Dynamic volume deformation using surfels

Olafur Thor Gunnarsson

A dissertation submitted in partial fulfilment of the requirement for the award of the degree of MSc in Computer Game Technology

University of Abertay Dundee

School of Computing and Creative Technologies

October 2010

**University of Abertay Dundee**

Permission to copy

Author: Olafur Thor Gunnarsson

Title: Dynamic volume deformation using surfels

Qualification: MSc Computer Games Technology

Year: 2010

1. I certify that the above mentioned project is my original work
2. I agree that this dissertation may be reproduced, stored or transmitted, in any form and by any means without the written consent of the undersigned.

Signature …………………….........................

Date …………………………..

**Abstract**

**Acknowledgements**

To my beautiful wife and son.

For their love and support throughout my studies.

**Table of Contents**

[1 Introduction 1](#_Toc271483771)

[1.1 Project aim 2](#_Toc271483772)

[2 Literature Review 3](#_Toc271483773)

[2.1 Volume exterior 3](#_Toc271483774)

[2.1.1 Voxels 3](#_Toc271483775)

[2.1.2 Tetrahedrons 5](#_Toc271483776)

[2.1.3 Surfels 5](#_Toc271483777)

[2.2 Volume interior 8](#_Toc271483778)

[2.2.1 Mass-spring system 8](#_Toc271483779)

[2.2.2 Phyxels 9](#_Toc271483780)

[2.3 Continuum Mechanics 10](#_Toc271483781)

[3 Methodology 13](#_Toc271483782)

[3.1 Surfels 13](#_Toc271483783)

[3.2 Surfel edges 15](#_Toc271483784)

[3.3 Vertex buffer grid 16](#_Toc271483785)

[3.4 Physics 17](#_Toc271483786)

[3.5 Phyxels 18](#_Toc271483787)

[3.6 Surfel resampling 19](#_Toc271483788)

[3.7 Volumetric object setup 20](#_Toc271483789)

[4 Discussion 22](#_Toc271483790)

[5 Conclusion 23](#_Toc271483791)

[6 Future work 24](#_Toc271483792)

[7 Appendices 25](#_Toc271483793)

[7.1 Appendix A – Mass spring systems 25](#_Toc271483794)

[7.2 Appendix B – Material properties of various objects 27](#_Toc271483795)

[7.3 Appendix C – Grid 28](#_Toc271483796)

[8 References 30](#_Toc271483797)

[9 Image references 33](#_Toc271483798)

[10 Bibliography 34](#_Toc271483799)

**Table of Tables**

[Table 1 - Running times of the marching cubes algorithm at various resolutions. 4](#_Toc271483767)

[Table 2 - Object properties 20](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271483768)

[Table 3 - Surface properties 20](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271483769)

[Table 4 – Material properties of various real world objects 27](#_Toc271483770)

**Table of Pseudo Codes**

[Pseudo code 1- Code for phyxel force propagation 12](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271485856)

[Pseudo code 2 - Simple mass-spring system (Müller et al. 2008) 25](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271485857)

[Pseudo code 3 - Setup a 3D spatial array 28](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271485858)

[Pseudo code 4 - Get index of position function 28](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271485859)

[Pseudo code 5 – Get position of index function 29](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271485860)

**Table of figures**

[Figure 1 - Fallout 3 mini nuclear weapon launcher (Sigger 2008) 1](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488377)

[Figure 2 - 15 Cube triangulations (Lorensen and Cline 1987, p165) 3](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488378)

[Figure 3 - Voxel sphere of resolution 20, 40 and 80 respectively (Author’s student project) 4](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488379)

[Figure 4 - Tetrahedral mesh. Left: Only external triangles shown. Right: Internal structure shown (O'Brien 2000, p.31) 5](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488380)

[Figure 5 –A black surfel 5](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488381)

[Figure 6 - Elliptical surfels covering a smooth 3D surface. The surfels draw over the adjacent surfels to cover up holes (Pajarola, Sainz And Guidotti 2004, p 599) 6](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488382)

[Figure 7 - 3D model of Charlemagne (600.000 points) with 2.000, 10.000, 70.000 and 600.000 surfels respectively (Gross and Pfister 2007, p 133) 6](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488383)

[Figure 8 - Max Planck represented with surfels and phyxels. Two leftmost heads are undeformed. The four right heads are deformed elastically, plastically, melted and solidified (respectively) (Müller et al. 2004, p 356) 9](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488384)

[Figure 9 - Hooke's law (Gross and Pfister 2007, p343) 10](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488385)

[Figure 10 - Object's undeformed state (left) and deformed state (right) (Gross and Pfister 2007, p345) 11](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488386)

[Figure 11 - Step 1 of the drawing process 14](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488387)

[Figure 12 - Step 2 of the drawing process 14](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488388)

[Figure 13 - Surfel draw process 14](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488389)

[Figure 14 - Surfel Clip planes 15](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488390)

[Figure 15 - Step 1 of the drawing process for an edge with a clipping plane (0, 1, 1) 16](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488391)

[Figure 16 - Step 2 of the drawing process for an edge with a clipping plane (0, 1, 1) 16](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488392)

[Figure 17 - Surfel wall. Left image - Surfel radius scaled down by 50%. Right image- Surfel radius not scaled down 16](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488393)

[Figure 18 - Phyxels distributed by phyxels (Pauly et al. 2005, p. 961) 18](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488394)

[Figure 19 - Quadtree. Thin lines represent surfels and thick lines represent the quadtree nodes 18](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488395)

[Figure 20 - Force applied to a surfel. Surfels are displayed as black lines and phyxels are displayed as green dots. 19](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488396)

[Figure 21 - Jordan Curve Theorem. The green point is inside the polygon while the red point is outside of the polygon 19](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488397)

[Figure 22 - Example of neighbour selection (a), surfel polygon creation (b) and surfel resampling (c) (Guennebaud, Barthe and Paulin 2004, p 830) 20](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488398)

[Figure 23 - Grid 29](file:///C:\Dropbox\My%20Dropbox\Skoli\Programming\Masters\Dissertation%20-%20Olafur%20Thor%20Gunnarsson%20-%200900128.docx#_Toc271488399)

# Introduction

Computer game realism has improved tremendously since the days of Doom and Wolfenstein 3D. Bill Gates said in a promotional video for Windows ’95 as he was superimposed into Doom 2: “These games are getting really realistic” (Matthewandrewtaylor 2006). Fifteen years later this statement is still true, games are getting really realistic. But what is video game realism? Is it the visual quality of the game world, or is it the behaviour of the world as the player expects it to be? Video game realism is a combination of the two, it needs to look well and behave properly. There is no reason for creating the most real looking video game when there is no logic to the world behaviour.

The game world behaviour has many layers, from physics to night and day cycles. The behaviour that will be covered in this project is the volumetric deformation of walls. This feature is more often than not excluded from modern big budget games. Fallout 3 (Bethesda Game Studios 2008) is a perfect example of this discrepancy. In this game the player is given a rocket launcher that fires mini nuclear weapons. While this weapon works wonders on killing mutants and any other computer generated villain, it does no damage to structures. In a computer generated world that is so rich of interesting characters and beautiful graphics, this omission makes the game world feel like its lacking something. The behaviour of the game world therefore plays an important role in maintaining the user immersion.



Figure - Fallout 3 mini nuclear weapon launcher (Sigger 2008)

A few reasons exist why this is not incorporated into every 3D realistic game. The first one is because static environments simplify the game development by a large margin. In a game world where structures are static, a single wall’s vertices can be pre-calculated and placed into a vertex buffer that will not change throughout the game. This vertex buffer is in turn used when the wall needs to be drawn. If the wall is dynamic and destructible there are many systems that need to be updated. First of all there needs to be an internal mechanism that controls how the wall deforms, which needs to be updated every time a force acts on the wall. In addition to the interior mechanism, the exterior needs updating so the user can see the wall deform and since the exterior changes, the collision detection system needs to be updated, so collisions continue to be realistic. In addition to these steps, the deformation needs to be realistic, which is something which can be hard to do.

## Project aim

This project will aim to find a realistic and yet efficient method of deforming 3D volumes dynamically in video games, such as metal walls. The methods used for the exterior need to be dynamic and be easily regenerated, while the methods for the interior mechanism will need to resemble realistic deformations.

# Literature Review

This chapter is dedicated to the immense research and knowledge that exists in the field of volumetric deformation. Each method will be summarized and discussed with special focus on this project.

## Volume exterior

This sub-chapter will cover the various mesh-free methods for surface visualization.

### Voxels

The Marching Cubes algorithm was introduced in 1987 by Lorensen and Cline for 3D visualization of medical data, such as magnetic resonance (MR) and computed tomography (CT) scans. This algorithm takes eight voxels (points) that make up a cube and checks if the eight points are inside or outside of the model. Based on the states of the points, the algorithm approximates the surface by placing triangles inside the cube. Since each point can be in two states the triangulation possibilities are 28 = 256. Through intuitive thinking, Lorensen and Cline reduced the number of possible triangulation from 256 down to 15 which can be seen in Figure 2. This reduction comes from the use of inverses and rotation. As can be seen from case 0 no triangulation occurs when all or none of the points are defined as “inside”. Alternatively, a single triangle is inserted when a single point is defined as “inside” or “outside” as case 1 depicts. When a single cube has been triangulated, the algorithm marches to the next cube (Lorensen and Cline 1987).

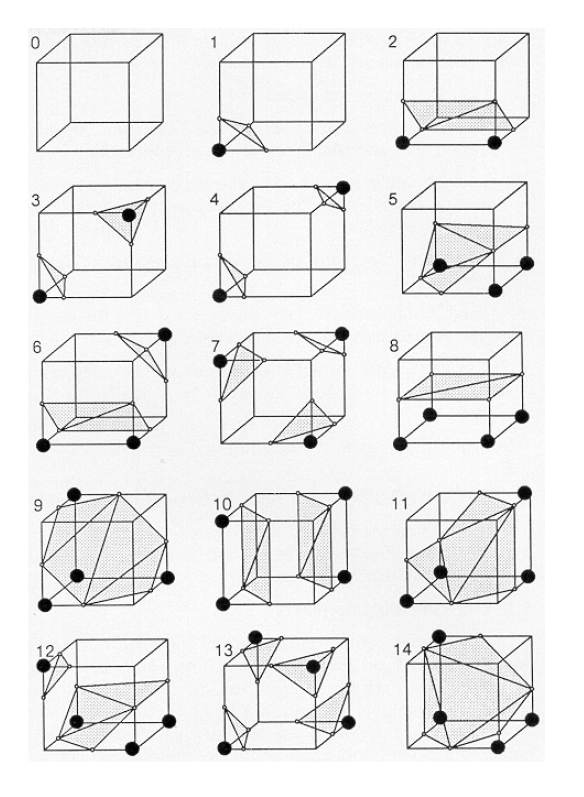


Figure - 15 Cube triangulations (Lorensen and Cline 1987, p165)

However, this method does have its downside because the point set needs to be dense enough so the volume does not look triangulated. In Figure 3 three voxel spheres were created using the marching cube algorithm. These spheres are all of equal size, but vary in their resolution and voxel size. The leftmost sphere is 20 voxels in height, depth and width and has a voxel size of 1. This results in a very course representation of the sphere. When the resolution becomes higher than 40, the voxelized object starts to resemble a sphere. At this resolution the sample point count is 64000. As a reference the sphere of resolution 80 has a sample point count of 512000. It is therefore apparent that a detailed model will need a lot of sample points. Because of this reason, this method was not chosen. Also when the volume is deformed, the whole point set needs to be reconstructed by running the marching cube algorithm again. If the point set is very large, the marching cube algorithm takes a while to finish. This is cooperated by a test done by the author on the author’s student project (Figure 3). The algorithm was run 10 times at various voxel resolutions and the mean time calculated, shown in Table 1:



Figure - Voxel sphere of resolution 20, 40 and 80 respectively (Author’s student project)

|  |  |  |
| --- | --- | --- |
| Voxel resolution | Point count | Time to run Marching Cubes (in seconds) |
| 20x20x20 | 8000 | 0.1337 |
| 40x40x40 | 64000 | 0.9319 |
| 80x80x80 | 512000 | 7.3937 |

Table - Running times of the marching cubes algorithm at various resolutions.

### Tetrahedrons

Just like triangles can represent any surface, tetrahedrons can represent any volume (O’Brien 2000). A tetrahedron is defined as four vertices and four triangles, where three triangles meet at each vertex (Kunkel 2004). The reason why tetrahedrons are useful when it comes to volumetric deformation is because tetrahedrons provide both the surface representation as well as the volumetric representation. The triangles of the tetrahedron that are on the boundary of the mesh are drawn to display the surface of the mesh. The tetrahedron’s vertices can be used as the volume’s physical elements, where the edges between the vertices can be used to simulate the volumes internal structure. The single most importance of tetrahedrons, when it comes to deformation, is the fact that when a tetrahedron is split by any arbitrary plane, the resulting pieces can be decomposed into more tetrahedrons (O’Brien 2000). This plays an important part in deformation since it guarantees a water tight structure. Tetrahedrons were considered as a surface and volume representation for this project, but because of the fact that a lot of tetrahedrons are needed to produce realistic effects, this method was not chosen.

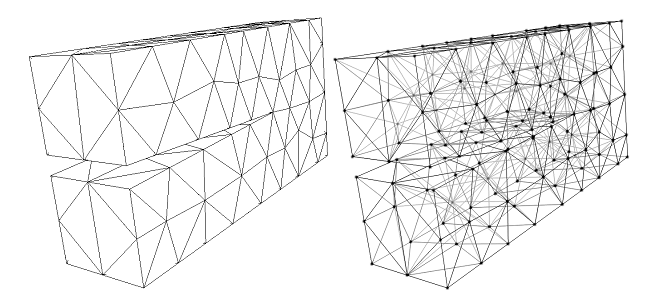


Figure - Tetrahedral mesh. Left: Only external triangles shown. Right: Internal structure shown (O'Brien 2000, p.31)

### Surfels

Over the last decade, research on point based graphics has been ever increasing. Surface elements (surfels) have become very popular when it comes to point based graphics, mainly because of their efficiency to approximate surface (Gross and Pfister 2007). A surfel is defined as a point with a normal and two tangent axes that define an ellipsoidal plate. This plate can in addition contain surface information, such as texture coordinates or colour. Figure 5 shows an image of a surfel with normal **N**, two tangents **t1** and **t2** and position **P**. The tangents are tangent to each other as well as to the normal.

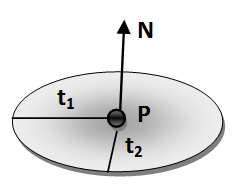


Figure –A black surfel

A surfel based surface is created by splatting surfels onto an approximated surface based on a set of input points. If the point set is sufficiently dense and the surfels are correctly set up, the surfel surface can resemble the expected model (Figure 6). It is the author’s opinion that this method is more efficient than using voxels since only a set of exterior points is used to approximate the surface. However, this method does have its drawbacks. Firstly this method has considerable overdraw, because of the way surfels are used to create a watertight surface. It is apparent from Figure 7 that surfel based rendering techniques do draw over pixels that have been drawn already. Secondly, edges and corners pose a problem with surfels, since they are elliptical plates. One solution to the edge problem is to detect intersecting surfels and divide into smaller surfels until the intersection is below a certain threshold (Adams and Dutré, 2003). This method does however increase the number of surfels by a large margin and might result in visual defects if the structure is looked at closely. Another solution has been to use an algorithm that goes through the point set and reconstructs the 3D object based on these points. This solution is very popular for systems that receive a large set of points obtained by a laser scanner, but is beyond the scope of this thesis. Those interested in such reconstruction algorithms are pointed to a paper by Öztireli, Guennebaud and Gross (2009).

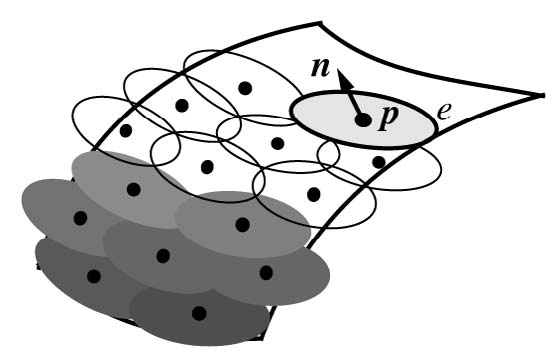


Figure - Elliptical surfels covering a smooth 3D surface. The surfels draw over the adjacent surfels to cover up holes (Pajarola, Sainz And Guidotti 2004, p 599)



Figure - 3D model of Charlemagne (600.000 points) with 2.000, 10.000, 70.000 and 600.000 surfels respectively (Gross and Pfister 2007, p 133)

The traditional method of splatting surfels, as introduced by Pfister et al. (2000), rendered an image of the surfel object to a texture of a certain size. The frame-rate this technique accomplished was relatively low, since it used six steps to reconstruct the textures and was created 10 years ago. Over the last few years, advantages in graphic processing unit (GPU) power and features have been utilized to speed up the surfel splatting method. Botch et al. (2005) displayed a model in a 512x512 window, generated by 137.000 splats, at 20.1 frames per second (fps) with Phong shading and shadow mapping, using their two pass algorithm. Ochotta, Hiler and Saupe (2006) doubled Botch’s et al. frame’rate by using a single pass algorithm to render a 138.654 splat balljoint at 45 fps, with Phong shading.

Because surfels do need a reasonably small point set size to create detailed models and due to the fact that this field is relatively young, this method was chosen as the project’s surface representation. However, for this method to work in a computer game, some simplifications will need to be done.

## Volume interior

For volumetric deformation to be applied to a volume, the volume needs to be converted into a form that is easy to use in computer graphics. There exist a number of different methods to represent how force is distributed through the volume, which will be covered in this sub-chapter.

### Mass-spring system

The simplest way to simulate deformable objects in computer graphics is to use a mass-spring based system. It is set up by using particles with masses that are connected by springs (Müller et al. 2008). In the simplest mass-spring systems the points have only position and velocity attributes. It can also be extended by adding acceleration to the points. With these attributes it is possible to add forces to the system and distribute them through the model. Appendix A shows a short pseudo code to emphasize the simplicity of mass-spring systems.

This method does however have its drawbacks. The first problem is that the deformable behaviour of the object is very dependent on how the mass-spring system is set up. For example if a wall is implemented with a mass spring system, the mass of the points as well as the springs stiffness and damping constants need to be correctly set up so the correct behaviour is simulated. It is therefore inevitable to encounter some variable fiddling. The final drawback is due to this system being only stable for relatively small time steps, from 0.1ms – 1ms (Müller et al. 2008).

### Phyxels

Another particle based volumetric representation is the physical element (phyxels) systems. Similar to mass spring particles, phyxels contain body location (***x***) and force (***f***). Additionally, phyxels contain attributes that model density (), deformation (***u***), volume (*v*), strain () and stress () (Müller et al. 2004). With these additional attributes the model’s material can be simulated relatively realistically using Hook’s law (see chapter 2.3). Figure 8 shows a modelled head of the German physicist Max Planck, simulated by surfels and phyxels. The leftmost image depicts the Max Planck model, where the surfels are scaled down so the internal structure is showing. In this image the phyxels are coloured yellow and the surfels are blue.



Figure - Max Planck represented with surfels and phyxels. Two leftmost heads are undeformed. The four right heads are deformed elastically, plastically, melted and solidified (respectively) (Müller et al. 2004, p 356)

Instead of springs between phyxels, the phyxels incorporate a weighing function () which is a scalar-kernel and is radially symmetric. To implement this function the phyxels need to have a support radius, that is the phyxel’s area of effect. Müller et al. (2004) defined the weighing function as , where *r* is the distance from the position of the phyxel and *h* is the phyxel’s support radius. This weighing function can be used to calculate how much of a force is distributed from one phyxel to its neighbours. Another functionality of the weighing function is to calculate the phyxel’s density , where (Gross and Pfister 2008). This volumetric representation was chosen as the project’s representation because this method does provide a reasonably realistic representation of volumetric deformations. Because of this, the phyxel method will be used throughout this thesis as the interior representation of the volumes.

## Continuum Mechanics

In continuum mechanics a single law, created by Robert Hooke in the 17th century, has been extensively used to represent deformation of 3D computer generated objects. Hooke’s law can be used on so called Hookean materials, which are materials where stress and strain are linearly related (Müller 2008). Figure 9 shows a beam with a cross section that has an area *A*. Force ***f****n* is applied perpendicular to the area, resulting in an elongation of the beam by ∆*l*. The beam’s stress  is defined as and therefore has the unit . The strain  however has no unit as it is defined as . The strain to stress ratio does depend on *E*, called the Young’s modulus which represents the elastic stiffness of the material. Various Young’s modulus’s for a variety of real world materials can be seen in Table 1 in Appendix B.



Figure - Hooke's law (Gross and Pfister 2007, p343)

The object in Figure 9 is only deformable in one dimension, along the force and therefore its stress and strain variables are one dimensional (1D). However if the object is deformable in 3D, these variables get more complicated. According to Müller et al. (2008) and Gross and Pfister (2007), 3D materials that behave equally in every direction can express the stress and strain variables as 3x3 symmetric matrices (or tensors):

|  |  |  |  |
| --- | --- | --- | --- |
|  | (.) |  | (.) |

Since the stress and strain tensors are symmetric, they only contain 6 individual variables. Because of this, the stiffness modulus can be written as a 6x6 symmetric matrix:

|  |  |
| --- | --- |
|  | (.) |

In equation (1.3) *E* is the Young’s modulus and *v* called Poisson’s ration, which describes how much of the volume is conserved within the material and is limited between 0 and .

When a deformation object is represented in computer graphics, it is typically defined by two states: its undeformed state and by a set of parameters that define its deformed state. As forces are applied to phyxels, the material coordinate (***x*** *=* [*x, y, z*]T) is moved to its world coordinates (***p****(****x****)**=* [*x’, y’, z’*]T), both of which are defined as vector fields. The deformation of the object can be defined by a displacement field (Müller 2008). This displacement can be seen in Figure 10 where the undeformed position is moved to its deformed position by applying the material properties.

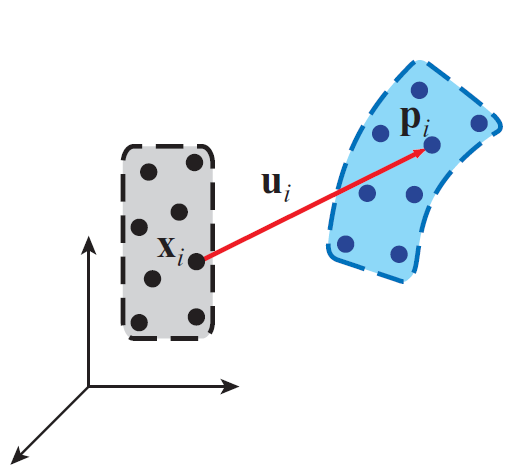


Figure - Object's undeformed state (left) and deformed state (right) (Gross and Pfister 2007, p345)

When updating the stress and strain a few things need to be calculated. Firstly, according to Pauly et al. 2005 and Müller 2008, a moment matrix (also known as a system matrix) is used to calculate the displacement derivative of each phyxel. This moment matrix is defined in equation 2.4 and the derivative of the displacement matrix is defined in equation 2.5:

|  |  |
| --- | --- |
|  | (2.4) |
|  | (2.5) |

This displacement field is used to update the strain, which can in turn be used to update the stress. Each spatial derivative of the displacement field is calculated by:

|  |  |
| --- | --- |
|  | (2.6) |
|  | (2.7) |
|  | (2.8) |

where *u* is defined as the displacement along the x-axis, *v* is the displacement along the y-axis and *w* is along the z-axis. Using the displacement derivative the strain can be updated with the Green’s nonlinear strain tensor:

|  |  |
| --- | --- |
|  | (2.9) |

Finally, a single matrix (***B***) is used to distribute internal forces throughout the volume:

|  |  |
| --- | --- |
|  | (2.10) |

The following pseudo code shows how a complete system of phyxels is initialized and updated (Gross Pfister 2007):

**forall** phyxel i

initialize i,i = 0, = 0, , i, ;

calculate using equation (2.4);

**endfor**

**loop**

**forall** phyxels i **do endfor**

**forall** phyxels i **do**

calculate and using equations (2.6, 2.7 and 2.8);

calculate using equation (2.9);

calculate using equation (2.3);

calculate using equation (2.10);

**forall** neighbouring phyxels j

;

**endfor**

**endfor**

**forall** phyxels i

; **;**

**endfor**

render system at ;

**endloop**

Pseudo code - Code for phyxel force propagation

This volumetric representation was chosen as the project’s representation because of the realistic simulation of deformable objects. The reason that this algorithm has been used and is tried and tested by many also played a big part in the choice (Pauly et al. 2005, Müller et al. 2008 and O’Brien 2000 to name a few).

# Methodology

This chapter will cover the creation of the program developed for this project. Each important aspect will be delved into deeply with respect to the aims of the project (create aim’s) ☺.

## Surfels

As mentioned earlier some simplifications need to be done to the surfel technique in order for them to be useful in computer games. With DirectX 10, a new shader was introduced as an addition to the vertex and pixel shaders (VS and PS respectively), called a geometry shader (GS). The purpose of this shader is to change rendering primitives in ways needed by the program (Blythe 2006). This new shader does bode well for the surfel technique, since it can transform the surfel primitive into two triangles that is easily be rendered by the GPU. The project’s surfel contains a position, normal and two tangent vectors as depicted by Figure 5 in chapter 2.1.3. Additionally, surface texture coordinates () and a delta texture coordinates (), which are two dimensional vectors, are added to the surfels. The delta texture coordinate is used for the calculation of each point’s texture coordinate and is based upon the tangents of the surfel and the size of the surface. Both of the components in represent how far the surface goes along the tangent axes.

The surfel rendering process is twofold. Firstly, using the geometry shader, a surfel is sent to the graphics hardware where a quad is created by the GS (Figure 11). The quads created by the GS are outputted to a vertex stream as triangle strips. This vertex stream is in turn used by the second step to draw the quads (Figure 12). Whenever the surfels change in any way, the vertex stream containing the quads is rebuilt as shown in Figure 13. There are two textures loaded to the shader when step two is performed; one texture is used as the surface texture, while the other is used as an alpha channel texture for the splats. The splat texture is white with alpha values . The surface texture of the quad is multiplied by this alpha texture to create the surfel disk (Figure 12).

The purpose of the first step of the drawing process is to correctly setup the quad so it can be represented as a splat. Firstly the points positions are calculated based upon the surfel position and the tangent axes, for example the position of the point in Figure 11 is , where and are the positions of and respectively. Each point’s surface texture coordinate is calculated in the same way as its position is. The only difference is that is used instead of the tangents and instead of the position. Using again from Figure 11, its surface texture coordinate is calculated by . Since the quads are represented as elliptical splats, the quad need texture coordinates for the alpha texture, . Because the alpha texture is circular, the coordinates are always the same for every surfel that is not an edge (covered in the next chapter). These coordinates are , , and for ,, and respectively. The normal of the quad is always the same as the normal of the surfel’s.

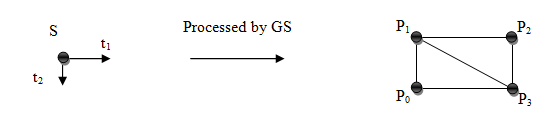


Figure - Step 1 of the drawing process

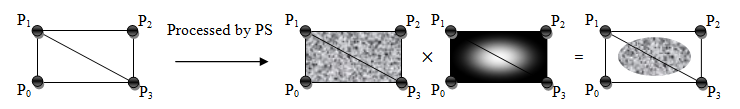


Figure - Step 2 of the drawing process

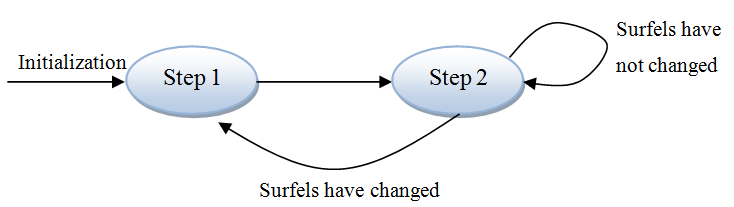


Figure - Surfel draw process

## Surfel edges

One of the troubles with surfels is how to render edges and corners. The method used in this project is a very simple one. Each surfel contains a clipping plane aligned along either or both of its axes. The clippling plane is represented as a 3D vector with values depicted in Figure 14:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

Figure - Surfel Clip planes

The calculation of the quad properties is slightly different for edges, since the clipping plane has an effect on the surfel’s position, and. The two steps of the drawing method are therefore slightly different for edges. Firstly the position of the edge is not in the surfel’s centre, it is instead placed on the clipping plane. Due to this change, the texture coordinates are also slightly different. As Figure 16 shows, the calculation of these properties for points and are different from Figure 12. If the surfel’s clipping plane depicts that the surfel’s size is not extended into the negative direction (as is shown in Figure 16), the texture coordinates are not extended in that direction. This is also true for every other clipping plane. Using these clipping planes, a wall and any cubical volume can easily be created. Figure 15 shows a wall created by surfels.

## Vertex buffer grid

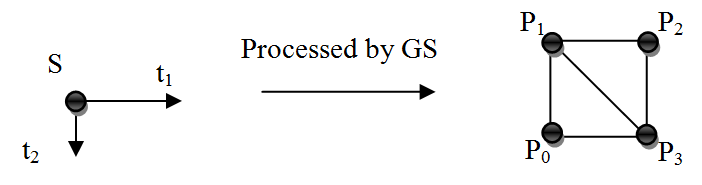


Figure - Step 1 of the drawing process for an edge with a clipping plane (0, 1, 1)

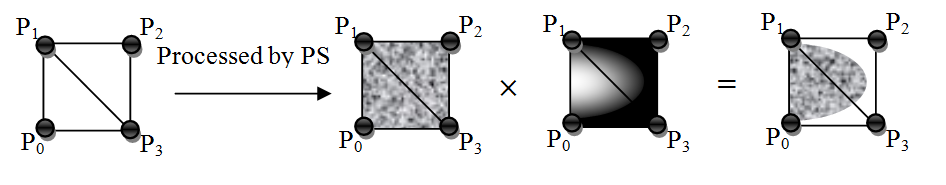


Figure - Step 2 of the drawing process for an edge with a clipping plane (0, 1, 1)

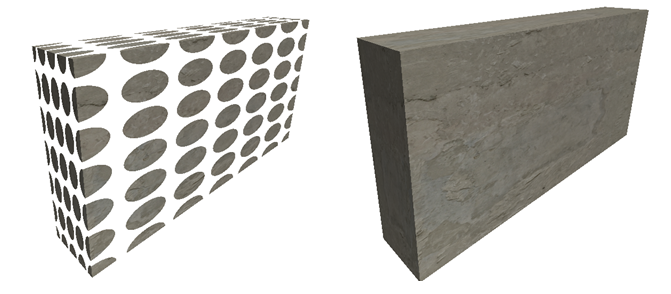


Figure - Surfel wall. Left image - Surfel radius scaled down by 50%. Right image- Surfel radius not scaled down

When a surfel changes in the volume, the vertex buffer used to draw the surface needs to be recreated. If this vertex buffer is large enough, it can take a few milliseconds to recreate, which is unacceptable in computer game times. This is because most game developers try to make their games run at more than 30 frames per second (reference?), which means that each frame cannot take more than 33.333 milliseconds. If a few milliseconds are used to recreate a big vertex buffer, it does not leave many milliseconds to be used in other things.

An easy way to split up an object spatially is to use a 3D grid (covered in Appendix C – Grid). For each object, that is drawn using the surfel technique, a grid is created based on the object’s position and its dimensions. Each surfel added to the object is also added to this grid, where the cell index of the surfel is based on the surfel’s position. When every surfel has been added to the object, the vertex buffers of the cells are created based on the surfels. During every object’s draw call the grid is traversed and each populated cell is drawn. When any surfels get changed, the corresponding grid cell’s vertex buffer has to be recreated.

However, the size of the grid cells needs to be carefully setup based upon the surfel count. The grid cell size cannot be too big because that would result in a vertex buffer that is very large and therefore slow to rebuild. In turn, the grid cell size cannot be too small, because then it would take a while to traverse the grid on every draw call. The size of the grid cells that was decided upon is 2.5. This configuration results in a good frame rate as well as a quick vertex buffer reconstruction time. **DO YOU THINK THIS LAST PARAGRAPH BELONGS IN THE RESULTS CHAPTER?**

## Physics

Since the project’s aim is to model a behaviour that occurs when a deformable object is struck by a projectile or a wrecking ball, there needs to be some physics simulation incorporated into the project. For this purpose the Havok system was chosen. The reason for this choice is that it has been used in many block buster video games for a few years; Fallout 3, Starcraft 2 and Bioshock 2 to name a few (Havok 2010a).

The program features a wrecking ball, projectiles and surfaces which need physical properties. Creating rigid body properties for the wrecking ball and projectiles can be set up relatively easily by following some examples that come with the downloadable Havok SDK (Havok 2010b). However, creating rigid bodies for each surfel is a bit trickier. Since the Havok system uses vertices to create arbitrary rigid bodies the elliptical surfels are hard to represent. This can be alleviated by using the surfel’s quad representation created by the GS. The quad representation of all surfels, within a vertex buffer grid cell, is stored in a vertex buffer that is readable by the central processing unit (CPU). When the surfel rigid bodies are created by the program, the readable vertex buffer is read. Based on the rectangular vertex coordinated of the surfel, a rectangular rigid body is created. Additionally a custom contact listener is created for each surfel to detect collisions between it and the wrecking ball or it and the projectiles.

## Phyxels

To represent the internal mechanism of deformable objects, the phyxel representation was used. The phyxels need to be distributed within the volume so the deformation can be realistically animated. Initially the method proposed by Pauly et al. (2005) was considered. This method uses an octree to distribute the phyxels based on the position of the surfels as shown in Figure 18. A single phyxel is inserted in every populated octree node. However some trouble was encountered with this method when a surfel is added to the octree. With every surfel added to the octree, the four quad points are used as a measure of the surfel’s size. For a surfel to be added to an octree node, all the four points need to be enclosed in the node. This results in the fact that a corner wall cannot use an octree to distribute the phyxels. The reason for this is that there will be at least a single surfel that will only be fully contained in the first level node. This is shown in Figure 19 where a two dimensional (2D) representation of an octree is shown with a corner wall made up of surfels. If the method proposed by Pauly et al. (2005) had been used a phyxel would have been added in the centre of the red node, which would have resulted in some unrealistic effects.

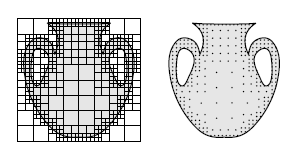


Figure - Phyxels distributed by phyxels (Pauly et al. 2005, p. 961)

A different method was therefore devised using a grid (covered in Appendix C – Grid). Using this method, the phyxels are inserted into the grid cells that are populated by surfels. This results in an equally sampled volume and a very controllable spacing of phyxels. When the phyxel grid is initialized, phyxels are only inserted into cells that include one or more surfels. Therefore more work needs to be done to populate the cells that are empty within the volume. This is done by utilizing a very simple method of ray-tracing, based upon the Jordan Curve Theorem (Cismasu 1997) shown in Figure 20. In this method, a ray is cast from each empty cell in its six directions from the cell (up, down, left, right, forwards and backwards). If the ray intersects an odd number of cells in every direction, then the cell is inside the volume and is populated by a phyxel. The size of the phyxel grid cells needs to be carefully setup to provide a balance between speed and realism. If the phyxel grid cells are too big, the behaviour of the deformable objects becomes unnatural. This is because many surfels will be linked to the same phyxel and if the phyxel is displaced, then all the surfels connected to it will also be displaced. However if the phyxel grid cells are too small, then each surfel has only an effect on a very small percentage of the wall.

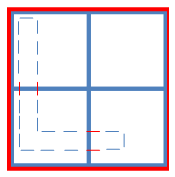


Figure - Quadtree. Thin lines represent surfels and thick lines represent the quadtree nodes

Each surfel that is added to the phyxel grid knows which grid cells it is contained in and therefore knows which phyxels belong to it. This makes the force distribution very simple when the deformable object is deformed. When the custom contact listener detects a collision between a surfel and another rigid body, the force applied by the rigid body is distributed to the surfel’s phyxels. The surfel applies the exterior force to the phyxels it is adjacent to, shown in with red lines. The affected phyxels apply the force to their neighbours as proposed in , shown in with dotted lines.

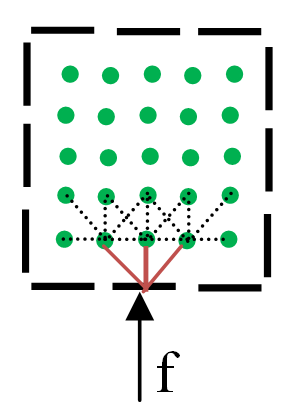


Figure - Force applied to a surfel. Surfels are displayed as black lines and phyxels are displayed as green dots.

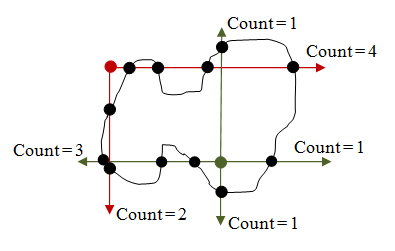


Figure - Jordan Curve Theorem. The green point is inside the polygon while the red point is outside of the polygon

## Surfel resampling

When the surfels get displaced, the surfels need to be resampled. The resampling method used in this project is based on a method devised by Guennebaud, Barthe and Paulin (2004). The first thing needed for this method to be useful is a surfel neighbourhood around the surfels. This neighbourhood of surfel **S** is determined by surfels that are within a certain radius of the **S**. These surfels are projected onto the tangent plane of **S** and sorted by their increasing angle between them and the first neighbour. If two neighbours have an angle that is too small, the neighbour further away is removed (Figure 22a). According to Guennebaud, Barthe and Paulin (2004), has been proven to be a good choice for this smallest angle. Every neighbouring surfel is also a neighbour of **S**. When the neighbouring graph has been created, the surfel polygons can be created. Depending on the neighbour count of surfel S the amount of polygons can vary. A single surfel polygon is created by finding all surfels that are neighbours of each other. For example if surfel **S** has a neighbourhood. The condition needed for the creation of a surfel polygon, then the condition and must be fulfilled.

Guennebaud, Barthe and Paulin (2004) created a method of resampling the polygon based upon how many surfels are included in the surfel polygon. The method used in this project is however slightly different and requires little computational power. A surfel is inserted in the middle of the polygon as well as in the middle of the polygon’s edges (Figure 23). Newly created surfels receive averaged properties from their parents (black dots in Figure 22 and Figure 23). Finally after the refinement procedure, the tangent axes and of the parent surfels are reduced by 50%. However, with this method, a check needs to be performed to prevent surfels being inserted where another surfel already exists. This check is simply done by checking the distance between two surfels are more than their combined maximum axes length.

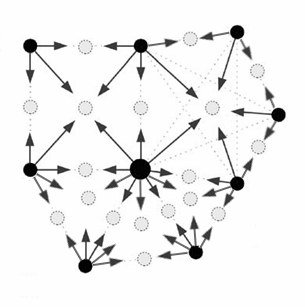


Figure - Updated surfel refinement



Figure - Examples of (a) neighbour selection, (b) surfel polygons creation and (c) surfel resampling (Guennebaud, Barthe and Paulin 2004, p 830)

## Volumetric object setup

In order for the object creation to be dynamic, a XML file was created. This XML file contains material properties, lighting information and object dimensions. During initialization the XML file is parsed and volumetric objects created. While the program is running, the XML file can be changed and reloaded. This results in a program that can be tweaked easily. The object properties that can be changed are shown in . In addition with these properties, the XML file contains a single “volume” XML node, which can contain as many “surface” nodes as needed to represent the object. Table 3 shows the properties of each surface node where the surface’s dimensions can be altered.

|  |  |  |  |
| --- | --- | --- | --- |
| Property name | Description | Variables | Type |
| texture | Relative path and texture name |  | String |
| deformable | Is the volume deformable |  | True/False |
| youngs\_modulus | The material’s Young’s modulus |  | Number |
| poisson\_ratio | The material’s Poisson’s ratio |  | Number |
| toughness | The toughness of the material |  | Number |
| vertex\_grid\_size | The size of each vertex grid cell |  | Number |
| phyxel\_grid\_size | The size of each phyxel grid cell |  | Number |
| lighting | The lighting information for the material | sigma, rho | Numbers |
| worldtranslation | The translation of the object | x, y, z | Numbers |
| worldscale | The scaling of the object | x, y, z | Numbers |
| worldyawpitchroll | The rotation of the object | x, y, z | Numbers |

Table - Object properties

|  |  |  |  |
| --- | --- | --- | --- |
| Property name | Description | Variables | Type |
| Position | Centre of the surface | x, y, z | Numbers |
| normal | Normal of the surface | x, y, z | Numbers |
| majorAxis | Major axis of the surfels that make up the surface | x, y, z | Numbers |
| minorAxis | Minor axis of the surfels that make up the surface | x, y, z | Numbers |
| count | Determines how many surfels make up the surface | u, v | Numbers |

Table - Surface properties

Since this project is only concerned with cubical walls, every surface that is created with this method is rectangular. This is done by adding *u* surfels along the *majorAxis* (one of the tangent axes) and *v* surfels along the *minorAxis* (the other tangent axes) and since these axes only define half of each surfel this *u* and *v* surfels are added along the negative axes. Surfels containing clipping planes are added to the edges of the surfaces, to create the sharp corners of the volume.

# Discussion

# Conclusion

# Future work

# Appendices

## Appendix A – Mass spring systems

Pseudo code 1 shows implementation for a simple mass-spring system:

// initialization

**forall** particles i

initialize **x**i, **v**i and mi

**endfor**

// simulation loop

**loop**

**forall** particles i

(Eq. A.5)

**endfor**

**forall** particles i

(Eq. A.7)

(Eq. A.8)

**endfor**

display the system every nth time

**endloop**

Pseudo code - Simple mass-spring system (Müller et al. 2008)

Before the simulation begins each particle is initialized with a position (**x**), velocity (**v**) and mass (*m*). During each simulation step the gravity force (***f****g*) and collision force (***f****coll*) are accumulated to the force of the particle. The particle’s spring forces are calculated from its adjacent particles (S) in equations A.1 and A.2.

(A.1)

(A.2)

Here attributes with the *ith* and *jth* indices represent adjacent particles *i* and *j*, *k****­****s* stands for the stiffness of the spring and *l0* represents the initial length between the two particles. Similarly the damping force between the two particles needs to be calculated using equations A.3 and A.4:

where *k*d represents the spring’s damping constant. If equations A.1 and A.3 are combined, the complete spring force is:

(A.5)

(A.3)

(A.4)

Finally when all forces have been calculated for each particle and its adjacent particles, the particles need to be moved. By combining Newton’s second law of motion and two kinematic equations, the equations for updating the particle’s velocity and position are constructed:

(A.6)

(A.7)

(A.8)

## Appendix B – Material properties of various objects

|  |  |  |  |
| --- | --- | --- | --- |
| Material | Young’s Modulus (GPa) | Poisson’s ratio | Density (kg/m3) |
| Aluminium (MatWeb 2010a) | 68 | 0.360 | 2700 |
| Concrete (The Engineering ToolBox [No Date]) | 14 – 41 | 0.20 – 0.21 | 2240 - 2400 |
| Steel (MatWeb 2010b) | 68.9 – 317 | 0.220 – 0.346 | 190 - 9010 |
| Titanium (MatWeb 2010c) | 116 | 0.340 | 4500 |

Table – Material properties of various real world objects

## Appendix C – Grid

The grid created for this project can be thought of as a big bounding box that is cut into equally sized cells. In this project it is used for the vertex buffer grid and the phyxel grid. To set up the grid it needs to receive the minimum and maximum object coordinates, its position and the grid cell size. The size of the grid is determined by the grid cell size and its minimum and maximum coordinates. This calculation is demonstrated in Pseudo code 2:

Vector3D halfDimensions = (maximum– minimum) / 2;

float halfGridCellSize = gridCellSize \* 0.5;

Grid grid = Grid(halfDimensions.x / halfGridCellSize +1, halfCellCount.y / halfGridCellSize +1, halfCellCount.z / halfGridCellSize +1);

When a 3D position is inserted into the grid, the index needs to be calculated. For example the index of the Minimum point in Figure 20 is (0, 0, 0), whereas the index of the Position point is (3, 2, 0) or (3, 2, 1) depending on the implementation. Pseudo code 3 shows how an index is calculated from the 3D position and pseudo code 4 shows how the position of an index is calculated:

Pseudo code - Setup a 3D spatial array

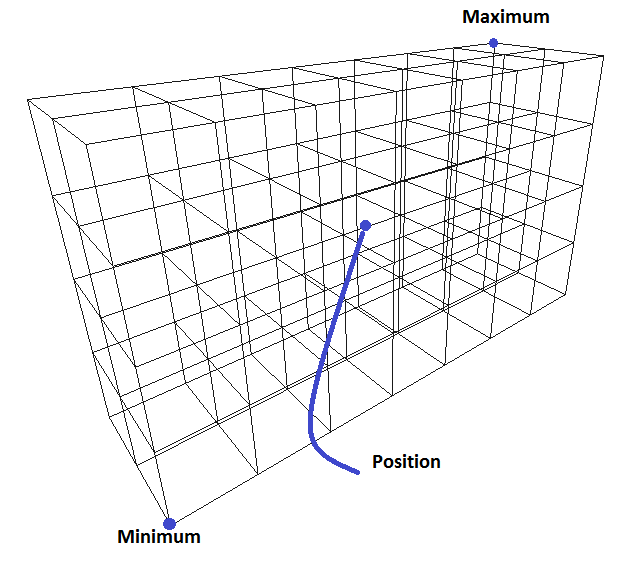


Figure - Grid

Pseudo code - Get index of position function

Vector3D **function** GetIndexOfPosition( Vector3D pos){

Vector3D translation =;

Matrix invWorld = ;

Vector3D index = ;

return index;

}

In the position needs to be translated to the grid’s local space. This is done by translating the position to the negative minimum coordinate, which is equal to the inverse of translating to the positive minimum coordinate. At this point, the input position is in the grid’s local space and therefore the input position is between the maximum and minimum coordinates. The index can therefore be determined by dividing the result by the grid cell size.

Pseudo code – Get position of index function

Vector3D **function** GetPositionOfIndex( Vector3D index){

Vector3D halfCellCount = Vector3D(grid.width, grid.height, grid.depth) \* 0.5;

return ;

}

The inverse of this is to find the position of a 3D index. This is done by subtracting the index by half of the cell count of the array. This results in an index between . To get the index position into the local space of the grid it needs to be multiplied by the size of the grid cells. Additionally, to get the middle position of the cell, the half size of the grids needs to be added to the result.

# References

Adams, B. and Dutré, P. 2003. Interactive Boolean operations on surfel-bounded solids*.* In: *International Conference on Computer Graphics and Interactive Techniques, San Diego, 27-31 July*. New York: ACM. pp 651-656. [Online]. Available from: <http://portal.acm.org/citation.cfm?id=1201775.882320> [Accessed 1 August 2010]

Bethesda Game Studios. 2008. *Fallout 3.* [Disk]. Windows XP. Ubisoft.

Blythe, D. 2006. *The Direct3D 10 system*. [Online]. Microsoft Corporation. Available from:

[http://download.microsoft.com/download/f/2/d/f2d5ee2c-b7ba-4cd0-9686-](http://download.microsoft.com/download/f/2/d/f2d5ee2c-b7ba-4cd0-9686-b6508b5479a1/direct3d10_web.pdf)

[b6508b5479a1/direct3d10\_web.pdf](http://download.microsoft.com/download/f/2/d/f2d5ee2c-b7ba-4cd0-9686-b6508b5479a1/direct3d10_web.pdf) [Accessed 8 May 2010]

Botsh et al. 2005. High-quality surface splatting on today’s GPUs. In M. Pauly and M. Zwicker, eds.: *Point-Based Graphics 2005. Eurographics/IEEE VGTC Symposium Proceedings, 20 – 21 June*. pp 17 – 141

Cismasu, O. 1997. *The Jordan Curve Theorem for Polygons*. [Online]. McGill University. Available from: <http://cgm.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Octavian/compgeom.html>. [Accessed 18 June 2010]

Gross, M. and Pfister, H. eds. 2007. *Point based graphics.* Burlington, MA: Morgan Kaufman.

Guennebaud, G., Barthe, L. and Paulin, M. 2004. Dynamic surfel set refinement for high-quality rendering. *Computers and Graphics.* 28(6), pp.827 – 838.

Havok. 2010a. *Available games.* [Online]. Available from: <http://www.havok.com/index.php?page=available-games> [Accessed 4 September 2010]

Havok. 2010b. *Try Havok.* [Online]. Available from: <http://software.intel.com/sites/havok/> [Accessed 8 June 2010]

Lorensen, W.E. and Cline H.E. 1987. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics.* 21(4), pp. 163 – 169

Kunkel, P. 2004a. *The Tetrahedron.* [Online]. Available from: <http://whistleralley.com/polyhedra/tetrahedron.htm> [Accessed 27 August 2010]

Matthewandrewtaylor. 2006. *Bill Gates in Doom 2*. [Online]. YouTube. Available from: <http://www.youtube.com/watch?v=xh0JM7pD4qM> [Accessed 24 August 2010]

MatWeb. 2010a. *Aluminium, Al*.[Online]. Available from: <http://www.matweb.com/search/DataSheet.aspx?MatGUID=0cd1edf33ac145ee93a0aa6fc666c0e0> [Accessed 15 August 2010]

MatWeb. 2010b. *Overview of materials for Stainless Steel*. [Online]. Available from: <http://www.matweb.com/search/DataSheet.aspx?MatGUID=71396e57ff5940b791ece120e4d563e0> [Accessed 15 August 2010]

MatWeb. 2010c. *Titaium, Ti*.[Online]. Available from: <http://www.matweb.com/search/DataSheet.aspx?MatGUID=66a15d609a3f4c829cb6ad08f0dafc01&ckck=1> [Accessed 15 August 2010]

O’Brien, J.F. 2000. *Graphical Modeling and Animation of Fracture*. [M.Phil. thesis]. Georgia Institute of Technology.

O’Brien, J.F., Bargteil, A. W. and Hodgins, J.K. 2002. Graphical modelingand animation of ductile fracture. *ACM Transactiona on Graphics (TOC).* 21(3), pp 291-294.

Ochotta, T., Hiller, S. and Saupe, D. 2006. *Single-Pass High-Quality Splatting.*[Online] Available from: <http://www.inf.uni-konstanz.de/cgip/bib/files/OcHiSa06.pdf> [Accessed 1 July 2010]

Müller, M. et al. 2004. Point based animation of elastic, plastic and melting objects. In: *Proceedings of the 2004 ACM SIGGRAPH, Grenoble 27 – 29 August 2004*. New York: ACM. pp 141-151.

Müller, M. et al. 2008. Real time physics: class notes. *International Conference on Computer Graphics and Interactive Techniques, Los Angeles, 11 – 15 August 2008*. New York: ACM

Nealen, A. et al. 2006. Physically based deformable models in computer graphics. *Computer Graphics Forum.* 25(4), pp. 809 – 836.

Pajarola, R., Sainz, M. And Guidotti, P. 2004. Confetti: Object-Space Point Blending and Splatting. *IEEE transactions on visualization and computer graphics.* 10 (5), pp 598 – 608

Pfister, H., et al. 2000. Surfels: surface elements as rendering primitive. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques.* New York: ACM. pp. 335 – 342. [Online]. Available from: <http://portal.acm.org/citation.cfm?id=344779.344936> [Accessed 21 April 2010]

Pauly, M., et al. 2005. Meshless animation of fracturing solids. *ACM Transactions on Graphics (TOG)*. 24(3), pp. 957 – 964.

The Engineering ToolBox. No Date. *Concrete Properties*. [Online]. Available from: <http://www.engineeringtoolbox.com/concrete-properties-d_1223.html> [Accessed 3 July 2010]

Öztireli, A. C., Gennebaud, G., And Gross. M. 2009. Feature preserving Point set surfaces based on non-linear kernel regression. In: *Proceedings of Eurographics, Munich, 30 March – 3 April.* 28(2), pp. 493-501.

# Image references

Sigger, J. 2008. *No fat man for you!*. [Online image]. Available from: <http://armchairgeneralist.typepad.com/my_weblog/2008/11/no-fat-man-for.html> [Accessed 24 August 2010]

# Bibliography