0}$ is performed with one iteration of the Newton-Raphson root-finding method

$$\mathbf{x}_0^{'} = \mathbf{x}_0 + \nabla f(\mathbf{x}_0)t \tag{3.7}$$

to get the approximate isosurface point at $\mathbf{x}_0^{'}$. The function $f(\mathbf{x}_0)$ can be defined as in equation 3.1 and t is calculated with help of the gradient ∇f of $f(\mathbf{x}_0)$

$$t = \frac{f_I - f(\mathbf{x}_0)}{\nabla f(\mathbf{x}_0) \times \nabla f(\mathbf{x}_0)} \tag{3.8}$$

for isovalue f_I.

One advantage of the Iso-splatting algorithm is that the projection step can be calculated on GPU with a simple floating point division and vector dot product. The gradient and point location is already sent to the graphics hardware for surface shading and the isovalue in equation 3.8 can be sent once per frame.

Iso-splatting outperforms a standard Marching Cubes algorithm in most cases and use less memory. One more advantage is that no conversion to a polygon based representation is needed. This means that all work follows the simple point-rendering paradigm.

For a complete derivation of the equations and details regarding the algorithm, see [CHJ03].

3.1.4 Other algorithms

There exists several other methods for isosurface extraction that are not oriented for SPH or other particle simulation methods. The most widely used algorithm, Marching Cubes [LC87], generates polygons and is therefore not considered in this work.

Another method, Marching Tetrahedrals, uses a technique similar to Marching Cubes, with the obvious difference that tetrahedrals instead of cubes are used for extraction. The thing that makes this method interesting, is the fact that it can be hardware accelerated with a GPU algorithm. This method divide the isosurface region in tetrahedrals. Each corner of the tetrahedral is sent to GPU as a vertex with isovalue for that position. With help of this information a quad or triangle that describes the isosurface can be generated for the tetrahedra [Pas04, Höf06]. Similar to Marching Cubes, this method generates polygons and therefore it does not fit directly into the point-based paradigm used in this thesis.

3.2 Methods for surface refinement

Some surface refinement methods only use points and normals for the refinement process. These methods is used not only for particle simulations but for real-time refinement of static meshes as well.

Other surface refinement methods are built around the methodology of isosurfaces, introduced in 3.1.1. The main use of these methods is scientific visualization of large data sets or as in our case, visualization of dynamic isosurfaces.

The main thing to achieve with the surface refinement step, is to increase the point-cloud density. Other things that refinement could handle is for example regularization of the point-cloud and filling of large holes in the geometry [GBP05]. Another refinement issue that is specific for granular matter is how to handle singular particles that them self are not enough to generate a surface.

3.2.1 Point-cloud refinement

The main source for real-time point-cloud refinement methods is the work [GBP05] by G. Guennebaud (and its predecessors [GBP04a, GBP04b]). This method is based purely on point-clouds and does not deal with the notion of isosurfaces.

Algorithm

The method consists of two major steps. First, points are inserted between the known points (sampling) and second, the new points are displaced to smooth the surface.

Each point in the point set, $\mathbf{p}$, have a position, a normal and a radius. The refined point set is iteratively defined. If the starting point set is P^0, the first refined point set is P^1. This leads to a sequence of point sets, $P^0, P^1, ..., P^n, ...$, where $P^n \subset P^{n+1}$, i.e. the refined point set P^{n+1} contains all points from the previous step P^n, plus the newly inserted points. To generate the new points to be inserted, local refinement for each point $\mathbf{p} \in P^n$ is performed.

Local refinement of a single point $\mathbf{p}$ proceeds in several steps (figure 3.2).

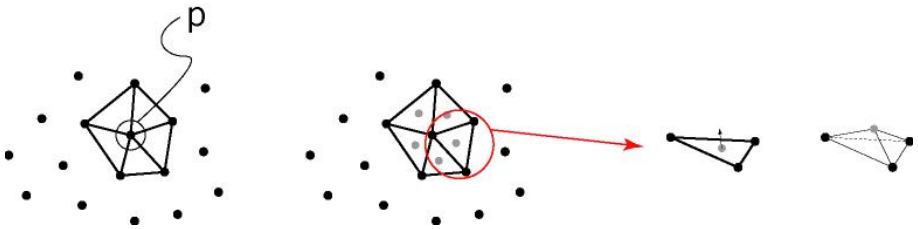

Figure 3.2: Local refinement of point $\mathbf{p}$. (left) Forming a neighborhood. (middle left) New points inserted at center of gravity. (middle right) Detail of one implicit triangle with new point inserted. (right) Displacement of the new point to produce a smooth surface.

First, a one-ring neighborhood, N_p, of $\mathbf{p}$ is computed. The point $\mathbf{p}$ together with the sorted neighborhood forms an implicit triangle fan on the surface (left figure).

Second, new points are inserted into the center of gravity of the implicit triangles, taking into account already refined points while also making sure that the sampling is regularized (middle left figure).

Finally, the newly inserted points, are displaced in order to produce a smooth surface (middle right and right figure).

All steps need to be handled with great care in order to produce a good result. Details on how to perform these steps can be found in [GBP05].

Comments

The algorithm can be enriched with a framework to handle sharp features and large holes in the surface. This, together with handling of uneven sampling of the input point-set, makes the algorithm versatile.

Real-time performance can be achieved with the algorithm, and somewhere around 400k-700k points per second can be generated. Crucial for the algorithm is a fast way to find neighbors in form of a spatial data structure for closest points query, for example kd-tree or octree. Furthermore, LOD selection of what parts of the point-cloud that needs refinement, is also needed.

The method is mainly CPU oriented which makes it somewhat unsuitable to use with a granular matter simulation since the the simulation itself need to be run on the CPU.

3.2.2 SPH isosurface refinement

In the predecessor to this master's thesis [Ves04] a surface extraction and refinement algorithm that uses the properties of isosurfaces and the SPH simulation was designed. This algorithm builds upon the point-cloud extraction described in 3.1.2.

Algorithm

The point set (point-cloud) P^0, serves as a starting point for the algorithm. The solution is iterative, and for each iteration, points in P^n is either accepted as part of the final point-set S, or refined (splitted) and added to P^{n+1}, which serves as input to the next iteration.

Each point $\mathbf{p} \in P^n$ is locally refined in each iteration. This procedure contains several steps.

First, the particle $\mathbf{p}$ is moved closer to the real isosurface with help of an approximation in form of one Newton-Raphson step

$$p_{\mathbf{n}+1} = \mathbf{p}_n - \frac{f(\mathbf{p}_n)\nabla f(\mathbf{p}_n)}{|\nabla f(\mathbf{p}_n)|^2} \tag{3.9}$$

where

$$f(\mathbf{p}) = c_S(\mathbf{p}) - k_{isocolor} \tag{3.10}$$

is defined from the gradient of the color field c_S (defined in section 3.1.2). The length of the step, is an approximation to the error in the placement of the original point. Therefore if the step is to long, the original point is considered faulty and is discarded altogether.

Second, the point **p** (if not discarded in the first step) is either added to the final point-set S if its placement is considered good enough, or split into four new points. These points are spread out in an even fashion in the plane defined by the original point and the normal at that point (figure 3.3). The four new points are added to P^{n+1} as input to the next iteration in the refinement algorithm.

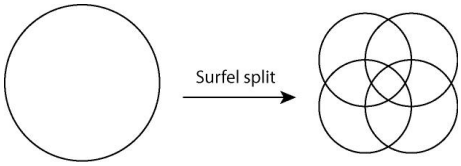

Figure 3.3: Surfel splitting

This iterative algorithm continues until the input point set P^n is empty or a fixed time-limit is reached.

For more details see [Ves04].

Comments

Backface culling (and other culling algorithms) can be incorporated into the algorithm if it is used for rendering purposes. This will make the algorithm view dependent and decrease the number of points that have to be processed. The algorithm can also be enriched with features to handle single isolated particles, or particles that have surface on all sides. Normally the algorithm only needs one or two iterations to generate a good solution.

Except for the splitting process, the algorithm is at many points similar to the Iso-splatting surface extraction algorithm presented in section 3.1.3.

This method is CPU oriented, which means that it has to share CPU time with the granular matter simulation itself.

3.2.3 GPU refinement

No point-based GPU refinement algorithm have been found during our work, but there exists articles that discusses GPU refinement for polygon meshes.

GPU refinement is a recent topic that have become feasible during the last couple of years as GPU processing power and programmability have increased. Still, refinement on GPU is an open research area that will most likely see some more attention in the near future.

The main problem with mesh-refinement is that todays graphics hardware can't generate new vertices on the GPU. One simple approach around this problem is the technique used in [BS05]. They use vertex buffer objects (refinement patterns) to store a grid of vertices on GPU. When a simple triangle (3 vertices) is sent to the GPU, the grid of vertices on GPU can be used to generate new inner vertices for the triangle. These inner vertices can then be displaced in a vertex shader according to some function or attribute sent along with the triangle vertices. In similar fashion, a point primitive can be expanded to for example ten point primitives. These new primitives can then be displaced with a custom method. In this way, custom displacement mapping can be

performed on the GPU according to some predefined values or some procedural approach calculated on the graphics hardware.

Another approach to GPU refinement is to utilize already known and well defined facts from subdivision schemes used for long in computer graphics. This is exactly the approach used in the paper [SJP05] that is an implementation of Catmull-Clark subdivision on GPU.

3.3 Methods for surface rendering

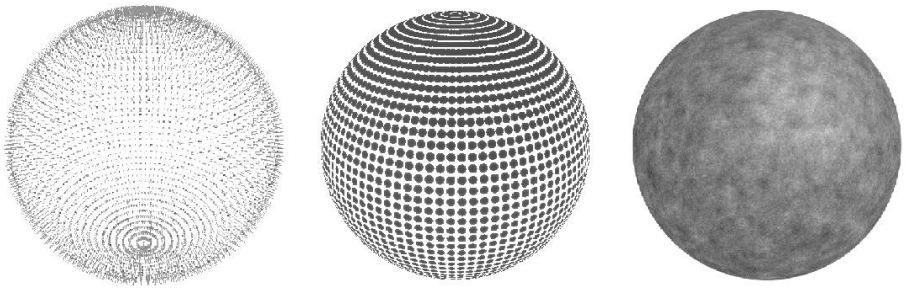

Figure 3.4: Overview of the surface rendering process

The point rendering problem consists of getting a cloud of points (*Point-cloud*) in three dimensions, with normals and other attributes delivered on a point basis to look like a three dimensional object in screen space (figure 3.4). Point rendering can be used both for volume rendering, when the inside of objects can be seen, and surface rendering where a solid surface is rendered. In this thesis focus lies on surface rendering since the main feature of a granular material is its surface.

All algorithms assume that there are no connectivity information, i.e. no explicit mesh information is provided with the point-cloud. The problem then becomes that of taking the point samples on the surface of the mesh and interpolate theses values to an occluding surface without knowing what neighbors a specific point has.

Two different rendering algorithms will be examined. First, a high quality hardware accelerated *Surface Splatting* algorithm will be covered. This algorithm is based on the work in [ZRB*04, BHZK05, ZPvBG01, GP03, RPZ02, PZvBG00]. Second *Voronoi rasterization* [TCH05] will be investigated since this surface splatting algorithm is simple to implement and shows promising results on sparse point-clouds.

Other point rendering algorithms that is not based on surface splatting exists. These methods are not as thoroughly researched as the surface splatting ones, and therefore not covered to the same extent in this work.

3.3.1 Surfels as rendering primitives

Depending on the point-cloud source or what should be rendered, the point samples can contain different data. Normally, samples contain at least a position (3 scalars), a point normal (3 scalars) describing the orientation of the surface at the point and a radius (1 scalar) specifying the influence of the point to the neighborhood. Other data can be color, texture coordinates or any other data that can be useful when rendering

the point. All in all this data can be collected in a structure called a *surfel* (figure 3.5) [PZvBG00], which can be seen as the rendering primitive used throughout the point rendering problem.

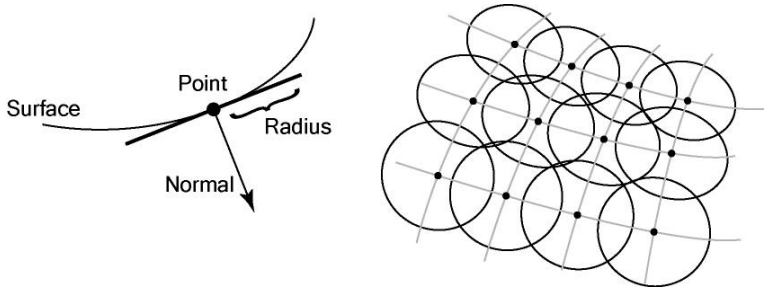

Figure 3.5: Definition of surfel. (left) Surfel attributes. (right) Surfels with influence area for each surfel shown.

The surfel position, normal and radius give a circular disc in object space which is the area of influence for the surfel (figure 3.5). This area can be quadratic, rectangular, circular or elliptical in object space or screen space depending on the specific rendering algorithm used. As seen in the figure the radius of the surfels are chosen such that neighboring surfels overlap. This is what makes it possible to create an occluding surface when rendering the surfels.

The step of rendering a surfel to screen is referred to as "splatting the surfel". From this notion it follows that surfels sometimes are called "splats".

Choosing a surfel rendering primitive

There exists two main methods to send surfels to the graphics hardware for rendering. One use polygons as rendering primitives and the other use point primitives.

In the first method, each surfel it converted to a normal oriented quad (a quad with the point radius as half width and the point normal as face normal). This approach is easy to implement, but introduces polygons and more vertices in the process. This means that the number of surfels that can be sent to the graphics hardware in a limited time frame is decreased.

The second alternative is to only send a point (one vertex) to the GPU and then calculate the extension of the surfel on the GPU from the point attributes. To do this, each surfel is replaced with a conic section. This conic section can be defined from the surfel position, normal and radius. The conic sections are projected through the rendering pipeline with custom vertex and pixel shaders to handle the calculations [ZRB*04]. The main advantage is that only one vertex per surfel is sent through the pipeline instead of four as with quads. This means that using points as primitives are especially attractive for large datasets. The main downside is that a purely point-based rendering pipeline is harder to implement than one based on quads.

3.3.2 Basic surface splatting

Surface splatting is an algorithm that reconstructs an unevenly point sampled three dimensional surface in image space. This means that the algorithm can be seen as a

way to fill in the blanks between the point samples (surfels).

Surface splatting algorithms are normally divided into two or more rendering passes. The two passes common for the algorithms are commonly named *Visibility Splatting* and *Surface Reconstruction*.

Visibility splatting decides which front-facing surfels that belongs to the foremost surface of the mesh from the current viewpoint. Surface reconstruction renders an occluding surface with help of the depth buffer constructed in the visibility splatting pass.

Visibility splatting

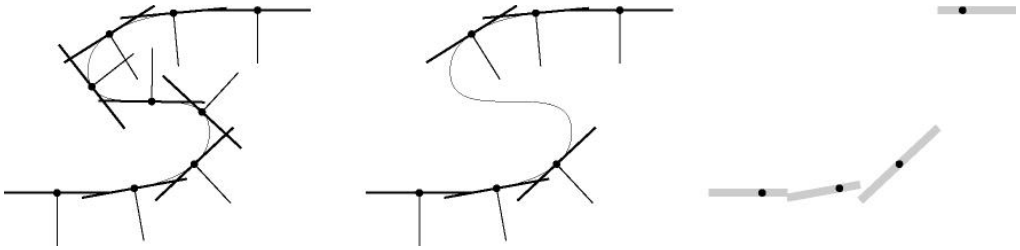

Figure 3.6: Visibility Splatting. (left) Send all surfels through rendering pipeline. (middle) Surfels after backface culling. (right) Result after z-buffering.

In figure 3.6 an overview of the Visibility splatting rendering pass is shown. All figures assumes that viewing direction is from bottom up.

To remove the front facing surfels that does not belong to the foremost surface, visibility splatting is used. All surfels is sent through the rendering pipeline (left figure) while only writing the depth buffer. Backface culling can be used (middle figure). This generates an occluding depth image of the current mesh where edges of splats that are covered by other splats are removed (right figure). This depth buffer can then be used in the next rendering pass.

Surface reconstruction

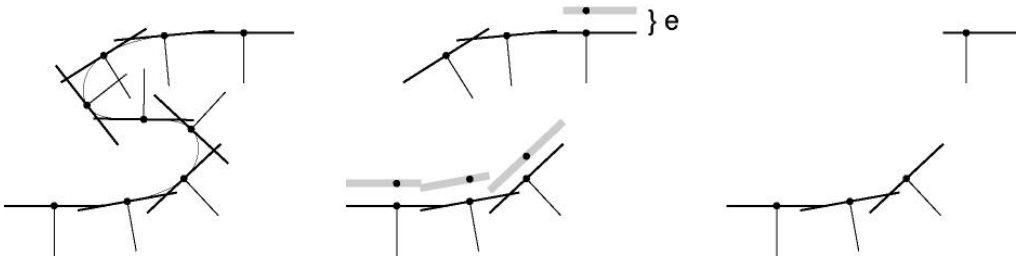

Figure 3.7: Surface reconstruction. (left) Send all surfels through rendering pipeline. (middle) Surfels after backface culling. Grey lines show contents of depth buffer from visibility splatting pass. (right) Surfels rendered after surface reconstruction.

In the next rendering pass, Surface reconstruction (figure 3.7), all surfels are once more sent through the rendering pipeline with backface culling enabled (left figure). This time the depth buffer is not written, but the color buffer (screen) is written instead. A modified z-buffer depth test is perform for the surfels against the depth buffer generated in the first pass (middle figure). This test incorporate a small offset e, in this way all surfels (or parts of surfels) that does not belong to the foremost surface are not written to the color buffer (right figure).

Algorithm

The complete algorithm for basic surface splatting proceeds as follows.

```
First pass (Visibility Splatting):
 Enable backface culling
 Enable depth test
 Render all surfels to the depth buffer

Second pass (Surface Reconstruction):
 Enable backface culling
 Disable depth buffer writing
 Render all surfels to the color buffer (screen)
 Discard surfels with modified z-buffer depth test
 Write non discarded surfels to color buffer
```

This algorithm produce an occluding surface, but the result is not visually pleasing (figure 3.8). To get a smoothly shaded surface modifications to the algorithm are needed.

Figure 3.8: Basic surface splatting.

3.3.3 Smooth surface splatting

To get the result produced in the section "Basic surface splatting" two rendering passes are not really needed. If all surfels are sent through the rendering pipeline, with z-buffer

depth testing, the same result will be acieved. Why the two passes are needed will become apparent in this section.

Transparent surfels

Up until now, surfels have been represented by opaque discs in object space. To get smooth shading this representation need to change to discs that are transparent at the edges and opaque in the middle (figure 3.9). This, together with the overlapping of the surfels, will make it possible to blend the surfels together at the edges, producing a smoothly shaded surface.

The idea is to use a Gaussian filter as a smoothing kernel for each surfel. If quads are used as surfel rendering primitives, these smoothing kernels can be precalculated and put into an alpha texture that can be used to texture the quad. If point primitives are used, the smoothing kernel can be calculated with help of the conic section (see 3.3.1) in the pixel shader.

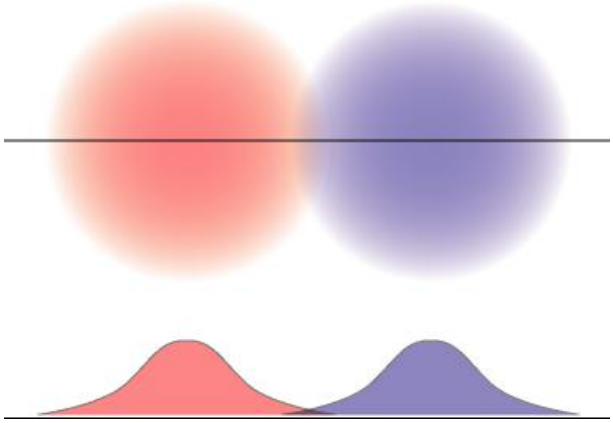

Figure 3.9: Gaussian surfels. (top) Splats shown from top with blending occurring at the middle between splats. (bottom) Side-view showing Gaussian distribution of the two splats.

Blending the surfels

To smooth the surface, depth testing is disabled to get the overlaps to blend. The problem is that only surfels that belongs to the foremost surface of the mesh should be blended. If all front facing surfels were blended together the complete surface would look transparent. The solution to this problem is what Visibility splatting is designed for. With this algorithm only surfels that belongs to the foremost surface will be blended (figure 3.7).

The algorithm for the Surface reconstruction then becomes:

```
Second pass (Surface Reconstruction):
 Enable backface culling
 Enable blending
 Disable depth buffer writing
 Render all surfels to the color buffer (screen)
 Discard surfels with modified z-buffer depth test
 Accumulate non discarded surfels to color buffer
```

This produces a smoothly shaded surface as seen in figure 3.10.

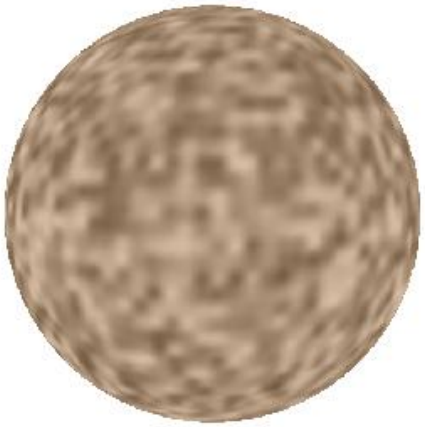

Figure 3.10: Smooth surface splatting.

3.3.4 Voronoi rasterization

If no surface refinement is performed, the point set obtained from the physical simulation can be sparse. As pointed out in 1.1, most point rendering algorithms are made to work on dense point sets obtained from laser scanners. If these algorithms are used on a sparse point-cloud problems such as blurry or non-occluding surfaces can arise. One suggested algorithm to avoid this problem is *Voronoi rasterization* [TCH05].

Voronoi rasterization is at its base a two pass surface splatting algorithm. The first pass, Visibility Splatting, is the same as in the basic surface splatting technique (see 3.3.2).

In the second pass, Voronoi rasterization or "r-buffering", the objective is to clip splats to each other, producing a Voronoi diagram over the surface (figure 3.11). This will give a flat-shaded look of the surface with the "flaps" seen in figure 3.8 removed.

Voronoi diagram

A voronoi diagram (figure 3.12) is a way to partition space from scattered point samples. The partition results in convex polygons which each contain one point. For each polygon, every part of the polygon is closer to the point within, then to any other point [Mat06].

Figure 3.11: Voronoi rasterization

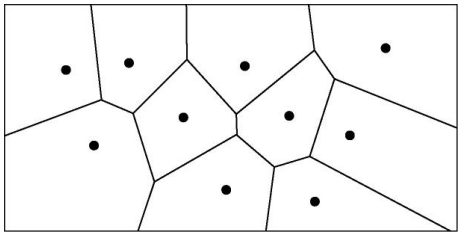

Figure 3.12: Voronoi diagram

This is the mathematical foundation that is used in voronoi rasterization, and which has given the method its name.

r-buffering

The name r-buffering comes from the test in the algorithm that is used to clip the splats to each other. This test is basically a z-buffer depth test, with the difference that the radius, r, is used instead of the depth, z, in the test. To get this behavior, shader programming is necessary, which is different from the basic surface splatting algorithm.

In figure 3.13 an example of r-buffring is shown. In this example, three splats are sent to the graphics hardware (left figure). For each fragment, the distance to the surfel center is written to the depth buffer. The values along the black line (left figure) that is written to the depth buffer, is shown in the middle top figure. With a depth test, fragments with shorter distance to the surfel center are prioritized before fragments futher back. The resulting fragments written along the line in this example are shown in the middle bottom figure. For the complete image, this gives the result shown in the right figure.

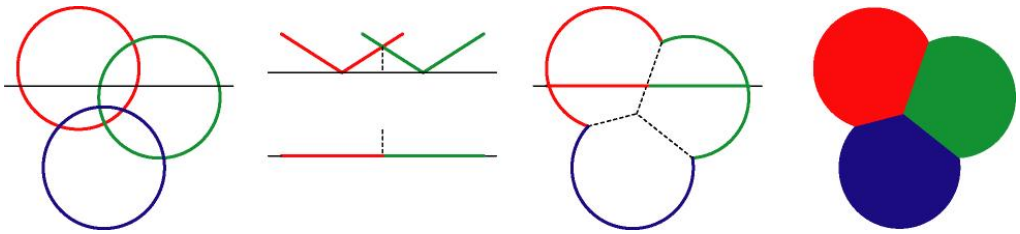

Figure 3.13: r-buffering. (left) Three splats sent to rendering. (middle top) z-buffer values along black line in left figure. (middle bottom) Resulting color along line after z-buffering with r-values has been performed. (right) Resulting image, with splats clipped to each other.

The complete algorithm with the visibility splatting pass from 3.3.2 and the second pass, r-buffering is:

```
Second pass (r-buffering):
 Enable backface culling
 Enable depth testing
 Bind the depth buffer from first pass as a readable texture
 Send all splats to the graphics hardware
 In fragment shader:
  Discard fragments with modified z-buffer depth test
  Write r, distance to surfel center, to z-buffer
  Write non discarded fragments to color buffer
```

The modified z-buffer depth test operates in the same way as in the surface reconstruction rendering pass in 3.3.2.

Attribute blending

To get a higher quality shading than flat-shading one extra rendering pass is needed. This rendering pass, attribute blending, will be performed between the two passes in the basic algorithm already described. The purpose with this rendering pass is to produce a smooth attribute field over the surface. Normally this attribute field consists of color, but any other attribute (for example normals or texture coordinates), that can be used for shading can be blended.

This rendering pass is somewhat similar to the surface reconstruction pass used in the smoothed surface splatting algorithm (see 3.3.3). The main difference is that the surfels are rendered with twice the surfel radius. With this approach surfels overlap the centers of neighboring surfels. This is needed if a smooth field should be rendered for the complete range between the surfel centers.

The smoothing kernel used to blend the splats is a quasi-Gaussian kernel

$$w(d) = 3\frac{d^2}{r^2} - 2\frac{d^3}{r^3} + 1 \tag{3.11}$$

over each disc of radius r where d is the distance to the center of disc. In the fragment shader this equation is used to scale the discs attribute value before it is added to the

output of the pixel. All in all, this produces a smoothly blended attribute field over the surface.

The algorithm for the pass is

```
Attribute blending pass:
 Enable backface culling
 Enable blending
 Disable depth buffer writing
 Render all surfels to a color buffer with radius x2
 In fragment shader:
  Discard surfels with modified z-buffer depth test
  Accumulate non discarded surfels to color buffer
```

The attribute field is stored in a (float) texture, and can later be used in the r-buffering pass for deferred shading (see 3.4.1) of the final surface.

Comments

The thing that separates the voronoi rasterization algorithm most from a high quality surface splatting algorithm is that flat shaded representations of the geometry can be produced. This can be useful for applications that should show the underlying geometry of the surface.

The method is also very easy to implement, requires no preprocessing and shows good performance on sparse point sets. This makes it a good candidated for visualization of dynamic isosurfaces as is our case with the granular matter simulation.

3.3.5 Other algorithms

There exists both simpler and more complex ways to point rendering than Surface splatting and Voronoi rasterization.

One simple, straightforward point rendering algorithm is presented in [BC03]. This algorithm uses pure GL_POINT's as rendering primitives, i.e. surfels are never introduced. The rendering primitive then becomes an opaque screen space quad. With this approach, no blending between points can be performed. Shading can only be performed for each point. This makes the method fast, but since no interpolation of attributes is performed, the method is not suited for sparse point-clouds.

A more complex approach is used in [KV03]. In this paper the surfel concept is extended to something named "Differential Points". These differential points includes the common surfel data but also adds curvature information. This makes it possible to get a more exact description of how the surface area around a point looks. With this approach, the surface can be rendered as a "collection of local geometries", which can produce higher quality rendering with fewer points due to the extra information.

In the paper [KK05], a rendering method that use pure points is introduced. This method is implemented on GPU and relies only on the points sent to GPU for surface rendering. The normals needed for shading are calculated from the points on the fly as rendering proceeds. This makes the method attractive for many applications, but the algorithm only performes well on dense point sets.

3.4 Methods for surface shading

The last step in the point rendering pipeline is the shading of the surface produced in the surface rendering steps. This shading is tightly coupled with the surface rendering itself, which introduce many restrictions.

In this section a few basic methods that can be used for surface shading is presented. After that their application are extended and discussed in the next chapter *"Our contribution: Point-based texture splatting"*.

3.4.1 Deferred shading

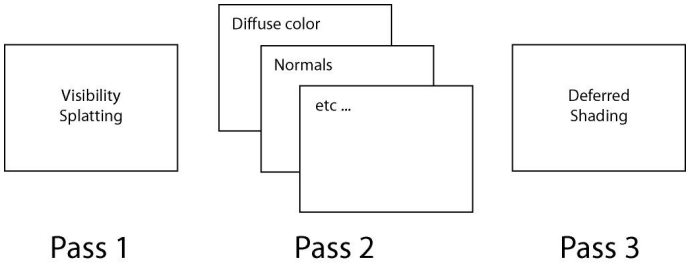

Figure 3.14: Deferred shading

The normal shading technique usually sends all primitives to the graphics hardware, and writes color for the primitives to the framebuffer in a single pass. In more advanced shading techniques, it is common to send the primitives several times to the graphics hardware and each time modify the color that was generated in a previous pass. This approach to shading is called multipass rendering.

With the introduction of better graphics buffers on the hardware and the possibility to write several buffers in one renderingpass (with Multiple Rendering Targets, MRT), deferred shading has been introduced. In this technique, the geometry is sent only once to the rendering hardware, and several buffers on the GPU are written when the geometry is rasterized. Then in a final rendering pass, a fullscreen sized quad is sent to the graphics hardware. When each fragment of this quad is rasterized, the graphics buffers generated in the earlier rendering pass are accessed. With this technique, final shading calculations with data from several graphics buffers at once are performed.

An example application for splatting can be seen in figure 3.14.

In the first rendering pass, visibility splatting is performed as usual. The depth buffer is stored in a texture on GPU for later use.

In the second surface reconstruction pass, multiple rendering targets is used. The diffuse color is splatted as normal in this pass, but this time to a texture. To a second texture, normals are written with a similar technique as for the diffuse color. This gives two textures, one with a smooth surface of diffuse color and another with a smooth normal field over the surface.

In the final rendering pass, a full sized screen aligned quad is rendered. For each fragment that is rasterized in this quad, the corresponding pixel in the diffuse color and normal buffer is accessed. The final per-pixel shading is calculated and the result is written to the frame buffer.

More information on deferred shading for surface splatting can be found in [BHZK05].

3.4.2 Perlin noise

Perlin noise is the foundation for many procedural shading algorithms. The idea was introduced in [Per85] and has since then been used in both computer games, image rendering and special effects for movies.

The basis of Perlin noise is the combination of several interpolated noise functions with different frequencies and amplitudes. With these functions, several natural looking patterns such as wood, marble or sand can be created.

Images from [Eli06] have been borrowed for illustrations.

Making noise

To generate Perlin noise, one first need a noise function, $n(x)$, as a starting point. The noise function should be a seeded random number generator, i.e. always returns the same output for a certain input. By using this generator, a two dimensional picture as seen in figure 3.15 (left) can be generated.

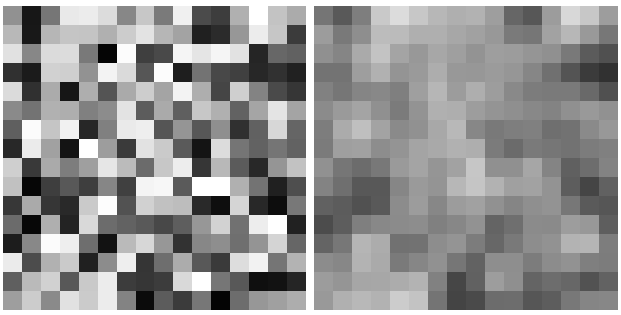

Figure 3.15: Perlin noise base function

Second, this generated noise is interpolated to get a continuous function, $i(x)$ (figure 3.15, right). This can be done with linear interpolation, or better, cosine alternative cubic interpolation.

The amplitude and frequency of the interpolated noise function, $i(x)$, can be varied. This can produce images as seen in figure 3.16.

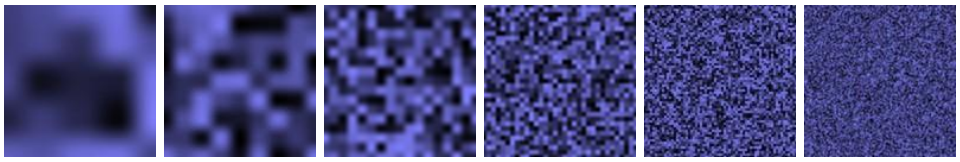

Figure 3.16: Noise octaves

If several interpolated noise functions with different amplitudes and frequencies are added together, a new function is produced, a Perlin noise function $p(x)$ (figure 3.17). In the function,

$$p(x) = i(x) + \frac{i(2x)}{2} + \frac{i(4x)}{4} + \frac{i(8x)}{8} + \cdots \tag{3.12}$$

each term in the summation forms what is commonly named an octave. In this way, Perlin noise functions with different number of octaves and varying frequencies and amplitudes can be produced.

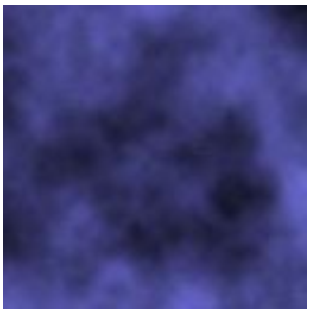

Figure 3.17: Perlin noise

With help of the perlin noise pattern in figure 3.17 and some extra calculations, several natural looking patterns such as wood and marble can be produced.

Chapter 4

Our contribution: Point-based texture splatting

In this chapter, a dynamic, granular matter shading algorithm will be presented. This algorithm builds upon texture splatting techniques, with parts that can be proceduraly defined. The new method is constructed to work in real-time, together with a hardware accelerated surface splatting algorithm.

4.1 Related work

Little work has been done on increasing the shading detail of low resolution mesh-free, granular matter simulations, especially for real-time point-rendering applications. Most of the point-rendering algorithms that work on sparse point sets, which is the current norm for simulations, only deal with non textured shading.

For our work, two articles can be mentioned as inspiration for what we want to achieve with our solution.

The first article, *"Animating Sand as a Fluid"* [ZB05] by Y. Zhu and R. Bridson, describes an implementation of a simulation that gives pleasant results on sand. But since this method is not even near real-time, it is not applicable to our situation. But the look of the animated sand bunny serves as a good measurement of what we want to achieve.

The second article, *"Animating Lava Flows"* [SAC*99] by D. Stora, P.O. Agliati, et al., shows a system for animating and rendering lava flows. The simulation method used is SPH, and one important thing achieved in their work, is texture details that flow with the underlying simulation. The method is not real-time and the visualization of the simulation is based upon polygons. This means that the algorithm is not directly applicable to our situation, but their method shows some similarities to the resulting implementation described in this thesis.

4.2 Shading requirements

The shading algorithm should be able to give the surface a high resolution granular matter texture. The method should be flexible enough to at least support some different kind of granular materials, such as sand, soil or salt.

Ideally, we want the shading algorithm to exhibit a few important features:

- Increase surface resolution
- Handle moving, deformable geometry
- Real-time performance

Increase surface resolution

As of today, it is possible to simulate approximately 3000 particles in real-time with an optimized SPH simulation. Of these particles, somewhere around 1000 particles are part of the surface that should be rendered. If a high resolution rendering of this simulation is wanted, for example an image of 800x800 pixels, the information from 1000 particles should fill 640 000 on screen pixels. This amounts to 640 pixels, or approximately 25x25 pixels per simulated particle. With these numbers, one realizes that rendering each particle with one single color, giving areas of 25x25 pixels with equal color, is not sufficient to give a realistic look of the simulated matter. Hence, a shading method that increases the resolution of the shaded surface is needed.

Handle moving, deformable geometry

The next problem is that the simulation makes the geometry (particles) move and deform very much in a limited time frame. To have a realistic granular matter simulation, the simulated matter should be able to be stationary in one frame, slowly be poured in the next and splashed around in the last frame. During all this movement, the high resolution look of the surface and single particles have to be intact. Plain texture mapping is not easily applicable to this situation, since defining texture mapping coordinates to handle the described situation is not feasible.

Real-time performance

Shading the surface must be done in real-time together with the other parts of the application. The simulation and surface extraction (and refinement) parts of the complete system are, as of today run on CPU. To get real-time performance, the shading algorithm therefore should be run mostly on GPU, since this would not put any more limitations on the other parts of the application.

4.3 Point-based texture splatting

Our solution enhance the SPH particles with some extra values that control the shading. These values are fed as seeds to the shading algorithm that produce a unique granular matter texture for that part of the surface. In this section, the different steps to produce this behavior is presented and discussed.

Base color

Each particle in the simulation is assigned a vector with three values. These values are chosen in a preprocessing step and is used as base color for the particle. When a particle is extracted from the simulation to be part of the rendering, the base color is assigned to the surfel corresponding the particle.

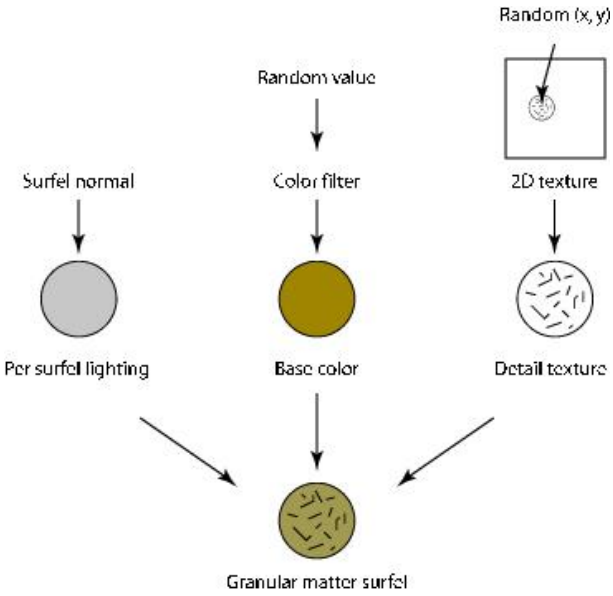

Figure 4.1: Shading algorithm outline.

How to chose the base color values can be done in many ways depending on what look that is wanted. As starting point, one random value between zero and one have been chosen. These values are then fed into a color filter step, that bias the single random value towards a random brown color with three RGB values.

The base color gives the surface large scale variations that can not be done with detail texturing. It is with this method possible to define darker and brighter parts (for example, dry or wet sand) in the granular matter.

Detail texture

Together with the surface rendering algorithm, the base color provides a smooth diffuse color over the surface. The resolution of this surface texture depends on how many particles that are extracted from the simulation. With a typical situation of, for example, 1000 particles, the detail is very low and does not give a realistic view of the simulation.

To enhance the resolution, we use a detail texture that is blended with the base color with multitexturing. Since the object that is textured is arbitrary and can change in any way, there exist no good way to generate normal texture coordinates for the surfels. Our approach to texturing is to generate a random texture coordinate, per surfel, that is used to find the area of texture that should be sampled to the surfel.

In figure 4.2 the composition of textured surfels is shown. The random values is used to find an area in a 2D texture. An interpolation of the texture coordinate over the surfel is produced. This is done strictly per surfel, without taking neighbors into the calculation. Each surfel is then rendered with a gaussian alpha mask over the surfel (as normally done with surface splatting). With blending, this procedure generates an occluding detail texture over the surface.

This approach to texturing is somewhat similar to the a method called ”Texture Splatting” that has been widely used to make detailed terrain rendering in games [Blo00].

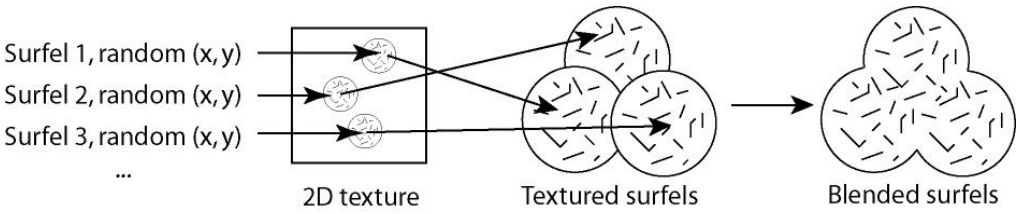

Figure 4.2: Texture splatting

Lighting

Basic lighting is done per surfel and is blended together with the base color and detail texture before rendering. With a multipass algorithm, or multiple render targets (MRT), per-pixel lighting is possible to combine with the algorithm.

4.4 Extensions

If the lighting is excluded from the point-based texture splatting algorithm, it basically is a way to generate only the diffuse color for the shading. This means that many other shading algorithms can be added on top of the method to enhance the shading. The method is for example perfectly suited to incorporate into a deferred shading algorithm (see 3.4.1).

Bump mapping

One interesting extension to point-based texture splatting is to use an equal approach for splatting normals. In this way, the detail texture becomes a detailed normal map that produce a varying field of normals over the surface. This normal map can then be used for bump mapping in a deferred per pixel shading algorithm.

4.5 Discussion

The algorithm gives plausible results on shading of granular matter simulations, but the method is not in any way general enough to texture any deformable material. Basically, the lack of structure in granular materials are what makes the texturing method feasible.

Texture limitation

To give a good result, the texture chosen should be fairly big ($>$ 512x512 pixels). Furthermore the texture can not contain any larger structures (low frequencies), since the areas sampled from the texture are small. Textures with patterns are also ruled out since no texture synthesis is done in order to correct the texture overlap between surfels. This means that textures such as wood or brick is not possible to use.

In our application, both procedural texture and photographed textures have been tested with varying results (see 4.6).

Texture stretch

In the current implementation, the texture coordinates is linearly interpolated over the surfels. This is done per surfel with the texture coordinate at the center as base value. The value together with a radius r, is used to define a circular texture area that is mapped directly to the circular area of the surfel.

How to chose the radius r is an open topic. In our implementation, r is a user defined value that can be changed for experimental purposes. How to chose r in a way that avoids aliasing and other common texturing problems have not been analyzed.

Chosing texture coordinates

The texture coordinates used to access the detail texture is in the current solution pseduorandom numbers. How to chose this coordinates is an open question. In our current solution, the first two values of the base color is used to access the texture. One good thing with this approach is that surfels with similar colors also access similar areas of the detail texture. This could give some more coherent shading for textures with larger areas of different colors.

4.6 Example renderings

See section 6.2, *"Rendering quality"*, for example pictures of the texturing method.

Chapter 5

Implementation overview

The focus of our work have been to examine different point-based techniques that could be used to render a particle-based granular matter simulation. This means that a finished implementation haven't been done and also couldn't be done because the final goal has never been thoroughly specified. Therefore this chapter will start with an overview of a suggested system, rather than a description of a finished system. After that, a short summary of what parts we have implemented will follow.

5.1 Implementation guidelines

In this section, a suggested implementation of a complete system will be presented. These suggestions builds upon the theoretical work put into methods together with things learned with our own prototype implementation (see 5.2).

5.1.1 System overview

The ideal implementation would divide the system modules into the parts pointed out in "Problem description" (chapter 2). With these four modules, together with the simulation, the complete system becomes what is shown in figure 5.1.

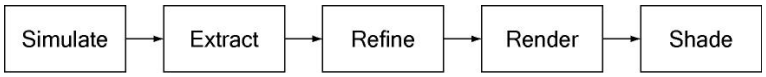

Figure 5.1: System overview

As usual, theory and practice do not follow the same principles. The main concern when implementing a point-based granular matter shading pipeline, is that the rendering and shading are tightly coupled and put many restrictions on each other. Depending on choice of algorithm, the extraction and refinement parts can also be performed in one or two steps (figure 5.2).

With our suggested shading algorithm (chapter "Our contribution: Dynamic procedural shading"), some dependencies even exists between the simulation and the shading since the simulation particles have to "carry" information on how the surface should be shaded in the area surrounding the particle (see 4.3).

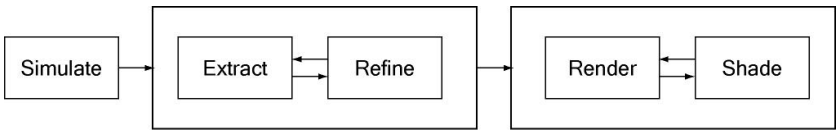

Figure 5.2: Refined system overview

5.1.2 Pipeline overview

The complete simulation and rendering pipeline can be divided into five basic steps. A walkthrough of the data dependencies is shown in figure 5.3.

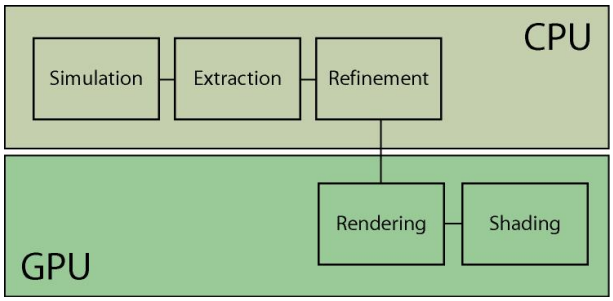

Figure 5.3: Pipeline overview

Simulation

Simulation is in our case done on CPU. Output from the simulation is particles which have many values calculated per particle. The most interesting ones are positions and the gradients of the color field.

An hash map (or some other spatial data strucutre) for neighborhood search is normally also present with the simulation since this is crucial for the performance of the simulation itself. This hash map is useful for some of the extraction and refinement algorithms described.

Extraction

The extraction algorithm basically use the positions of the particles together with the color field gradient values to identify surface particles. Output from the step consists of positions with normals (calculated from the color field gradient).

Refinement

Surface refinement algorithms such as "Point-cloud refinement" (see 3.2.1) only use the surface particles positions and normals to generate more surface particles.

Other surface refinement algorithms, such as "SPH isosurface refinement" (see 3.2.2) use the particles from the surface extraction step as basis. When new particles are generated, the hash map from the simulation is used to recalculate the color field gradients

for the new particles. This makes the extraction and refinement somewhat tangled and therefore not completely separable in this algorithm.

With both approaches the output from the refinement step surface particles with positions and normals.

Rendering and shading

The positions and normals are all that is needed to perform non-textured surface splatting with high quality lighting. To do this the procedure in section 3.3.2 is followed.

The points and normals are both sent to the graphics hardware, and from there on, no job is done by CPU. The information sent between the rendering passes, such as depth information, is at all times stored as textures on GPU.

5.1.3 Pipeline throughput

One of the bottlenecks in the described pipeline is the surface refinement step. This step is expensive and often occupies equal or more CPU time then the simulation itself. Therefore, a boost in performance can be expected if this step could be moved to GPU. The problem is that the suggested algorithms all use some kind of spatial hash-map or other spatial data structure to do neighborhood searches, and no simple and efficient way to perform this on GPU exists.

If the number of simulated particles increase, or the surface refinement step is performed on CPU, another bottleneck can come in focus, namely the transport of points and normals between CPU and GPU. One way to deal with this bottleneck is to send point primitives instead of quad primitives as surfel representations to the GPU. This would send four times less vertices to GPU but instead the calculation burden on the GPU increase somewhat and the implementation is more complex.

5.1.4 Accessing the graphics hardware

The OpenGL API [Ope06] have been used for testing the methods described in this thesis. A brief summation of useful techniques that is needed for implementation will be covered in this section. Basic OpenGL knowledge is assumed.

OpenGL Extensions

Since quite a few recent graphics hardware features are needed to support the algorithms, OpenGL extensions are heavily used. To handle extension initialization, the utility library GLEW have been used [GLE06].

For detail control of color and depth buffers, the Frame Buffer Object (FBO) extension have been used. This is made available through the `GL_EXT_framebuffer_object` extension.

Floating point textures are used through the `GL_NV_float_buffer` extension. All algorithms operates without floating point textures, but to get most out the methods, floating point textures are needed.

Full 32-bit floating point support through the entire rendering pipeline (with floating point blending) makes things easier when developing the algorithms, since for example clamping of values does not become an issue.

When doing deferred shading, Multiple Rendering Targets (MRT) comes in handy since writing several buffers in one rendering pass boosts performance. MRT is exposed through the `GL_ARB_draw_buffers` extension.

GL Shading Language

GL Shading Language (GLSL), have been used for the programmable shader parts of the project. Both vertex and pixel shaders are heavily utilized. Since OpenGL 2.0, these extensions belong to the base OpenGL API.

No extra tools for shader handling or debuging have been used. This have made the process of learning GLSL slower. One advantage have been that since all functionality have been implemented from scratch, GLSL and OpenGL shading functionality have been thoroughly learned.

5.2 Our implementation

The implemented prototype have mainly been developed to learn two things. First, how the methods work, and second, how to put the different methods together to a complete system.

Point-based methods are not always straightforward to implement and can require some tinkering to become fully functional. Furthermore, GPU implementations is not as easy to debug as their CPU counterparts. A high-level graphics library could help somewhat in the implementation, but also hide some details of the algorithms. Because of this, the system have been implemented stand alone, without a direct connection to the simulation or other API.

Some downsides follow from this. First of all, no performance measurements of a complete simulation and visualization system have been done. Second, only raw SPH particle data, without any spatial data structure, have been used in the rendering process. Since no effort have been put into implementing or finding such a data structure, no extraction or refinement algorithms that require neighborhood searches have been implemented.

Hardware issues

A full floating point rendering pipeline and Multiple Render Targets (see 5.1.4) have not been avavible on the development platform. This have meant that some extra care have been taken during implementation and that some functionality have been harder to implement.

Surface extraction and refinement

For surface extraction and refinement, only the simplest possible algorithm have been implemented, namely the SPH color field algorithm in section 3.1.2. This means that the point-cloud data used for surface rendering and shading is sparse and quite noisy. The downside with this is that a really smooth surface have been impossible to render (under these circumstances). One good thing though, is that this noisy, sparse point-set have put higher demands on the rendering scheme to produce a good result with bad input data.

Surface rendering and shading

Mainly three surface rendering methods have been implemented.

First, a Voronoi rasterization (see 3.3.4) renderer. With this renderer, the complete shading system described in chapter four, have been tested.

Second is a surface splatting algorithm based on the method described in 3.3.2. This implementation also features the complete shading system described in chapter four, and is therefore directly comparable to the Voronoi rasterization renderer.

Last, a EWA surface splatting algorithm that is based on the implementation in PointShop3D [Poi06] have been incorporated into our implementation. This renderer only features basic shading without detail textures. The interesting part is instead that the renderer is completely based on GL_POINTS, and thus sends less vertices to GPU than the other renderers.

Details on the texturing implementation can be found in chapter 4.

5.2.1 Development platform

All development and testing have been performed on Linux with NVIDIA graphics hardware. The latest OpenGL 2.0 drivers have been used. Basic window handling have been done with GLUT. All used libraries are cross platform so everything should apply to other supported platforms as well. Some of the extensions pointed out in 5.1.4 might have different names or not exist if other hardware than NVIDIA's is used.

The main graphics hardware used for development and testing have been a NVIDIA GeForce FX Go5200, but systems with GeForce 6200 and GeForce 7300 have also been available for testing. The main downside with the graphics hardware that have been available for testing, is that all three graphics cards are at the low-end in the respective series. One thing common with the cards is the much lower fillrate than for high-end cards in the graphics card series. This could put some restrictions on the performance of the algorithms since fillrate intensive features such as blending with much overdraw is used in the algorithms.

Chapter 6

Results and discussion

As pointed out in the objective (see 1.3), the main aim was to investigate point-based methods that could be used to render a granular matter simulation. This have resulted in identification of four steps, which each needs their own method, that are part of the complete visualization problem.

Some extra effort have been put into understanding hardware accelerated surface splatting and point-based rendering techniques in general, but since this is a large field with many recent papers on the subject, a conclusive summary is hard, if at all possible, to produce.

6.1 Performance

Performance is always an important issue when it comes to interactive applications. To get the most out of the methods, a thorough analysis of bottlenecks and optimization of these need to be done. Since this thesis work is mostly theoretical, no extra effort have been put into doing this work for our implementation. The more important part have been to identify methods that are near real-time and give good visual results. Furthermore, it is important that the methods are mostly GPU-oriented since the CPU runs the simulation of the granular matter. Some preliminary notes on performance and scalability will be given though.

Test description

Three datasets have been used to test the algorithms (figure 6.1). First, a sphere dataset consisting of 10000 points. Second, a SPH simulation dataset of 2000 points. Finally, a SPH simulation dataset of 12000 points. The SPH datasets have been exported from a real simulation and have been used both to test performance and how the shading algorithm behaves under deformation of the surface due to simulation.

All tests have been performed with a screen size of 512x512 pixels. For testing of the shading algorithm, the procedurally generated texture seen in bottom left of figure 6.2 have been used.

Surface extraction and refinement

No real effort have been put into the implementation of a good surface extraction and refinement algorithm. Furthermore, these parts of the problem is somewhat dependent

Figure 6.1: Test datasets

on what type of data the underlying simulation can provide and our implementation is standalone. Therefore no notes on either performance or scalability can be given.

Surface rendering

The most important thing in this work have been to propose a high quality shading algorithm that renders a plausible granular matter surface. But since the algorithm should work in a real-time application, performance is also important. Therefore preliminary frames per second (FPS) measurements are provided.

Two different rendering pipelines have been tested. First, a surface splatting pipeline based upon the method in 3.3.2 and second, a voronoi rasterization pipeline with the method in 3.3.4. Both pipelines support the complete shading system described in chapter 4.

Table 6.1: Performance (FPS)

		Sphere	SPH 2000	SPH 12000
Geforce 5200 FX	Surface splatting	5	17	10
	Voronoi rasterization	2	10	4
		Sphere	SPH 2000	SPH 12000
Geforce 6200	Surface splatting	28	110	52
	Voronoi rasterization	4	26	4
		Sphere	SPH 2000	SPH 12000
Geforce 7300	Surface splatting	40	170	85
	Voronoi rasterization	5	32	4

Table 6.1 shows the frames per second (FPS) results for the different tests. For interactive use, frame rates (FPS) above 24 is preferred. The frame varies somewhat with changing viewpoints since different amounts of surfels are rendered due to culling, therefore the FPS measurements in the table is approximate.

As seen in the table, real-time performance is achieved, with marginal, for the surface splatting algorithm on the GeForce 6200 and GeForce 7300 graphics cards. This shows the validity of this algorithm and suggests that there exists possibilities to improve the quality of the shading algorithm and still achieve real-time performance.

The Voronoi rasterization algorithm on the other hand, shows low performance when

the number of primitives increase. This suggest that the graphics hardware is not the bottleneck in this algorithm for large datasets.

Performance measurements on the different parts of the application have not been performed.

Comments

The performance measurements is very rudimentary, but since a plausible real-time solution have been achieved the main goal is still reached. Analysis of the different parts of the rendering pipeline and location of bottlenecks would be interesting as a future development.

6.2 Rendering quality

Since the rendering method is made to work with a real-time deformable geometry, still pictures of it will not make it justice. Some screen shots will be provided though, to give an idea of what visual results that can be expected.

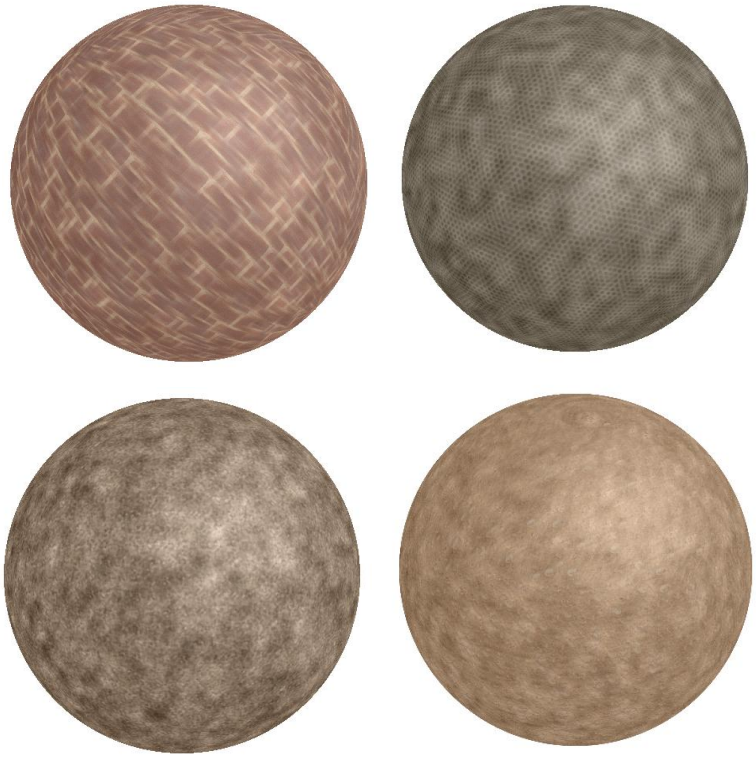

Figure 6.2: Rendering quality

In figure 6.2 four different textured spheres are shown. The upper left, upper right and lower right images all use textures from photographs. The lower left image use a

procedurally generated texture developed specific for a granular matter simulation of sand or soil.

The upper right and lower right pictures features sample textures with only high frequency contents. As seen these two pictures give a high resolution look with details that exceeds the underlying surfels (see 3.10 for comparison of shading without detail texture).

As pointed out, the algorithm fails if the texture contains lower frequencies, i.e. patterns that stretch over long distance. A typical failure of the texturing method is shown in the upper left picture where a brick pattern have been used to texture the sphere.

Procedural generation of the texture (lower left) give a detailed control of what surface texture you want to have and give a result that is comparative to one with a photographed texture (lower right).

Figure 6.3 shows two images with the complete system in action. The left image show a simulation of 2000 particles and the right, 12000 particles.

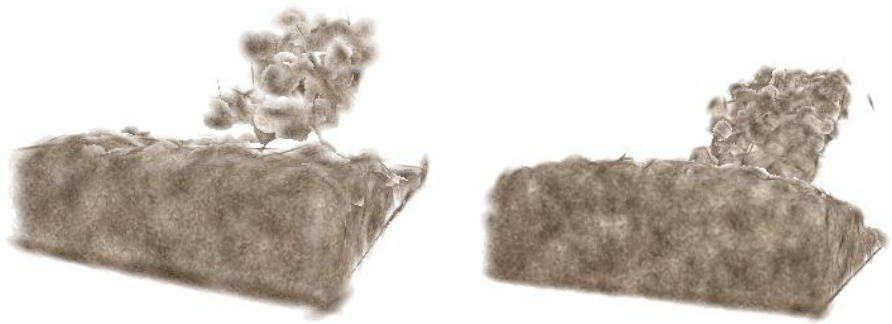

Figure 6.3: Rendering of SPH simulation

The most noticeable thing in both images is the lack of surface refinement which makes the results less attractive. In the left image, single surfels can be seen which is far from optimal.

6.2.1 Limitations

The main limitation with the implemented solution is the low surface density. As pointed out in section 5.2, non of the surface refinement methods have been implemented, and as seen in figure 6.3 this shows in the result. Still, the result is plausible and give a good feel of movement in the deformable simulation, which is the most important thing with the texturing algorithm.

Chapter 7

Summary and conclusions

Visualization of a granular matter simulation is a multi part problem. Four main steps have been identified and methods for each step have been evaluated. Even though many methods have been studied, and some have been implemented, the complete problem is nowhere near a solution. One result of the thesis is to provide a common framework in which methods for each subproblem can be analyzed.

A working implementation, together with a suggestion of a complete system, have been produced. The main contribution with this implementation is the novel texturing technique, which have proved to give plausible results on granular matter simulations.

7.1 Future work

The main identified problem is the surface refinement step. A real-time point-set surface refinement method, that work alongside the simulation and rendering, is yet to be found. Here it would be interesting to investigate the new developments in GPU data structures that could make it possible to implement fast neighborhood searches on GPU.

One thing that have not been thoroughly analyzed in this thesis is how granular matter should be visualized under different circumstances. Sand, for example, can be slowly poured over an edge to provide a "rain" of grains of sand. Since it is not yet feasible to handle the simulation one grain at a time, it would be interesting to find out if the visualization method could handle this by custom particle systems or some other method.

7.2 Limitations

One topic that have not been covered is the more advanced aspects of high quality surface splatting. This includes EWA filtering, normalization, correct z-buffer offset for splats and more. An overview of these methods should have been good to have in the thesis, but time did not suffice.

During the work, much time was spent on the different problems that comes up when trying to implement hardware accelerated surface splatting. Therefore a complete source code together with a walkthrough would have been good to have in the thesis.

7.3 Acknowledgements

Thanks goes to my supervisor, Anders Backman, VRlab, Umeå University for his support and ideas throughout the work and Kenneth Holmlund, VRlab, Umeå University for his help with SPH, surface extraction and ideas in the startup of the thesis work.

References

[BC03] Brentzen J., Christensen N.: Hardware accelerated point rendering of isosurfaces. In *The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2003* (2003).

[BHZK05] Botsch M., Hornung A., Zwicker M., Kobbelt L.: High-quality surface splatting on today's gpus. In *Eurographics Symposium on Point-Based Graphics* (2005), Pauly M., Zwicker M., (Eds.).

[Blo00] Bloom C.: Terrain texture compositing by blending in the frame-buffer. Webpage, 7 2000. http://www.cbloom.com/3d/techdocs/splatting.txt, visited 2006-06-07.

[BS05] Boubekeur T., Schlick C.: Generic mesh refinement on gpu. In *Proceedings of ACM SIGGRAPH/Eurographics Graphics Hardware* (2005).

[CHJ03] Co C. S., Hamann B., Joy K. I.: Iso-splatting: A Point-based Alternative to Isosurface Visualization. In *Proceedings of Pacific Graphics 2003* (Oct. 8–10 2003), Rokne J., Wang W.,, Klein R., (Eds.), pp. 325–334.

[Eli06] Elias H.: Perlin noise. Webpage, 13 2006. `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`, visited 2006-06-13.

[GBP04a] Guennebaud G., Barthe L., Paulin M.: Dynamic Surfel Set Refinement for High Quality Rendering . In *Computer & Graphics* , vol. 28. Elsevier Science, 2004, pp. 827–838.

[GBP04b] Guennebaud G., Barthe L., Paulin M.: Real-Time Point Cloud Refinement. In *Symposium on Point-Based Graphics, Zurich* (2-4 juin 2004), Alexa M., Gross M., Pfister H.,, Rusinkiewicz S., (Eds.), Eurographics/IEEE Computer Society TCVG, pp. 41–49.

[GBP05] Guennebaud G., Barthe L., Paulin M.: Interpolatory Refinement for Real-Time Processing of Point-Based Geometry . In *Computer Graphics Forum, Eurographics 2005 conference proceedings* , vol. 24. septembre 2005, pp. 657–667.

[GLE06] GLEW: Glew: The opengl extension wrangler library. Webpage, 5 2006. `http://glew.sourceforge.net/`, visited 2006-06-05.

[GP03] Guennebaud G., Paulin M.: Efficient screen space approach for Hardware Accelerated Surfel Rendering . In *Vision, Modeling and Visualization*

, Munich (19-21 novembre 2003), IEEE Signal Processing Society, pp. 485–495.

[Höf06] Höfler M.: Real-time visualization of unstructure volumetric cfd data sets on gpus. In *Central European Seminar on Computer Graphics 2006* (2006).

[KB04] Kobbelt L., Botsch M.: A survey of point-based techniques in computer graphics. In *Computer & Graphics* (2004), vol. 28, pp. 801–814.

[KK05] Kawata H., Kanai T.: Direct point rendering on gpu. In *ISVC* (2005), pp. 587–594.

[Kri03] Krivanek J.: *Representing and Rendering Surfaces with Points.* Tech. Rep. DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague, 2003.

[KV03] Kalaiah A., Varshney A.: Modeling and rendering of points with local geometry. *IEEE Transactions on Visualization and Computer Graphics 9*, 1 (2003), 30–42.

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM Press, pp. 163–169.

[Lev88] Levoy M.: Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37.

[Mat06] MathWorld: Voronoi diagram – from mathworld. Webpage, 8 2006. `http://mathworld.wolfram.com/VoronoiDiagram.html`, visited 2006-06-08.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 154–159.

[Nor06] Norén J.: *Real-time Rendering of Granular Materials.* Master's thesis, Umeå University, Sweden, 2006. UMNAD 631/2006.

[Ope06] OpenGL.org: Opengl - the industry standard for high performance graphics. Webpage, 5 2006. http://www.opengl.org/, visited 2006-06-05.

[Pas04] Pascucci V.: Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym)* (2004), pp. 293–300.

[Per85] Perlin K.: An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM Press, pp. 287–296.

[Poi06] PointShop3D: Pointshop3d. Webpage, 13 2006. `http://graphics.ethz.ch/pointshop3D/`, visited 2006-06-13.

[PZvBG00] Pfister H., Zwicker M., van Baar J., Gross M.: Surfels: Surface elements as rendering primitives. In *Siggraph 2000, Computer Graphics Proceedings* (2000), Akeley K., (Ed.), ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 335–342.

[Räs02] Räsänen J.: *Surface Splatting: Theory, Extensions and Implementation.* Master's thesis, Helsinki University of Technology, 2002.

[RPZ02] Ren L., Pfister H., Zwicker M.: Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum (Eurographics 2002)* (Sept. 2002), vol. 21, pp. 461–470.

[SAC*99] Stora D., Agliati P.-O., Cani M.-P., Neyret F., Gascuel J.-D.: Animating lava flows. In *Graphics Interface* (Jun 1999), pp. 203–210.

[SJP05] Shiue L.-J., Jones I., Peters J.: A realtime gpu subdivision kernel. *ACM Trans. Graph. 24*, 3 (2005), 1010–1015.

[TCH05] Talton J. O., Carr N. A., Hart J. C.: Voronoi rasterization of sparse point sets. In *Proceedings of the Second Annual Eurographics Symposium on Point-Based Graphics* (June 2005), Pauly M., Zwicker M., (Eds.).

[Ves04] Vesterlund M.: *Simulation and Rendering of a Viscous Fluid using Smoothed Particle Hydrodynamics.* Master's thesis, Umeå University, Sweden, 2004. UMNAD 537/2004.

[ZB05] Zhu Y., Bridson R.: Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (2005), 965–972.

[ZPvBG01] Zwicker M., Pfister H., van Baar J., Gross M.: Surface splatting. In *SIGGRAPH 2001, Computer Graphics Proceedings* (2001), Fiume E., (Ed.), ACM Press / ACM SIGGRAPH, pp. 371–378.

[ZRB*04] Zwicker M., Räsänen J., Botsch M., Dachsbacher C., Pauly M.: Perspective accurate splatting. In *GI '04: Proceedings of the 2004 conference on Graphics interface* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004), Canadian Human-Computer Communications Society, pp. 247–254.