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

(i) I certify that the above mentioned project is my original work

(ii) 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

In recent years, the realism in computer games has been rising steadily. However there is often a lack of volumetric deformation, such as metal walls buckling from exterior forces.

This project aims to simulate deformable metallic walls realistically and efficiently. Since the walls are deformable, an efficient surface representation had to be created which has a low vertex buffer regeneration time. Additionally, to propagate exterior forces and deforming the volume, a structure for the interior needed to be incorporated.

Each wall created in this project is spatially divided into interior and exterior systems. The exterior system is constructed out of a vertex buffer grid. Each grid's cell contains part of the wall's surface, represented with a relatively new point based rendering technique called surfels. The interior system is created out of a grid of physical elements, called phyxels. The phyxels determine the volume's material properties, as well as how much the volume has been deformed.

When a wall is deformed, its surfels need to be displaced as well as resampled. The resampling procedure inserts new surfels between deformed surfels to maintain the water tightness of the structure.

The results show that surfels are well suited for drawing damaged and undeformed volumes, especially with the inclusion of the vertex buffer grid. While the volume deformation algorithm does not run in real time for volumes with a high surfel count, it does provide a very realistic damage simulation.

Acknowledgements

To my beautiful wife and son,
for their love and support throughout my studies.

Thanks to:
Dr. Euan Dempster for his supervision and proof reading throughout the project's duration under less than optimal circumstances.

Table of Contents

Table of Tables

Table of Pseudo Codes

Table of figures

1 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 where 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 villains, 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.

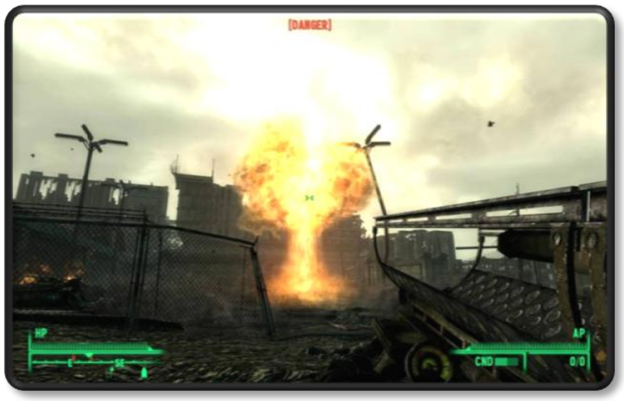

Figure 1 - Fallout 3 mini nuclear weapon launcher (Sigger 2008)

A few reasons exist why this is not incorporated into every three dimensional (3D) 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 forces act on the wall. In addition to the interior mechanism, the exterior needs updating so the user can see the wall deform. In order to continue the realistic collision detection between the wall and any other physics object in the world, the collision detection system needs to be updated. In addition to these steps, the deformation needs to be realistic, which is something which can be hard to do.

1.1 Project aims

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 allow for easily regenerated surfaces, while the methods for the interior mechanism will need to resemble realistic deformations. In addition to these goals, the question of whether or not surfels are viable as a choice for deformable volumes in computer games will be answered.

1.1.1 Objectives

The project can be easily split into three tasks. Firstly is the task of modelling the exterior of the volume. This task involves finding a drawing method that can easily draw and reconstruct the surfaces. The second task includes the incorporation of a physics system into the program, since the project is based upon physical simulation of deformable objects. Lastly the program needs an interior system which reacts to collision forces. These interior and exterior systems must be linked in such a way that the exterior system passes forces to the interior system. After the forces act upon the interior, the exterior receives some displacement which is visible to the user, via the deformation of the surface.

2 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.

2.1 Volume exterior

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

2.1.1 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 $2^8 = 256$. 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).

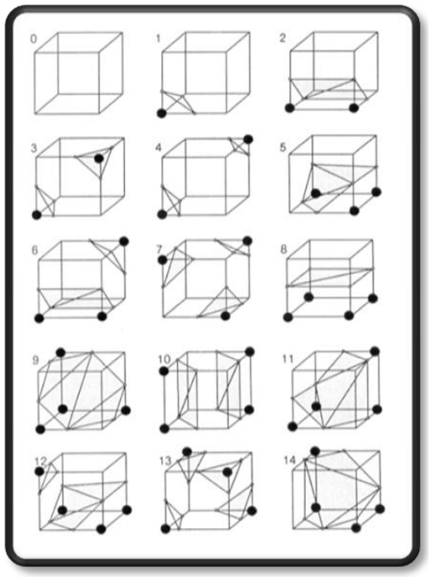

Figure 2 - 15 Cube triangulations (Lorensen and Cline 1987, p165)

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:

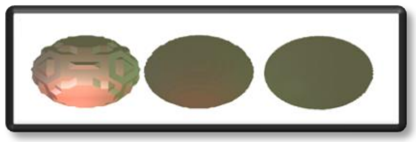
Figure 3 - Voxel sphere of resolution 20, 40 and 80 respectively (Author's student project)

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

Table 1 - Running times of the marching cubes algorithm at various resolutions.

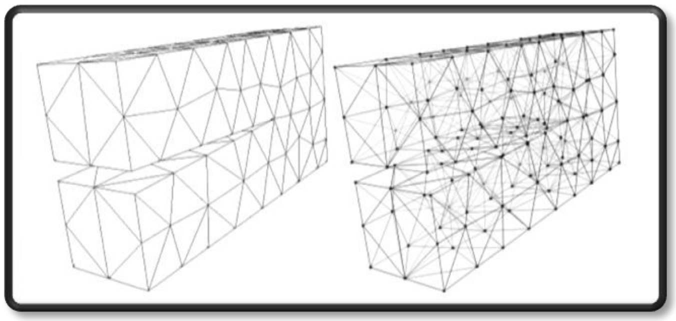

Figure 4 - Tetrahedral mesh. Left: Only external triangles shown. Right: Internal structure shown (O'Brien 2000, p.31)

2.1.2 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.

2.1.3 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.

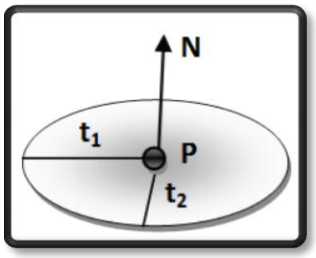

Figure 5 –A black surfel

Figure 5 shows an image of a surfel with normal **N**, two tangents $\mathbf{t_1}$ and $\mathbf{t_2}$ and position **P**. The tangents are tangent to each other as well as to the normal.

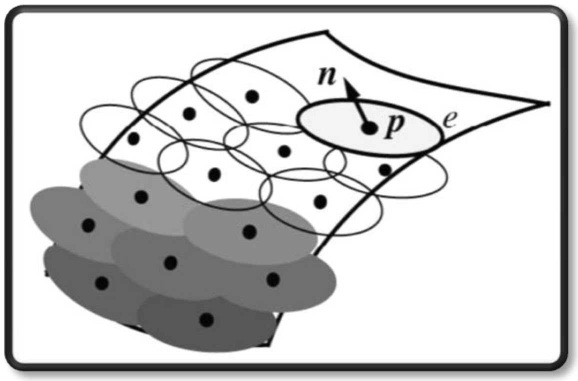

Figure 6 - Elliptical surfels covering a smooth 3D surface. The surfels draw over the adjacent surfels to cover up holes (Pajarola, Sainz And Guidotti 2004, p 599)

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 7). 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 6 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

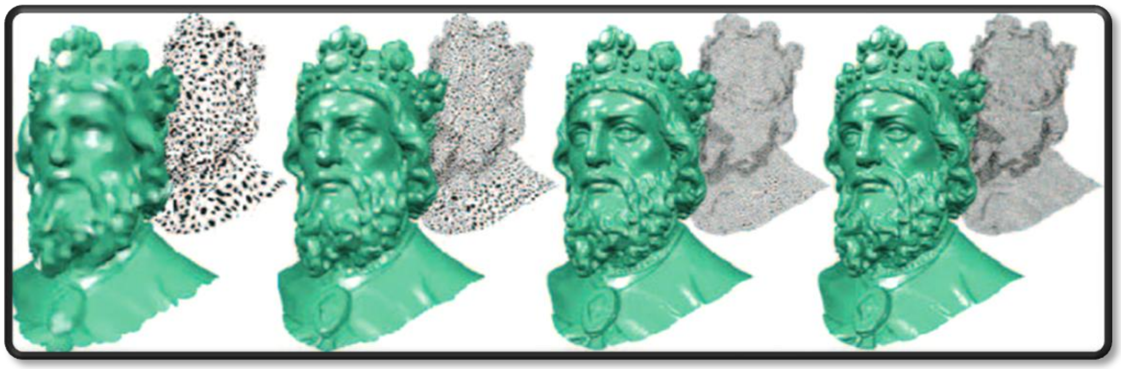

Figure 7 - 3D model of Charlemagne (600.000 points) with 2.000, 10.000, 70.000 and 600.000 surfels respectively (Gross and Pfister 2007, p 133)

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).

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.

2.2 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.

2.2.1 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 – Mass spring systems 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).

2.2.2 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 (ρ), deformation (***u***), volume (*v*), strain (ε) and stress (σ) (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.

Instead of springs between phyxels, the phyxels incorporate a weighing function (ω) 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 $\omega(r,h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3 & if\ r < h \\ 0 & otherwise \end{cases}$, 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 $\rho_i = \sum_j m_j \omega_{ij}$, where $\omega_{ij} = \omega(|x_j - x_i|, h_i)$ (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.

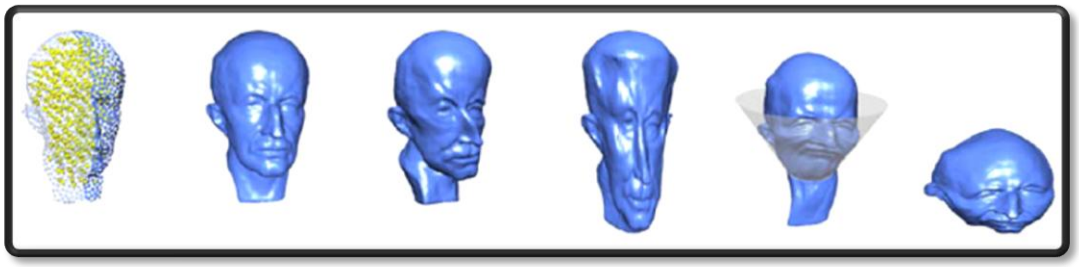

Figure 8 - Max Planck represented with surfels and phyxels. Two leftmost heads are undeformed. The four right heads are deformed elastically, plastically, melted and solidified (respectively) (Müller et al. 2004, p 356)

2.3 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 $\boldsymbol{f}_n$ is applied perpendicular to the area, resulting in an elongation of the beam by Δ*l*. The beam's stress σ is defined as $\frac{\boldsymbol{f_n}}{A}$ and therefore has the unit $\frac{N}{m^2}$. The strain ε however has no unit as it is defined as $\frac{\Delta l}{l}$. 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 7 in Appendix B.

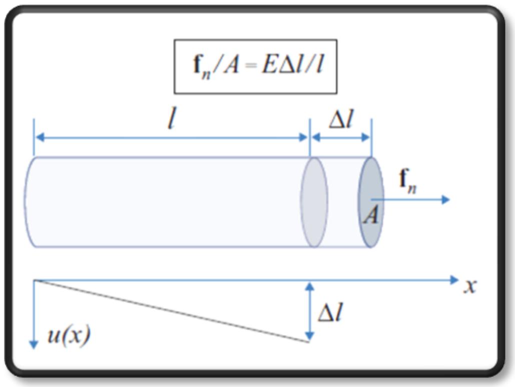

Figure 9 - Hooke's law (Gross and Pfister 2007, p343)

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):

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{xy} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{xz} & \varepsilon_{yz} & \varepsilon_{zz} \end{bmatrix} \quad (2.1) \qquad \sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \quad (2.2)$$

Since the stress and strain tensors are symmetric, they only contain 6 individual variables.

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & v & v & 0 & 0 & 0 \\ v & 1-v & v & 0 & 0 & 0 \\ v & v & 1-v & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2v & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2v & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2v \end{bmatrix} \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \epsilon_{xy} \\ \epsilon_{yz} \\ \epsilon_{zx} \end{bmatrix} \qquad (2.3)$$

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 $\frac{1}{2}$.

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 ($\boldsymbol{x}$ = $[x, y, z]^T$) is moved to its world coordinates ($\boldsymbol{p}(\boldsymbol{x})$ = $[x', y', z']^T$), both of which are defined as vector fields. The deformation of the object can be defined by a displacement field $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{p}(\boldsymbol{x}) - \boldsymbol{x}$ (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.

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:

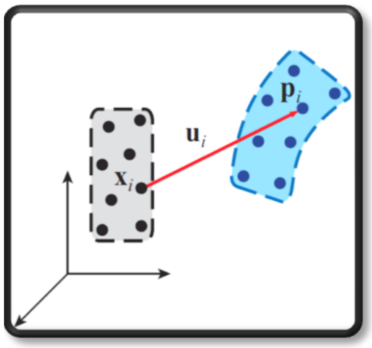

Figure 10 - Object's undeformed state (left) and deformed state (right) (Gross and Pfister 2007, p345)

$$\boldsymbol{M} = \sum_j \boldsymbol{x}_{ij}\boldsymbol{x}_{ij}^T\omega_{ij} \tag{2.4}$$

$$\nabla\boldsymbol{u} = \begin{bmatrix} u_{,x} & u_{,y} & u_{,z} \\ v_{,x} & v_{,y} & v_{,z} \\ w_{,x} & w_{,y} & w_{,z} \end{bmatrix} = \begin{bmatrix} \nabla u^T \\ \nabla v^T \\ \nabla w^T \end{bmatrix} \tag{2.5}$$

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:

$$\nabla u = \boldsymbol{M}^{-1}\left(\sum_j (u_j - u_i)\boldsymbol{x}_{ij}\omega_{ij}\right) \tag{2.6}$$

$$\nabla v = \boldsymbol{M}^{-1}\left(\sum_j (v_j - v_i)\boldsymbol{x}_{ij}\omega_{ij}\right) \tag{2.7}$$

$$\nabla w = \boldsymbol{M}^{-1}\left(\sum_j (w_j - w_i)\boldsymbol{x}_{ij}\omega_{ij}\right) \tag{2.8}$$

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:

$$\epsilon = \frac{1}{2}(\nabla \boldsymbol{u} + [\nabla \boldsymbol{u}]^T + [\nabla \boldsymbol{u}]^T \nabla \boldsymbol{u}) \tag{2.9}$$

Finally, a single matrix ($\boldsymbol{B}$) is used to distribute internal forces throughout the volume:

$$\boldsymbol{B} = -2v(\nabla \boldsymbol{u} + \boldsymbol{I})\sigma \boldsymbol{M}^{-1} \tag{2.10}$$

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

```
forall phyxel $i$
  initialize $\boldsymbol{x}_i$, $\boldsymbol{u}_i = 0$, $\dot{\boldsymbol{u}}_i = 0$, $\boldsymbol{v}_i$, $m_i$, $\rho_i$;
  calculate $\boldsymbol{M}_i^{-1}$ using equation (2.4);
endfor
loop
  forall phyxels $i$ do $\boldsymbol{f}_i = \boldsymbol{f}_{ext}(\boldsymbol{x}_i + \boldsymbol{u}_i)$; endfor
  forall phyxels $i$ do
    calculate $\nabla u, \nabla v$ and $\nabla w$ using equations (2.6, 2.7 and 2.8);
    calculate $\epsilon$ using equation (2.9);
    calculate $\sigma$ using equation (2.3);
    calculate $\boldsymbol{B}$ using equation (2.10);
    forall neighbouring phyxels $j$
      $\boldsymbol{f}_j = \boldsymbol{f}_j + \boldsymbol{B}(\boldsymbol{x}_{ij}\boldsymbol{\omega}_{ij})$;        $\boldsymbol{f}_i = \boldsymbol{f}_i - \boldsymbol{B}(\boldsymbol{x}_{ij}\boldsymbol{\omega}_{ij})$;
    endfor
  endfor
  forall phyxels $i$
    $\dot{\boldsymbol{u}}_i = \dot{\boldsymbol{u}}_i + \Delta t \frac{f_i}{m_i}$;        $\boldsymbol{u}_i = \boldsymbol{u}_i + \Delta t \dot{\boldsymbol{u}}_i$;
  endfor
  render system at $\boldsymbol{x}_i + \boldsymbol{u}_i$;
endloop
```

Pseudo code 1 - 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).

3 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.

3.1 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 can 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 (**UV**) and a delta texture coordinates (**∇UV**), 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 **∇UV** 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 11a). 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 11b). Whenever the surfels change in any way, the vertex stream containing the quads is rebuilt as shown in Figure 11c. 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 $\begin{cases} 1, if\ dist((x,y),(radius,radius)) < radius \\ 0, if\ dist((x,y),(radius,radius)) \geq radius \end{cases}, for\ x,y \in [0, 2 * radius]$. The surface texture of the quad is multiplied by this alpha texture to create the surfel disk.

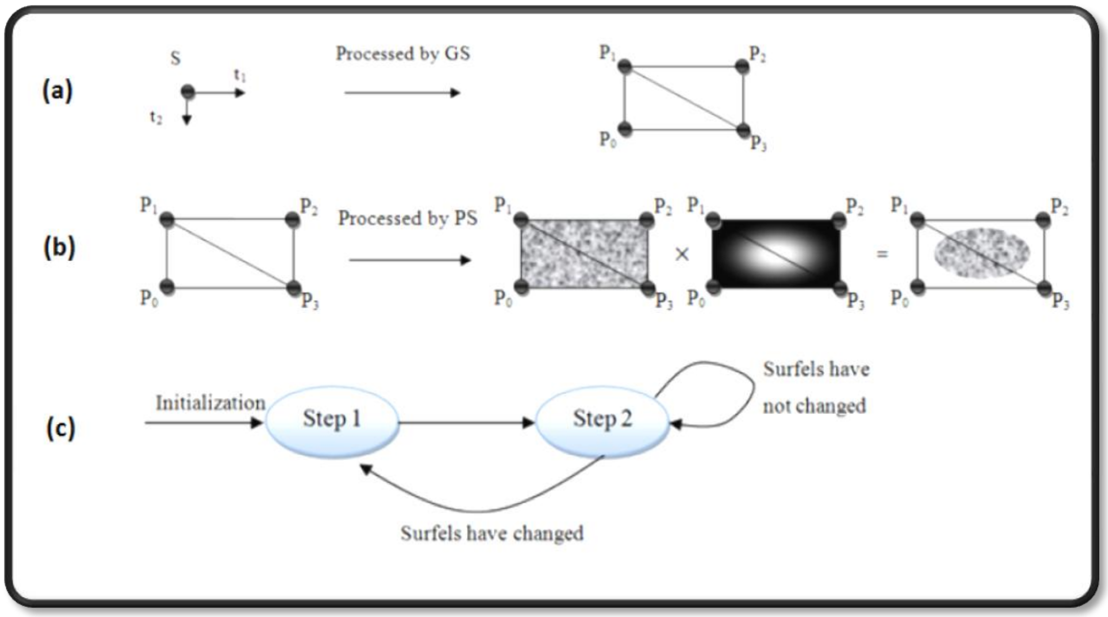

Figure 11 - (a) First step of the drawing method. (b) Second step of the drawing method. (c) The drawing method

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 P_0 in Figure 11a is $P_{0p} = S_p + t_2 - t_1$, where P_{0p} and S_p are the positions of P_0 and S respectively. Each point's surface texture coordinate is calculated in the same way as its position is. The only difference is that $\nabla\mathbf{UV}$ is used instead of the tangents and $\mathbf{UV}$ instead of the position. Using P_0 again from Figure 11a, its surface texture coordinate is calculated by $P_{0uv} = S_{uv} - \nabla\mathbf{UV}_{\mathrm{t}_1} + \nabla\mathbf{UV}_{\mathrm{t}_2}$. Since the quads are represented as elliptical splats, the quad need texture coordinates for the alpha texture, $\mathbf{EWAUV}$. Because the alpha texture is circular, the $\mathbf{EWAUV}$ coordinates are always the same for every surfel that is not an edge (covered in the next chapter). These coordinates are $(0, 0)$, $(1, 0)$, $(1, 1)$ and $(0, 1)$ for P_0, P_1, P_2 and P_3 respectively. The normal of the quad is always the same as the normal of the surfel's.

3.2 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 clipping plane is represented as a 3D vector with values depicted in Figure 12:

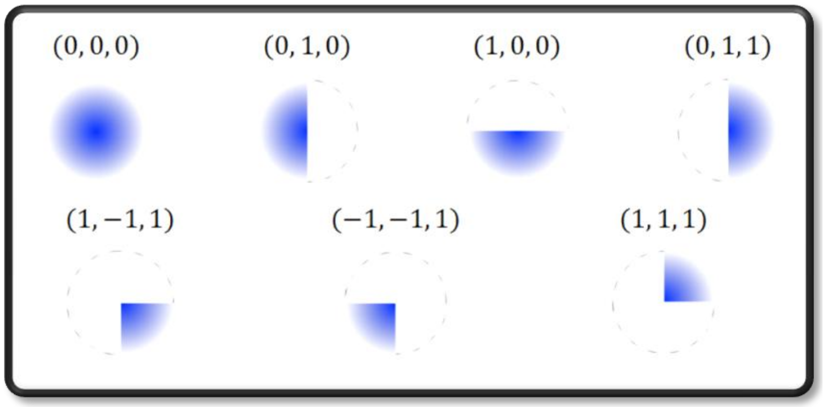

Figure 12 - Surfel Clip planes

The calculation of the quad properties is slightly different for edges, since the clipping plane has an effect on the surfel's position, $\mathbf{UV}$ and $\nabla\mathbf{UV}$. 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 13a shows, the calculation of these properties for points P_0 and P_1 are different from Figure 11a. If the surfel's clipping plane depicts that the surfel's size is not extended into the negative t_1 direction (as is shown in Figure 13b), 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 13c shows a wall created by surfels.

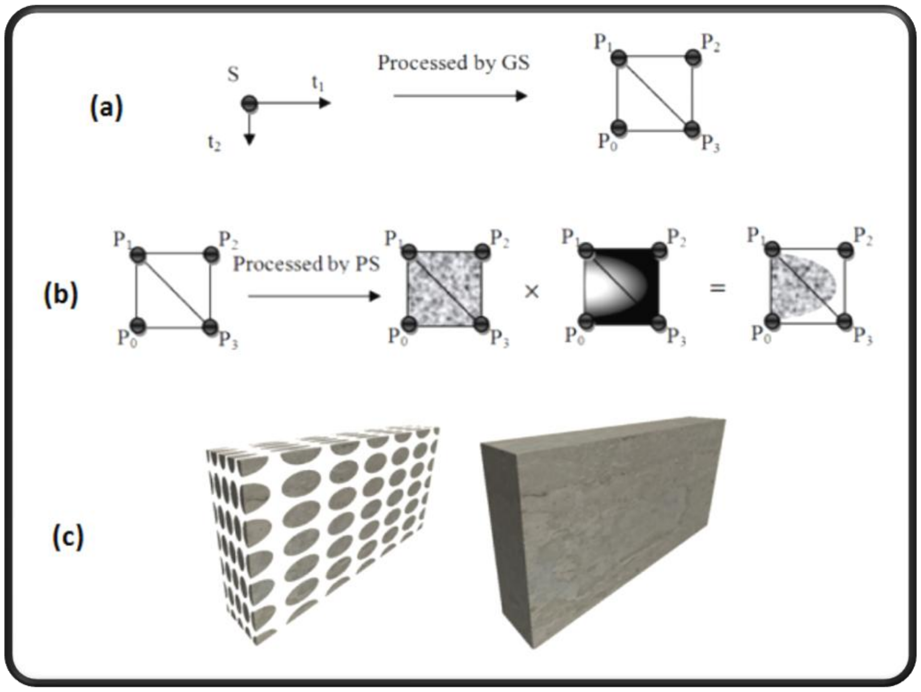

Figure 13 - (a) and (b) show rendering method for an edge with clipping plane (0, 1, 1). (c) Wall rendered with surfels. On the left the surfels are scaled down 50%

3.3 Vertex buffer grid

When a surfel changes in the volume, the vertex buffer used to draw the surface needs to be regenerated. If this vertex buffer is large enough, it can take a few milliseconds to regenerate, 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 (Murchison 2008), which means that each frame cannot take more than 33.333 milliseconds. If a few milliseconds are used to regenerate 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 vertex buffer grid (VBG), shown 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 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 regenerated.

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 2.0. This configuration results in a good frame rate as well as a quick vertex buffer reconstruction time.

3.4 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 VBG 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 and a rectangular rigid body created based on each surfel's vertices. Additionally a custom contact listener is created for each surfel to detect collisions between it and the wrecking ball or it and the projectiles.

3.5 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 14. 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 15 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.

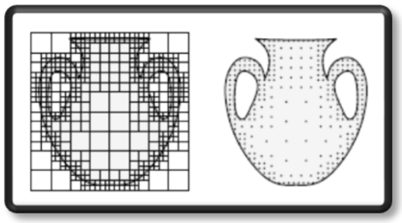
Figure 14 - Phyxels distributed by phyxels (Pauly et al. 2005, p. 961)

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 16. 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

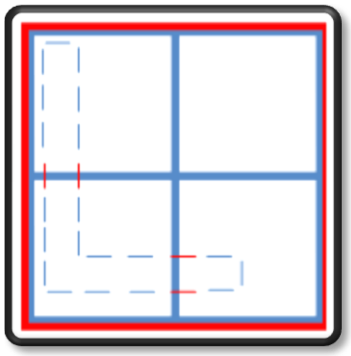
Figure 15 - Quadtree. Thin lines represent surfels and thick lines represent the quadtree nodes

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 of the weighing function ω described in Chapter 2.2.2, which controls how much affect a force has on a phyxel. This weighing function is used both to determine the force applied from a surfel to a phyxel, as well as from phyxel to neighbouring phyxels. The support radius of a phyxel is set as three times the average length to neighbouring phyxels: $\frac{\sum_{i=0}^{n} |x_p - x_i|}{n}$, where n is the neighbour count and x_p is the position of phyxel p. However when using ω with a large support radius, the weight drops off very quickly, resulting in a small area of affect. Due to this fact, the phyxel grid cell size that showed the best results was of size 1.25. This value is therefore used throughout the rest of the dissertation.

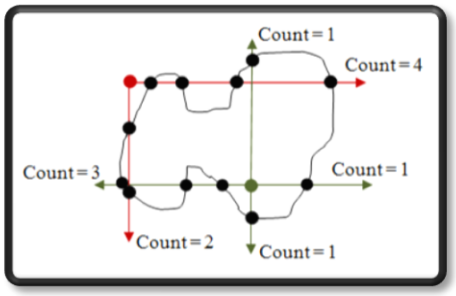

Figure 16 - Jordan Curve Theorem. The green point is inside the polygon while the red point is outside of the polygon

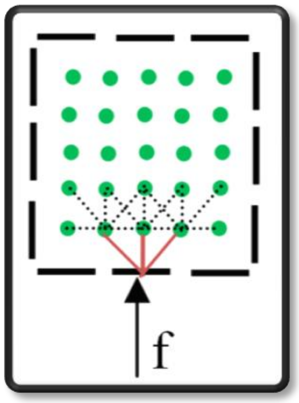

Figure 17 - Force applied to a surfel. Surfels are displayed as black lines and phyxels are displayed as green dots.

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 volume 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 Figure 17 with red lines. The affected phyxels apply the force to their neighbours as proposed in **Error! Reference source not found.**, shown in Figure 17 with dotted lines.

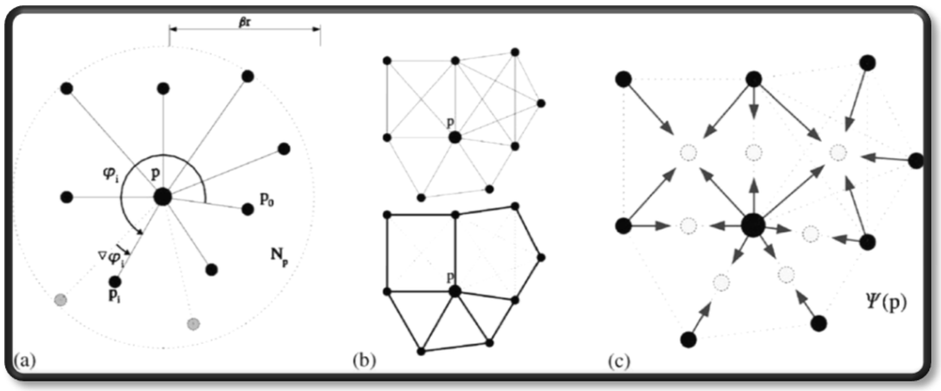

Figure 18 - Examples of (a) neighbour selection, (b) surfel polygons creation and (c) surfel resampling (Guennebaud, Barthe and Paulin 2004, p 830)

3.6 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 18a). According to Guennebaud, Barthe and Paulin (2004), $\frac{\pi}{8}$ has been proven to be a good choice for this smallest angle, however $\frac{\pi}{10}$ is used in this project because it allows surfels to have more neighbours. 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 $\boldsymbol{N_S} = \{\boldsymbol{S0}, \boldsymbol{S1}, \boldsymbol{S2}, \boldsymbol{S3}\}$. The condition needed for the creation of a surfel polygon $\boldsymbol{P} = \{\boldsymbol{S}, \boldsymbol{S0}, \boldsymbol{S1}\}$, then the condition $\{\boldsymbol{S}, \boldsymbol{S1}\} \in \boldsymbol{N_{S0}}, \{\boldsymbol{S}, \boldsymbol{S0}\} \in \boldsymbol{N_{S1}}$ and $\{\boldsymbol{S0}, \boldsymbol{S1}\} \in \boldsymbol{N_S}$ 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 19). Newly created surfels receive averaged properties from their parents (black dots in Figure 18 and Figure 19). Finally after the refinement procedure, the tangent axes and **∇UV** of the parent surfels are reduced down to 75% of their original size. However, with this improved 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.

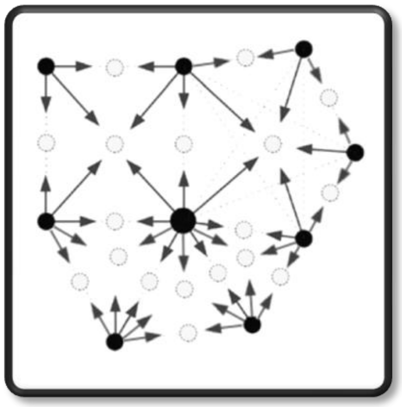
Figure 19 - Updated surfel refinement

When forces are applied to the volumetric objects, the surfels are moved by the displacement of the phyxels. Additionally to create craters in the volume, the surfels need to be rotated in such a way that the surfels create a watertight structure. This is done by combining vector cross-product calculations and a technique called billboarding. The

$$M_{rot} = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ l_x & l_y & l_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

billboarding technique has mostly been utilized for particle systems where the particles are represented as 2D sprites. The billboarding technique creates a rotational matrix that makes these 2D particles always face the camera and thus creating the illusion that the particles are in 3D (Luna 2008).

Equation 3.1 shows how the billboard rotational matrix is created, where r is the right

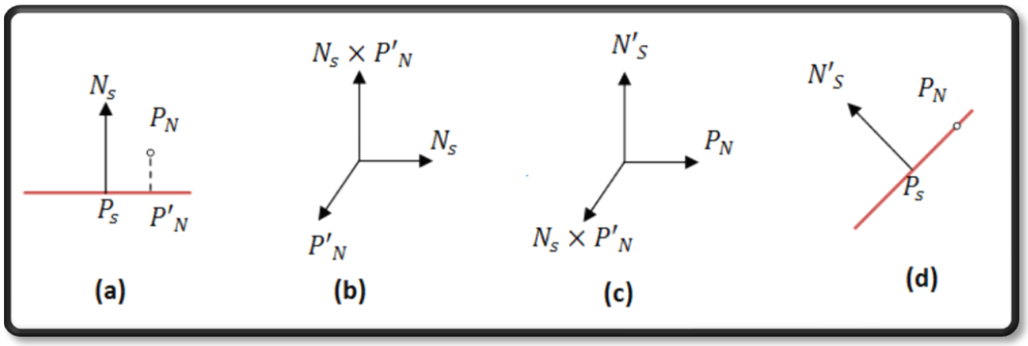

Figure 20 - (a) Neighbour position projected to surfel plane. (b) Surfel normal crossed by the projected position. (c) Neighbour position crossed with (b). (d) New normal of surfel

vector, u is the up vector and l is the look vector. The look vector is calculated as the difference between the camera's eye position and the centre of the particle. The up vector is always defined as the up vector of the sprite. Finally the right vector is the cross product between the two. However, since the surfels are not supposed to face the camera at all times, this matrix needs to be changed slightly by changing the look vector to the negative displaced normal of the surfel. To calculate the displacement of the surfel's normal, the positions of the neighbouring surfels are projected onto the plane of the surfel (Figure 20a). For each neighbour, its projected position is crossed with the surfel's normal (Figure 20b). The displaced normal is calculated by crossing the un-projected neighbour position by the crossed product (Figure 20c and Figure 20d). For all surfel neighbours, the average displaced normal is used as the new surfel normal.

The rotational matrix for the surfels is then setup by setting the look vector as the negative displaced normal, the up vector is the (0, 1, 0) vector and the right vector is the cross between the look and up vectors. This rotational matrix is used to calculate the new direction of the tangent vectors.

3.7 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 Table 2. 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.

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.

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 2 - 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 3 - Surface properties

4 Results

This chapter covers the results of merging the methods covered in Chapter 3. Firstly the program setup is discussed briefly, followed by an in-depth analysis of the project's success with regards to the project's goals. The performance analysis is twofold, with quantitative and qualitative analysis. Firstly, the performance of the program is measured with a quantitative analysis, where frame rate (FPS) and milliseconds per frame (MPF) is computed. Secondly the program is analysed with regards to the visual quality, called the qualitative analysis.

4.1 Program setup

Figure 21 shows the program created for this project. The program contains a white wall and a wrecking ball hanging from a chain, which can be moved in 3D. Additionally projectiles can be fired from the camera's position at a speed of 300 m/s. The projectiles have a weight of 1000 kg (1 ton), while the wrecking ball has a weight of 82.5 tons. While these weights are not exactly realistic, they do the job of damaging the wall. The wall's material properties are based upon the material properties for steel (Table 7 in Appendix B – Material properties of various objects).

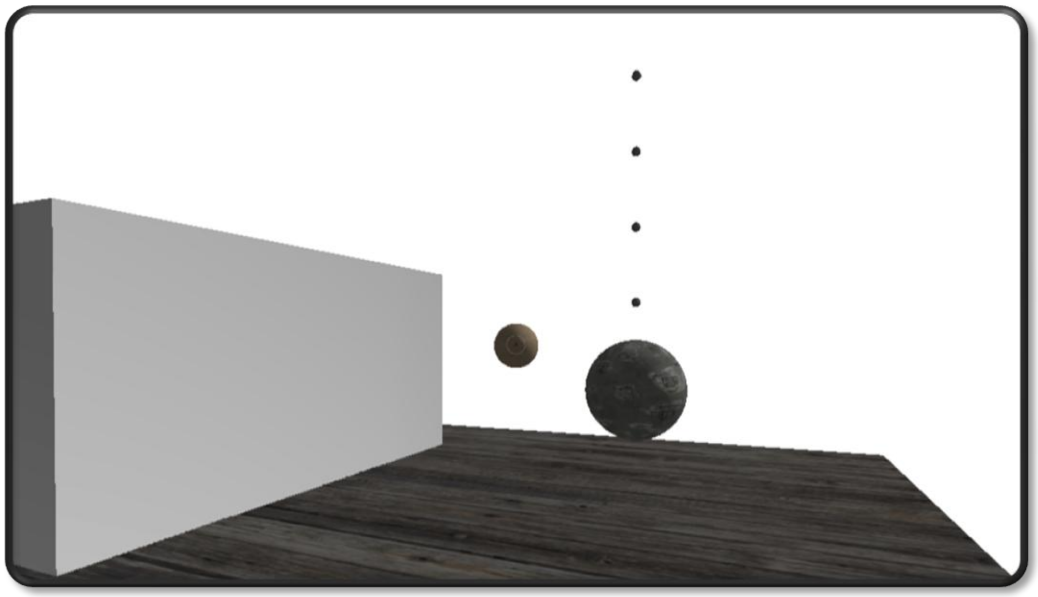

Figure 21- Deformable wall (left). Projectile being fired away from the camera (middle). Wrecking ball hanging from a chain (right). Un-deformable floor (bottom)

Additionally the system incorporates a simple lighting technique, called Oren-Nayar reflectance model. This lighting method was incorporated into the system to allow dents and deformations to be seen easily, because surfels with varying normal vectors have varying light intensity. For more information on the Oren-Nayar lighting method, Oren and Nayar (1994) describe the procedure in their paper where they generalize the Lambert's reflectance model. Because of this lighting method, the wall appears to be gray instead of actually being white, since the light's direction is angular towards the wall.

4.2 Quantitative analysis

This sub chapter covers the performance of the surfel draw method as well as the performance of the deformation algorithm. The tests were run on a 2.80GHz Intel Core2 Duo processor with a 1.0GB ATI Mobility Radeon HD 4650 video card.

4.2.1 Surfel draw method

When utilizing the surfel drawing method proposed in Chapter 3.1 – Chapter 3.3, objects made up from surfels can be represented in a few ways. The VBG's size can be changed in such a way that each cell contains many or few surfels. If the VBG cell size is very small it requires the system to traverse many cells during each draw call. However when the VBG cell size is very large, the vertex buffer reconstruction is reconstructing too many surfels. Additionally the size of the surfels can also be changed, with no loss of visual quality of the un-deformed wall. However when the surfels are very big, the deformation does not look realistic enough (Chapter 0).

Because of the nature of this test, only the wall is drawn. This is because the frame rate of the surfel draw method is being tested. The system is run for thirty seconds without any user interaction. Only one face of the wall is changed between tests in addition to the VBG cell size. This face is always visible by the camera throughout the thirty seconds of testing time. The quantities measured in this test are the FPS and MPF. These two quantities are linearly related, where $MPF = \frac{1000ms/s}{FPS}$. Table 4 shows the average and mode values of the FPS and MPF measured in this test; where the average shows the average from start to finish and the mode value shows the value that occurred most often during the testing time.

Surfel count	Vertex buffer grid cell size	Frame rate (Average / Mode)	Milliseconds per frame (Average / Mode)
54	100.0	739 FPS / 741 FPS	1.353 ms / 1.350 ms
54	10.0	699 FPS / 704 FPS	1.431 ms / 1.420 ms
54	1.0	688 FPS / 704 FPS	1.463 ms / 1.420 ms
541	100.0	698 FPS / 700 FPS	1.431 ms / 1.429 ms
541	10.0	659 FPS / 699 FPS	1.440 ms / 1.431 ms
541	1.0	83 FPS / 84 FPS	12.112 ms / 11.905 ms
7426	100.0	550 FPS / 593 FPS	1.828 ms / 1.686 ms
7426	10.0	512 FPS / 509 FPS	1.961 ms / 1.965 ms
7426	1.0	72 FPS / 74 FPS	13.889 ms / 13.514 ms
29206	100.0	475 FPS / 477 FPS	2.108 ms / 2.096 ms
29206	10.0	498 FPS / 501 FPS	2.011 ms / 1.996 ms
29206	1.0	82 FPS / 84 FPS	12.201 ms / 11.905ms
115966	100.0	242 FPS / 242 FPS	4.136 ms /4.132 ms
115966	10.0	252 FPS / 253 FPS	3.977 ms / 3.953 ms
115966	1.0	82 FPS / 85 FPS	12.089 ms / 11.765 ms

Table 4 - Performance of the surfel drawing method

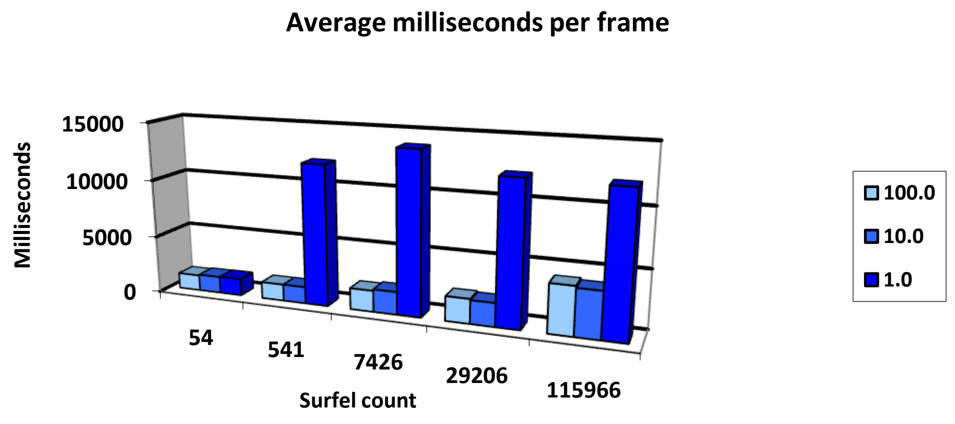

Figure 22 - Surfel drawing method chart at varying vertex buffer grid cell sizes

These results show that the surfel drawing method has a high FPS count and reasonably low milliseconds used for drawing each frame. The noticeable result of this test is that

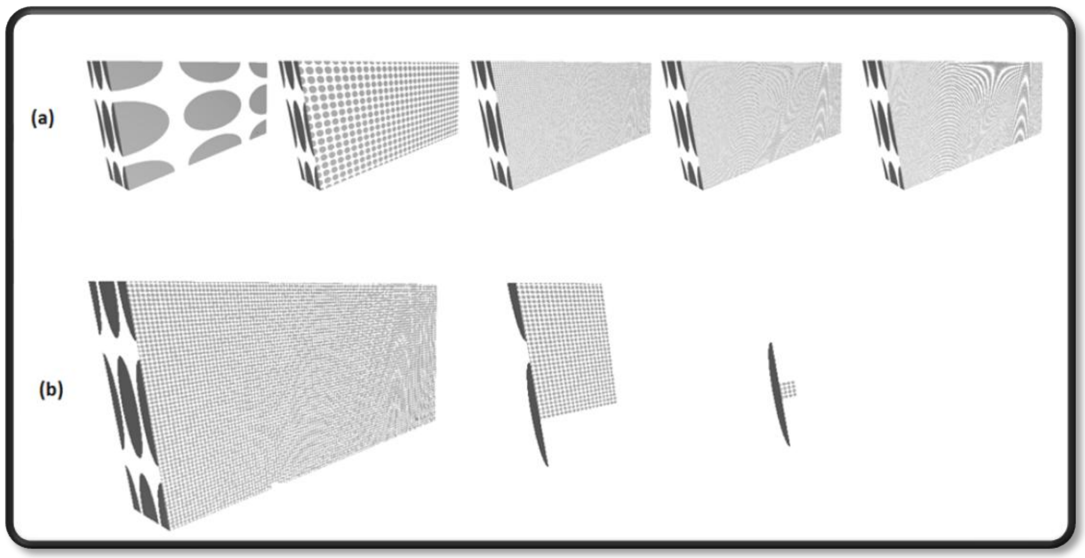

Figure 23 – (a) Wall with variable surfel count. From left: 54, 541, 7426, 29206 and 115966 surfels. Surfels are scaled down by 50%. (b) Wall with only one vertex buffer grid cell shown at each time. Cell sizes from left: 100, 10 and 1

for VBG cell size 100.0 and 10.0 the frame time increase between 54 surfels and 115966 surfels is less than three milliseconds. This shows that this method is very fast and useful for computer game graphics. The drawing method starts to slow down when the VBG cell size goes down to 1.0. This is because of there are very many cells that are populated and therefore it takes a long time to traverse. Figure 23a shows the difference between the surfel counts of the wall, while Figure 23b compares various VBG cell sizes. The cell sizes were strategically chosen because the largest cell size fully includes the whole wall and therefore mimics a single vertex buffer, while the smaller cell sizes show the effect of splitting up the volume.

One noticeable aspect of the results is the fact that the frame rate is quite stable when the cell size is set to one. This can be seen by comparing the difference between rendering the wall with 541 surfels and 115966 surfels with cell size set to ten and one. The difference between rendering time when the cell size is ten is around 2.5ms. However when the cell size is one the difference drops down to 0.14ms. This result clearly shows the power of splitting a large volume in multiple vertex buffers.

4.2.2 Deformation method

One of the project's goals was to create a real time wall deformation technique. The method to perform the quantitative analysis of the deformation technique was to measure the milliseconds it takes to finish. The deformation tests were set up similarly to the surfel draw method in the previous chapter. However, the surfel count range is much smaller because of limits in the Havok engine (discussed in Chapter 4.3).

The test that was carried out measures the deformation time of the deformable wall. The wall was set up with varying surfel count, where the surfels of a single face was changed at a time. During the test, the frame rate was kept constant at 60 FPS. This is because the delta time between frames is used to calculate the force propagation in **Error! Reference source not found.**. If the delta time is too small then the force is only applied to the phyxels for a very short time. This is because the impact only occurs in a single frame; therefore if the MPF is too low then the force propagation is very limited. On the other hand, if the MPF is very high, then there is a large amount of force that is propagated through the material.

The wall is deformed five times with the wrecking ball and a projectile. The average, minimum and maximum time it takes for the deformation algorithm to finish is shown **Error! Reference source not found.**:

Surfel Count	VBG cell size	Deformation time (ms) Average (Minimum / Maximum)	
		Projectile	Wrecking ball
541	100.0	462.4 (400 / 554)	843,4 (722 / 911)
541	10.0	322.8 (301 / 359)	599.4 (479 / 657)
541	1.5	309.4 (225 / 343)	617.4 (427 / 695)
1936	100.0	2690.8 (2590 / 2759)	4485.4 (4218 / 4633)
1936	10.0	1157.4 (1076 / 1306)	2139.0 (1797 / 2415)
1936	1.5	1139.0 (1158 / 1449)	1325.2 (1241 / 1436)
7426	100.0	25762.0 (25415 / 26423)	32448.0 (30499 / 33441)
7426	10.0	6991.0 (6835 / 7198)	12408.0 (11173 / 13785)
7426	1.5	4529.6 (4315 / 4883)	7865.4 (7332 / 8357)

Table 5 - Performance of the deformation algorithm

When the deformation test results are examined, a few things stand out. Firstly the immense speed improvement when the VBG cell size is changed from 100 down to 1.5, which for the dense wall is an improvement of approximately 75%. Secondly the link between how big the deformity is and how long the algorithm takes is apparent, since the wrecking ball deformation takes longer in every case. Lastly, it is clear that the deformation algorithm is far from running in real-time. A real-time run is defined as running in mostly one frame-time, which for 60 FPS is 16.67ms. The reason for this slow run time is the amount of work needed when a single surfel is displaced, which is described in Appendix D – Volume deformation.

4.2.3 Vertex buffer grid performance

The last quantitative test is to check if the VBG does improve vertex buffer regeneration. In this test a single surfel is changed, which results in changing the surfel's vertex buffer. As a control test, the whole wall is contained in a single vertex buffer cell, which mimics the use of a single vertex buffer. The VBG size is reduced by 75% four times. The surfel count is not changed during the tests and is kept constant at 115966 surfels. These tests were performed ten times and the results in Table 6 show clearly that the vertex buffer splitting does shorten the regeneration time.

VBG cell size	Average regeneration time (ms)	Mean regeneration time (ms)
100.0	37.0	36.0
25.0	23.9	24.0
6.25	1.5	2.0
1.5625	0.6	1.0

Table 6 - Vertex buffer regeneration test

4.3 Havok and surfels

During system testing, a rigid body cap was encountered in the Havok system. This cap limits the rigid body count to about 17000, due to the memory consumption of each rigid body. Capping the count of rigid bodies in the system directly affects the surfel count of the system, because of the fact that each surfel contains a rigid body. This results in less detail in the deformed surfaces. One method to overcome this might be to combine surfels that have not been displaced into a single rigid body. When a surfel is displaced, the single rigid body is then regenerated in such a way that deformed surfels contain their own rigid body and undeformed surfels are combined to create a single rigid body.

4.4 Qualitative analysis

One of the aims of the project is to create a deforming volume that is realistic. This is done by combining the method in **Error! Reference source not found.** and the surfel refinement method described in Chapter 3.6. To perform the qualitative analysis, the realism of the deformation needs to be evaluated as well as the surfel rendering method.

4.4.1 Surfel draw method

When using the surfel draw method the wall can be displayed with various surfel sizes, while maintaining the same wall structure. However as Chapter 2.1.3 stated, there is a problem with overdraw with the surfel draw method. This problem occurs when a single pixel is drawn multiple times. To see how much overdraw is occurring, a simple shader was constructed. This shader draws a white colour with a low alpha value at each pixel of a surfel. When overdraw arises, the current pixel's alpha value is added to the previous pixel value. With this technique, areas with the brightest white colour have the most overdraw (Figure 24).

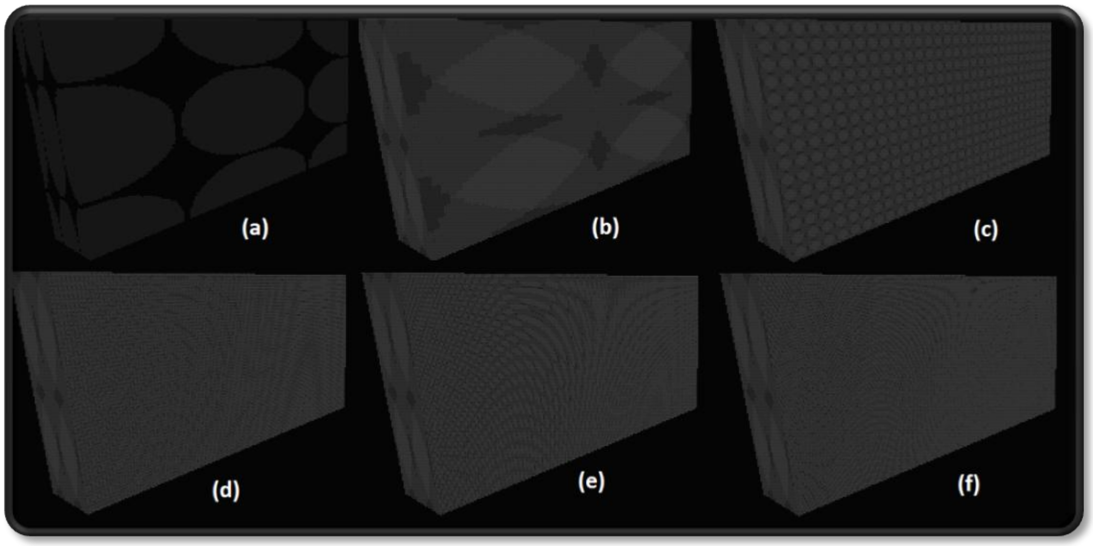

Figure 24 - Overdraw test. (a) No overdraw, surfels scaled down by 50%. Figures b, c, d, e and f show overdraw with 54, 541, 7426, 29206 and 115966 surfels respectively, which are not scaled down

Another problem with the surfel method is when the surfel count is very high. This results in visual artefacts when the wall is viewed from an angle, which is shown in Figure 25. This occurs because the surfels are represented as quads that contain an elliptical surfel texture that has no alpha value outside of the ellipse. When the surfels are projected into screen space during rendering, the surfels near the camera have a large enough pixel area to be rendered correctly. However, the surfels drawn further away from the camera have a smaller pixel area. If a surfel is projected to the screen space in such a way that the visible area of the texture is dominated by the invisible area, the surfel becomes invisible. This is shown in Figure 26 where four equally sized surfels are projected onto screen space. The eye looks at the screen, represented here as a gray grid, where filled gray cells represent cells with no alpha value. Each surfel contains a gray rectangle which represents the surfel quad. The green surfel is closest to the eye and thus has the largest pixel area. However, the purple surfel is very far away from the eye and has only its transparent texture pixels projected onto the screen space.

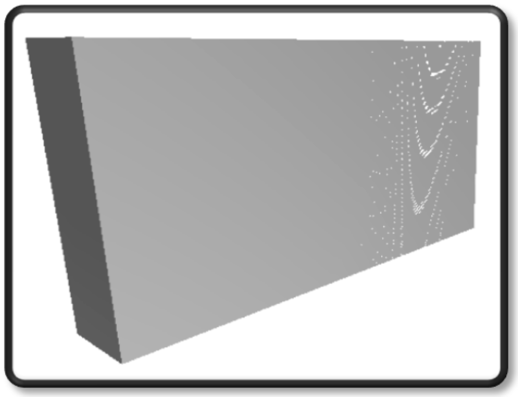

Figure 25 – Pixel swirling with surfel count 115966

To prove that this is the problem at hand, the surfel texture was replaced with a rectangular white texture with full alpha value. When this texture was multiplied to the surface texture, the pixel swirling effect disappeared. However, there is an alternate way to alleviate this visual artefact. That is by forcing the surfel's screen space size to be larger than a certain threshold. For instance, this threshold can determine that the smallest pixel area a surfel can have is 4x4 pixels in screen space.

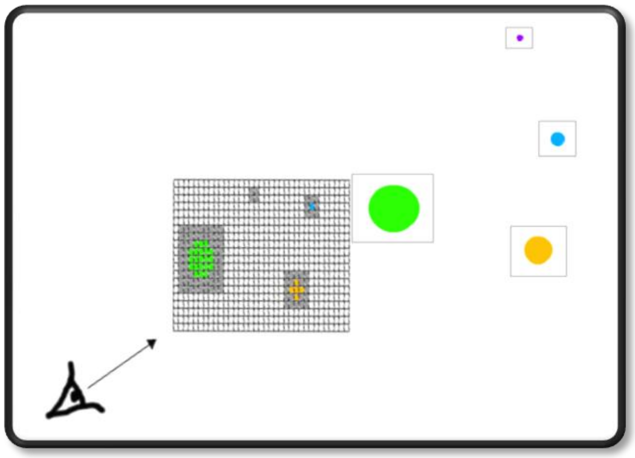

Figure 26 - Surfels projected into screen space. The purple surfel is invisible

4.4.2 Volume deformation

Since the wall can be deformed by both a wrecking ball and a projectile, both need to be evaluated. Additionally, because the program provides an opportunity of multiple material properties of deformable volumes, multiple property arrangements need to be evaluated. Figure 27 shows the wall deformation results where the left dent was made by a projectile, while the right dent was created by the wrecking ball. The wall created in these tests was given various materialistic properties, seen in Appendix B – Material properties of various objects. In Figure 27a the wall's material is aluminium, whereas in Figure 27b and Figure 27c the material has two different combinations of steel. Lastly the wall in Figure 27d is made from titanium. When looking at these results, it is clear that the force propagation method works very well. This can be seen by the fact that the relatively light projectile does less damage, if any, than the heavy wrecking ball. This is exactly the results one might expect in a realistically simulated world, that heavier objects inflict more damage than light objects. Another result that is worth noting is the fact that different materialistic properties are deformed in different ways. This can be seen by comparing the soft steel wall to the titanium wall. Both walls are made up of the same surfel count and are damaged similarly by the same objects.

However, to produce realistically looking deformation, the surfel count of the deformed surface needs to be high. This is evident from Figure 27a-d where the topmost wall has relatively few surfels on the deformed surface and therefore has a deformation that is not very realistic. This is because of the resampling algorithm, where a surfel is resampled into more surfels that are 75% of its original size. It is therefore evident that if surfels are large to begin with, they will still be somewhat large after resampling. This could be improved by introducing a resampling algorithm that resamples the deformed area multiple times until the area meets the quality expectations.

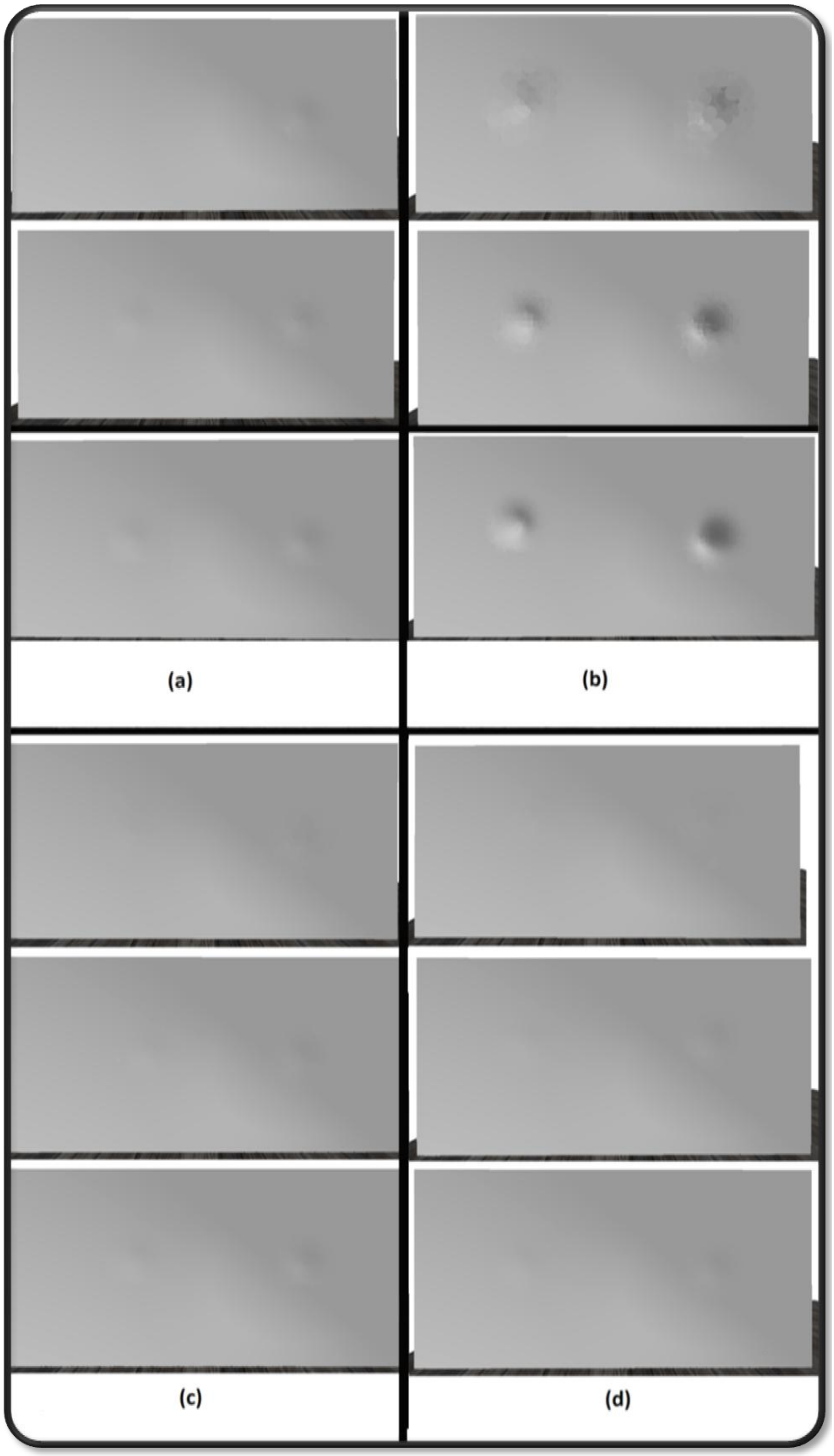

Figure 27 - Deformation of a wall with (a) aluminium, (b) soft steel, (c) hard steel and (d) titanium properties. Phyxel grid cell size is 1.25 in all cases. The surfel count in each case is 541, 1936, 7426 surfels, from top to bottom.

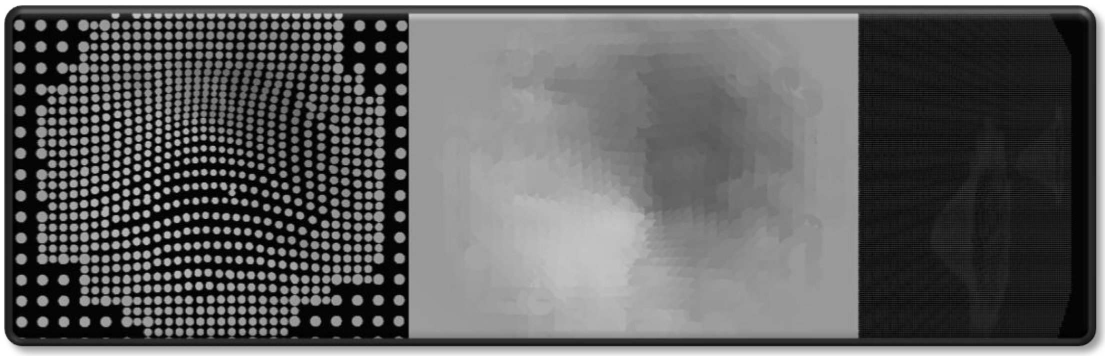
Figure 28 - Close up of Figure 27b with 7426 surfels. The left figure is scaled down by 70%. The middle figure is not scaled. The right figure shows the two deformations from the side with the overdraw shader

Looking at these results, it is clear that the surfel method provides an adequate method of displaying deformed volumes, while the surfel count is high enough. Figure 28 shows the deformation of a soft steel material with 7426 surfels. Even with such a high surfel count, there appear some visual artefacts because of surfel intersection. One way to remove the artefacts is to clip the surfels that intersect. This method is discussed by Wicke, Tehschner and Gross (2004) where they present a way to perform Boolean operations on volumes.

4.5 Expert opinions

This program was shown to a few experts, during informal interviews. These experts were asked to give their opinions on the surfel drawing method and the realism of the deformed surface at various surfel resolutions. However, the experts were not asked to give their opinion on the speed of the deformation algorithm, since it does not run in real time.

The response of the experts was generally positive, with every one of them discussing the immense potential of the surfel drawing method. They discussed that, with more and smaller surfels, the surfel drawing method could be used to great effect to mimic deformation of volumes. One expert said that some of the visual artefacts need to be ironed out, for example the lack of surfel clipping when the wall is deformed. However, this expert pondered whether or not this research would be worth doing, compared to the use of existing methods.

5 Discussion

With the results of Chapter 4 it is evident that surfels are a very viable option to draw volumetric objects in computer games. This can be seen with the drawing method's quantitative test as well as the quality of the deformations. Combined with the VBG, the surfel method describes a very powerful method of simulating the exterior of non-static volumes.

While the deformation algorithm does not run fast enough to be useful in computer games, it does create a realistic crater when the wall is struck with either the wrecking ball or projectile. This is evident by looking at Figure 27 and Figure 28, where the wall has clearly been deformed by a heavy object. One of the reasons why the deformation algorithm does not run in real time is because of the tight coupling of surfel count and deformation realism. If the deformation of the volumetric object is supposed to be very realistic, more surfels are needed. This results in more surfels that need to be displaced and resampled, which results in longer processing time. The process that takes the longest time to complete is the neighbourhood calculation for the surfels created during the resampling of deformed surfels. This is because of the high number of newly created surfels and the relatively high run time of the neighbourhood calculation method.

The frame-rate and frame-time calculations performed during testing were created by the author. During runtime, the frame-rates were accumulated after each second and the time taken to draw a single frame was accumulated after each frame. To calculate the average amounts the accumulated frame-rate was divided by the total seconds of runtime, while the collected frame-time was divided by the total frames during runtime. The mean values were calculated by counting how many times each frame-time and frame-rate value appears during runtime. Total confidence is in these methods and according to the results they appear valid.

According to the experts who gave their opinions regarding the project, the surfel based drawing method shows much promise for computer games. By sampling the surface with many small surfels, the deformations can become more realistic than it is already depicted by the program. One expert's opinion was that this rendering technique might need more research in the field of computer games, although whether such research would result in a better rendering than current techniques was contemplated by the expert. The author’s opinion is that such research would definitely be beneficial for the computer games development industry. This is because of the fact that surfels are very useful when it comes to approximating smooth surfaces. As a result, the more research carried out for surfels in computer games the more realistic the approximation of smooth surfaces becomes.

6 Conclusions and future work

The intention of this project was to find a realistic and efficient way to create deformations on non-static volumes with the use of a surface method that supports easy regeneration. To accomplish these objectives the surfel based rendering technique was chosen to render the deformable volume. By utilizing the geometry shader, the surfels were created as quads and stored in a VBG cell for efficiency purposes. When the volume was deformed, the vertex buffers of cells that contain deformed surfels were regenerated. The results of the tests performed in Chapter 4.2.3 show that regenerating changed VBG cells is much faster than regenerating a single vertex buffer which contains every surfel in the volume. Because of these results it is apparent that the surfel method can be easily used in computer games to represent deformable volumes.

When a deformable volume containing many small surfaces is struck by a heavy object, the amount of surfels that need resampling becomes very high. This is the main reason for the slow deformation time and is therefore a good choice for future work. Müller et al. (2008) discuss a few interior mechanism methods that could be developed to this extent. Additionally, it would be interesting to see the same deformation results as depicted in Figure 28 if the intersecting surfels would be clipped or if the surfels were blended with each other. Wicke, Tehschner and Gross (2004) discussed a clever clipping method that could be utilized for this purpose.

This document, in addition to the program's source code is available online at: http://code.google.com/p/olafurabertaymsc/.

7 Appendices

7.1 Appendix A – Mass spring systems

Pseudo code 2 shows implementation for a simple mass-spring system:

```
// initialization
  forall particles $i$
    initialize $\mathbf{x}_i$, $\mathbf{v}_i$ and $m_i$
  endfor

// simulation loop
loop

  forall particles $i$
    $\boldsymbol{f}_i \leftarrow \boldsymbol{f}^g + \boldsymbol{f}_i^{coll} + \sum_{j,(i,j)\in S} \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{x}_j, \boldsymbol{v}_j)$   (Eq. A.5)
  endfor

  forall particles $i$
    $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta t \frac{\boldsymbol{f}_i}{m_i}$   (Eq. A.7)
    $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + \Delta t \boldsymbol{v}_i$   (Eq. A.8)
  endfor

  display the system every nth time
endloop
```

Pseudo code 2 - 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 ($\boldsymbol{f}^g$) and collision force ($\boldsymbol{f}^{coll}$) 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.

$$\boldsymbol{f}^s(\boldsymbol{x}_i, \boldsymbol{x}_j) = k_s \frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \left(|\boldsymbol{x}_j - \boldsymbol{x}_i| - l_0 \right) \tag{A.1}$$

$$\boldsymbol{f}^s(\boldsymbol{x}_j, \boldsymbol{x}_i) = -\boldsymbol{f}^s(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{A.2}$$

Here attributes with the i^{th} and j^{th} indices represent adjacent particles i and j, k_s stands for the stiffness of the spring and l_0 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:

$$\boldsymbol{f}^d(\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{x}_j, \boldsymbol{v}_j) = k_d(\boldsymbol{v}_i - \boldsymbol{v}_j)\frac{\boldsymbol{x}_j - \boldsymbol{x}_i}{|\boldsymbol{x}_j - \boldsymbol{x}_i|} \qquad \text{(A.3)}$$

$$\boldsymbol{f}^d(\boldsymbol{x}_j, \boldsymbol{v}_j, \boldsymbol{x}_i, \boldsymbol{v}_i) = -\boldsymbol{f}^d(\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{x}_j, \boldsymbol{v}_j) \qquad \text{(A.4)}$$

where k_d represents the spring's damping constant. If equations A.1 and A.3 are

$$\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{x}_j, \boldsymbol{v}_j) = \boldsymbol{f}^s(\boldsymbol{x}_i, \boldsymbol{x}_j) + \boldsymbol{f}^d(\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{x}_j, \boldsymbol{v}_j) \qquad \text{(A.5)}$$

combined, the complete spring force is:

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

$$\boldsymbol{f} = m \times \boldsymbol{a} \qquad \text{(A.6)}$$

$$\boldsymbol{v_{i+1}} = \boldsymbol{v_i} + \frac{1}{2} \times \boldsymbol{a} \times t = \boldsymbol{v_i} + \frac{1}{2} \times \frac{\boldsymbol{f}}{m} \times t \qquad \text{(A.7)}$$

$$\boldsymbol{x_{i+1}} = \boldsymbol{x_i} + \boldsymbol{v_i} \times t \qquad \text{(A.8)}$$

constructed:

7.2 Appendix B – Material properties of various objects

Material	**Young's Modulus (GPa)**	**Poisson's ratio**	**Density (kg/m^3)**
Aluminium (MatWeb 2010a)	68	0.360	2700
Concrete (The Engineering ToolBox [No Date])	14 – 41	0.20 – 0.21	2240 - 2400
Soft steel (MatWeb 2010b)	317	0.220	1900
Hard steel (MatWeb 2010b)	68.9	0.346	3000
Titanium (MatWeb 2010c)	116	0.340	4500

Table 7 – Material properties of various real world objects

7.3 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 VBG 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 performed in Pseudo code 3:

```
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);
```

Pseudo code 3 - Setup a 3D spatial array

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 29 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 4 shows how an index is calculated from the 3D position and pseudo code 4 shows how the position of an index is calculated:

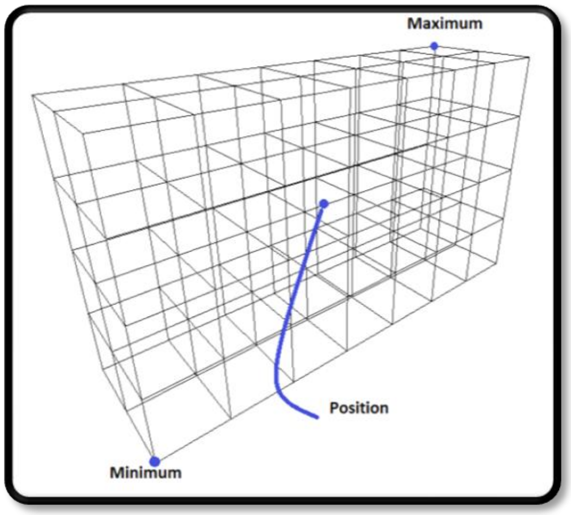

Figure 29 - Grid

```
Vector3D **function** GetPositionOfIndex( Vector3D index){
    Vector3D $halfCellCount$ = Vector3D(grid.width, grid.height, grid.depth) * 0.5;
    return $(index - halfCellCount) * GridCellSize + halfGridCellSize$;
}
```

```
    Matrix $invWorld = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ translation.x & translation.y & translation.z & 1 \end{bmatrix}$;
    Vector3D index = $(pos * invWorld) / (2 * halfGridCellSize)$;
    return index;
}
```

Pseudo code 4 - Get index of position function

Pseudo code 5 – Get position of index function

In Pseudo code 4 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.

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 $[-halfCellCount, halfCellCount]$. 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.

7.4 Appendix D – Volume deformation

This appendix describes the work done when a volume is deformed:

```
foreach surfel $\boldsymbol{S} \in \boldsymbol{AffectedSurfels}$
  S.displacement $= \sum \omega(\boldsymbol{S}, P_i) * P_i.displacement$
endfor

Resample($\boldsymbol{AffectedSurfels}$)

foreach surfel $\boldsymbol{S} \in \boldsymbol{AffectedSurfels} \cup \boldsymbol{NewSurfels}$
  ResetVertexBufferGrid($\boldsymbol{S}$)
  CalculateNeighbours($\boldsymbol{S}$)
  DisplaceSurfels($\boldsymbol{S}$)
  $\boldsymbol{ChangedVertexBufferCells}.insert(\boldsymbol{S}.vertexBufferCell)$
  RegenerateRigidBody($\boldsymbol{S}$)
endfor

foreach $\boldsymbol{cell} \in \boldsymbol{ChangedVertexBufferCells}$
  $\boldsymbol{cell}.createVertexBuffer()$
endfor
```

Pseudo code 6 - Volume deformation

Here each surfel that is affected by the exterior force calculates how much it is displaced by the phyxels. New surfels are created by resampling the affected surfels (shown in Chapter 3.6). For each new and affected surfel the surfel's VBG is updated as well as its neighbourhood awareness. After the neighbourhood has been recalculated, the displacement can be calculated as well. Now the surfel's position will not be changed and therefore the its rigid body can be regenerated. After every surfel has been updated it is time to create the vertex buffer for each VBG cell that has been changed.

8 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-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

Luna, F.D. 2008. *Introduction to 3D Game Programming with DirectX 10*. Plano, TX: Wordware Publishing.

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]

Murchison, J. 2008. *A look at frame rates and game loop history.* [Online]. Avaliable from: http://murcho.alumnaie.net/12/jimi-wants-more-frames-dammit/ [Accessed 11 September 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.

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 modeling and 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]

Oren, M. and Nayar, S.K. 1994. Generalization of Lambert's reflectance model. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. New York: ACM. pp. 239 – 246. [Online]. Available from: http://portal.acm.org/citation.cfm?id=192213 [Accessed 30 June 2010]

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]

Wicke, M., Teschner, M. and Gross, M. 2004. CSG tree rendering for point-sampled objects. In: *Proceedings of the 12th Pacific conference on Computer graphics and Applications, 2004.* Washington: IEE Computer Society. pp 160-168.

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.

9 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]

10 Bibliography