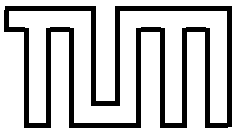

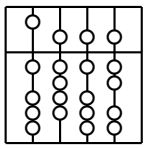

Efficient Realization of Mass-Spring Systems on Graphics Hardware

Realisierung effizienter Feder-Masse-Systeme auf Grafikhardware

Diplomarbeit

Florian Echtler

Technische Universität München

Fakultät für Informatik

Diplomarbeit

Efficient Realization of Mass-Spring Systems on Graphics Hardware

Bearbeiter: Florian Echtler
Aufgabensteller: Prof. Dr. Rüdiger Westermann
Betreuer: Joachim Georgii & Dr. Peter Kipfer

Abgabedatum: 15.9.2004

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

(*Florian Echtler*)

Abstract

Physics-based simulation of deformable objects is a valuable tool for creating realistic and plausible computer graphics. However, even the calculations involved for simple abstractions like mass-spring models are highly demanding, especially when real-time simulation is desired. This diploma thesis explores the possibility to use the powerful vector computation engine present in modern 3D graphics hardware for these calculations and shows a significant performance gain over CPU-based solutions, which in turn allows the use of larger and more detailed models.

Die physikbasierte Simulation von deformierbaren Objekten ist ein wertvolles Werkzeug zur Erschaffung realistischer und überzeugender Computergrafik. Jedoch sind die Berechnungen, die selbst für einfache Abstraktionen wie Feder-Masse-Systeme notwendig sind, sehr zeitaufwendig; insbesondere, wenn eine Simulation in Echtzeit gewünscht ist. Diese Diplomarbeit untersucht die Möglichkeit, die leistungsstarke Vektor-Recheneinheit, die auf modernen 3D-Grafikkarten verfügbar ist, für diese Berechnungen einzusetzen, und zeigt einen signifikanten Geschwindigkeitszuwachs gegenüber einer CPU-basierten Lösung, der wiederum den Einsatz grösserer und detailierterer Modelle erlaubt.

Acknowledgements

First of all, I would like to thank my two supervisors, Joachim Georgii and Dr. Peter Kipfer, for their helpful support and their patience (especially Peter's, who has at times endured loads of visits to his office from me every day).

I would also like to thank Prof. Westermann for his many inspiring hints.

Special thanks go to Andrea Barna (for emotional support ;-)), Thomas Preu and Chris Hodges (for proofreading). Finally, thanks to everyone else in my vicinity who has endured and supported me during the course of this thesis, particularly in the last two months.

Contents

List of Figures

Chapter 1

Introduction

> It's hard to say exactly what constitutes research in the computer world, but as a first approximation, it's software that doesn't have users.
>
> *Paul Graham, Oscon 2004 Keynote*

Let's see if we can prove Mr. Graham wrong..

When creating computer graphics, most of the time the goal is realism. One part of the equation is, of course, the plain visual realism that graces ray-traced images. However, once things start moving, another aspect becomes important - physical plausibility. With increasing computing power available, the simulation of realistic physics becomes feasible.

1.1 Simulation Overview

There exists a multitude of papers regarding the topic of physical simulation. This vast field of research can be partitioned roughly into three separate subtopics: particle systems (fluids/gasses), rigid bodies and deformable objects. This thesis primarily deals with the latter.

Simulating elastic or plastic objects has widely different uses. A significant amount of research is directed towards surgery training in virtual environments, as the simulated tissue should behave as close to the real thing as possible. In figure 1.1, a surgery simulation with tissue pieces is shown.

Entertainment is another possible field of use. Computer games or rendered movies always strive for more realism, and elastic objects or organic beings that move realistically do wonders for plausibility.

Figure 1.1: simulated brain surgery [Pfl04]

Designers, especially fashion designers, also have various uses for simulating flexible materials like cloth. It is a huge advantage to be able to get a first impression of a new piece of clothing without actually having to tailor it.

Of course, the ultimate goal is not only to create realistic motion, but to do it in realtime. Only then will an user be able to properly interact with the simulation.

1.2 Mathematical Methods

The field of deformable object simulation can itself be subdivided according to the mathematical foundations that are used. There are basically two important methods: the finite-element method and mass-spring systems.

In cases where accuracy is premium, the finite-element method, or FEM for short, is widely used. But where speed is of greater importance, simpler models like mass-spring systems have their merits, too. These approaches describe an elastic model as a (likely irregular) mesh of springs connected at

mass points, around which they can freely rotate. Motion and deformation of the entire object are thereby expressed through the relative motion of individual mass points.

Of course, the calculations involved are still time-consuming and require a lot of floating point processing power. A discrete integration method is used to calculate the state of the mass-spring system at a time in the future, based on its past and current states. This requires the calculation and manipulation of force, velocity, acceleration and displacement vectors at a high rate, thereby putting quite a strain on a common CPU when interactive frame rates are desired.

1.3 No CPU Attached

One possibility to take the load off the CPU comes from an unexpected direction. Modern-day graphics hardware, such as ATI's Radeon or Nvidia's GeForce, carries a high-performance graphics processing unit (GPU), which often surpasses the CPU in some performance aspects, like vector calculations. Of course, GPUs were primarily designed with 4-component (red, green, blue, alpha) color vectors and applications like texture mapping in mind, and earlier models, like the famed 3DFX Voodoo card, could indeed not be used for other purposes.

However, with the advent of so-called *shaders*, small assembly programs which are executed directly on the GPU and allow parts of its functionality to be customized, nothing prevents this vector crunching power to be used for the calculation of spatial vectors, as it is necessary for mass-spring simulation. Until recently, one major drawback to this approach has been the fact that the newly calculated position vectors, present in graphics memory, had to be read back by the CPU and re-inserted into the graphics pipeline.

But with the recently presented OpenGL[1] *Superbuffers* extension and its beta implementation by ATI, this performance bottleneck has been erased. Superbuffers allow independent management of memory blocks on the graphics card and thereby make it possible to use the same memory range either as color or spatial vectors without having to read it back through the CPU. In this thesis, the potential of this approach shall be examined.

[1] Open Graphics Library, developed by Sun Microsystems and widely used for 3-dimensional graphics

1.4 Chapter Overview

Chapter 2 - Basics gives an overview about the underlying hardware features and the mathematical and physical foundations.

Chapter 3 - Theory of Operation provides an in-depth explanation of the methods used to map the previously introduced theoretical concepts to the GPU hardware.

Chapter 4 - Class Structure describes the different classes used to implement the simulation and their relationships.

Chapter 5 - Implementation Details discusses miscellaneous aspects of the simulation, e.g. the file formats or the user interface.

Chapter 6 - Performance compares the speed and mathematical stability of this simulation to similar CPU-based solutions.

Chapter 7 - Future Work looks into possible extensions to the simulation, notably collision detection.

Chapter 8 - Conclusion gives a summary of this thesis.

Chapter 2

Basics

2.1 Inside the GPU

From a very basic point of view, a graphics card consists of a certain amount of video memory and a graphics processor, the GPU. The memory holds a representation of the image that is currently displayed on screen (usually called the framebuffer), along with other data like texture images.

Older graphics cards did not have a GPU like today's models do. They did little more that generating a video signal from the contents of the framebuffer and occasionally supported simple operations like copying of rectangular image regions (usually called the 'BitBlt' operation).

However, as three-dimensional graphics started to become widespread, the first GPUs were introduced as a means to relieve the CPU of some computationally intensive tasks like rasterizing triangles into the framebuffer.

The primary purpose of the GPU is therefore the transformation of objects, described through streams of three-dimensional points,[1] into two-dimensional images that are composed of pixels. As mentioned before, GPUs therefore are optimized towards operations on vectors.

The internals of a graphics card consist of three stages:

vertex stage This stage is responsible for the transformation of points, also called vertices, from world coordinates to so-called normalized device coordinates. Basically, this means that the visibility of vertices and geometric primitives with respect to a visible volume is decided. From an abstract point of view, this is accomplished by multiplying the incoming vectors with different matrices, e.g. the modelview and projection

[1] usually grouped into triangles

matrices (see also the famous 'Red Book' [Shr04] for a more detailed explanation).

rasterization The vertices and primitives inside the visible volume now have to be mapped to the pixel raster that will later be displayed on-screen. Often, this process is optimized by mapping to rectangular tiles of varying size instead of single pixels.

pixel stage Finally, the color of the pixels has to be determined. Lots of parameters influence the results of this stage, e.g. fog, lighting or, most notably, textures. A second visibility test, the so-called depth test, also happens in this stage. Although the pixels are mapped to a two-dimensional raster, they still retain a third coordinate that could be interpreted as distance from the screen plane or depth. Usually, only the pixels with the least screen distance are displayed in the end.

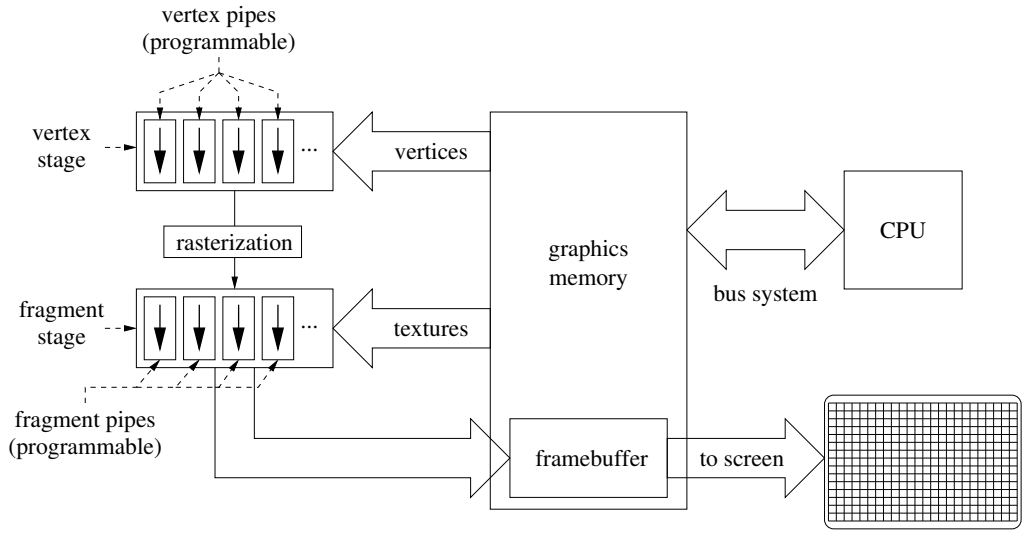

Figure 2.1: graphics card architecture

These stages are easily parallelized into so-called *pipes*, as they operate on streams whose elements do not interfere with each other (see figure 2.1 for a sketched layout of a recent GPU). A GPU therefore operates as a parallel SIMD[2] processor.

A more in-depth description of the analogy between a graphics card and a stream processor has been written by Suresh Venkatasubramanian [Sur04].

[2]Single Instruction, Multiple Data

2.2 Shaders and Superbuffers

Superbuffers, or {Über|Uber}buffers,[3] as they are sometimes called, greatly increase the flexibility of buffer management on the graphics card. In this context, a buffer is a one-, two- or three-dimensional array of vectors, which in turn consist of one to four floating point or integer values. Most commonly, a two-dimensional buffer of four-element vectors is used to represent a color image.

The true power of Superbuffers comes from the fact that they separate the actual hunk of card memory from its access semantics. Graphics libraries did usually not allow such separation - a buffer was allocated for a specific purpose, e.g. as part of a framebuffer or as a texture, and could not be used for other means. For OpenGL, several different approaches were tried to overcome this limitation. However, each only partially solved the problem and was incompatible with the others. This was the main reason for the Superbuffers proposal on the OpenGL Architecture Review Board (ARB).

With a Superbuffer, it becomes possible, for example, to use a buffer first as a framebuffer into which an image is rendered and later apply this image as a texture, without the need to read the image from the framebuffer first and load it into a texture afterwards, a step which usually consumes significant bus bandwidth. Other OpenGL extensions, like Pbuffers, have similar features, but none with the completeness and flexibility of Superbuffers, which even allow a buffer to be treated as an array of spatial coordinates, an option crucial to the approach of this thesis.

Of course, the GPU is by default unable to perform the necessary vector calculations, as its primary purpose is still the generation of color images. This is where the ability to reprogram the GPU with shaders comes in handy. Shaders were introduced in late 2001 and make it possible to change the functionality of two parts of the graphics pipeline, vertex processing and fragment generation (see figure 2.1). It is now possible to rewire the fragment processor to operate on arrays of *spatial* vectors, reading from and writing to appropriately set-up Superbuffers as if they still contained color vectors.

In this regard, it is important to remember that such a shader can be considered to be executed in parallel on each pixel (stream processor analogy). This means that the result vector (regardless of whether it is considered to be in color space or not) can not be dependent upon the result pixel at another

[3] used in the source code

location in the current buffer.

2.3 Physics 101

As mentioned previously, a mass-spring system consists of mass points and connecting springs. The mass points do not prevent the springs from rotating relative to each other. Therefore, the simplest stable element in such a model is the tetrahedron (see figure 2.2), and a larger object is easily composed of connected tetrahedra. It is assumed that the object which is to be simulated is already present in a tetrahedrized format.

One inherent problem of mass-spring systems is that of homogeneity. As algorithms that discretize a given section of space into tetrahedra often tend to generate clusters of small tetrahedra around certain critical points, the mass points are unevenly distributed in the resulting object, as are the edge lengths of the tetrahedra. This results in a object with areas of different density as well as differing stiffness. To simulate a homogeneous material, it should therefore be possible to assign individual masses and stiffness values to the mass points and edges of each tetrahedron. For a detailed description of this method, see [Gel98]. However, widely varying stiffness values might endanger the stability of the simulation for reasons detailed in section 2.4.

Under the influence of external forces, e.g. gravity or collision forces, the object deforms into a configuration where the external forces are compensated by opposing internal forces. In the most basic form of mass-spring model, only the springs themselves exert forces by which they seek to preserve their rest length when compressed or stretched. The resulting force can then be obtained using Hooke's Law

$$\vec{F_{s_{0n}}} = D_s \cdot (|\vec{l_n}| - r_n) \cdot \frac{\vec{l_n}}{|\vec{l_n}|} \tag{2.1}$$

where $\vec{F_{s_{0n}}}$ is the resulting force on point p_0 from spring n, D_s the hardness, $\vec{l_n} = \vec{p_n} - \vec{p_0}$ the distance vector between the two endpoints and r_n the rest length of spring n.

This basic model, however, has a major flaw. When such a tetrahedron is compressed hard enough, it may occur that it 'flips', i.e. one of the points is now on the other side of its opposing triangle. This is a stable state, too, as all springs have the same length as before. But when this tetrahedron is part of a larger model, unrealistic deformations may become visible.

To make this model more realistic, other forces can be introduced. Michael

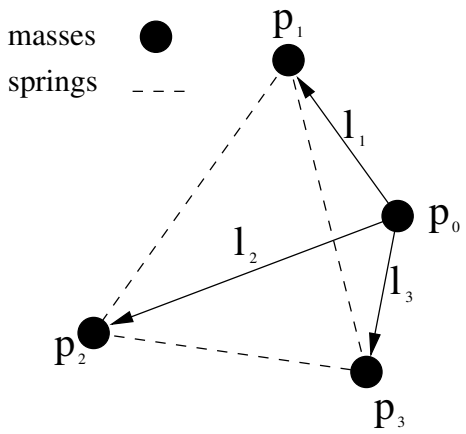

Figure 2.2: mass-spring tetrahedron

Teschner [Tes04] has shown that applying additional forces to preserve the original volume of a tetrahedron improves realism significantly, as it is now much harder to flip a tetrahedron. In such a case, the volume would become negative and a large force would push the tetrahedron back towards its initial state. This force can be calculated similar to the spring forces as

$$\vec{F_v} = D_v \cdot (v - v_0) \cdot \frac{\vec{n}}{|\vec{n}|} \tag{2.2}$$

where $\vec{F_v}$ is again the resulting force on p_0, D_v is the 'volume hardness', $\vec{n} = (\vec{l_2} - \vec{l_1}) \times (\vec{l_3} - \vec{l_1})$ is the normal vector of the opposing triangle (drawn dashed in figure 2.2), $v = \frac{1}{6}(l_1 \times l_2) \cdot l_3$ the current volume of the tetrahedron and v_0 its original volume.

The total force acting upon one point x of a larger model can then be calculated as

$$\vec{F_x} = \sum_{i=1}^{n}(\sum_{j=1}^{3} \vec{F_{s_{xj}}} + \vec{F_v}) + \vec{F}_{friction} \tag{2.3}$$

with n being the number of tetrahedra this point is part of, $\vec{F}_{friction} = \mu\vec{v}$ as a velocity-dependent friction force (the friction coefficient μ is usually negative, so the friction force is directed opposite to the velocity vector), $F_{s_{xj}}$ as the force of the j-th spring of the current tetrahedron (see equation 2.1) and F_v as the volume preserving force of the current tetrahedron (see equation 2.2). When an edge is shared between several tetrahedra, it is considered to contain several springs. The spring force is therefore calculated multiple times. While this may seem redundant, it is consistent with the approach of

viewing tetrahedra as atomar building blocks of the object.[4]

Now, as this total force is known, the acceleration of the mass point can be calculated using Newton's Laws of Motion, specifically No. 2:

$$\begin{aligned} \vec{F} &= m \cdot \vec{a} \\ \vec{a} &= \frac{\vec{F}}{m} \end{aligned} \tag{2.4}$$

From now on, the vectors $\vec{A}$, $\vec{V}$ and $\vec{X}$ will be used. These vectors simply represent a combination of all n three-dimensional acceleration, velocity and position vectors and have therefore $3n$ components each. They are also functions of time:

$$\begin{aligned} \vec{X} &= \vec{X}(t) \\ \vec{V} &= \vec{V}(t) = \dot{\vec{X}}(t) \\ \vec{A} &= \vec{A}(t) = \vec{A}(\vec{X}(t), \vec{V}(t)) \end{aligned} \tag{2.5}$$

As velocity and acceleration are but the first and second time derivatives of the position, the motion of the entire mass-spring system can now be described by the following second-order differential equation:

$$\ddot{\vec{X}}(t) = \vec{A}(\vec{X}(t), \dot{\vec{X}}(t)) = \frac{1}{m}\vec{F}(\vec{X}(t), \vec{V}(t)) \tag{2.6}$$

[4]Moreover, some of the properties of the hardware require this approach, which will be explained later in more detail.

2.4 Math 101

The problem that arises next is, of course, that of efficiently finding a solution[5] to equation 2.6.

Various numerical methods for solving differential equations systems exist. However, in this case, it is not desirable to obtain a solution,[6] which corresponds to a stable configuration of the object with all forces in equilibrium, in one expensive calculation step.

As an interactive simulation is desired in this thesis, the deformation of the object towards a stable state should, in an ideal case, be shown in real-time, and the user should be able to interact with the process, thereby influencing the final state.

As mentioned previously, the finite-element method is often used for this problem. This approach tries to approximate the position through an unknown function that is assumed to be the solution to a differential motion equation similar to eq. 2.6. The approximation is composed from base functions that are derived by discretizing the space upon which the unknown function is defined. For more detailed information about finite elements, see 'Das kleine Finite-Elemente-Skript' by Ansgar Jüngel [Jue04]. Unfortunately, these methods are quite complex and their implementation on a GPU is well beyond the scope of this thesis.

Several other numerical integration methods are available for this task, which advance the simulation by calculating successive 'snapshots' of its state. Unfortunately, they all have one limiting property in common: the time difference between two consecutive integration steps (timestep) must not exceed a certain upper bound, which depends on the constants in the differential equation system and the integration method used. Otherwise, the results become highly unstable, which usually causes the object to end its life in an explosion (see figure 2.3). When such a case occurs, the timestep must be reduced, thereby increasing the precision.

Some of these methods, especially Euler and related ones, are limited in respect to that they are only able to solve first-order differential equations. However, this is not a problem as equation 2.6 can also be expressed as a two-dimensional differential equation of first order:

[5] or an approximate solution

[6] often, several solutions exist

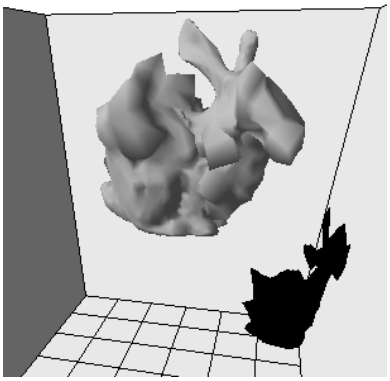

Figure 2.3: exploding bunny

$$\begin{aligned} y &:= \begin{pmatrix} \vec{X} \\ \dot{\vec{X}} \end{pmatrix} = \begin{pmatrix} \vec{X} \\ \vec{V} \end{pmatrix} \\ \dot{y} &= f(y) := \begin{pmatrix} \vec{V} \\ \vec{A}(\vec{X}, \vec{V}) \end{pmatrix} \end{aligned} \tag{2.7}$$

With regard to numerical integration methods, it should be noted that these methods generally exist in an implicit and an explicit version. The difference between these two kinds will now be detailed:

Assume we have a differential equation of the form $y'(x) = f(x, y(x))$.

When examining, for example, the basic explicit Euler method, the equation for the solution in step $i+1$ is $y_{i+1} = y_i + hf(x_i, y_i)$ where h is equivalent to the timestep mentioned above. Solving this is quite straightforward.

For the implicit Euler method, however, the solution is $y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$. Obviously, it is necessary to solve a (possibly non-linear) equation that is dependent on the function f to acquire a solution for y_{i+1}.

While the implicit methods are generally much more accurate, they therefore are also exceedingly more complex to calculate. For a more in-depth explanation, see the course notes by Bernd Simeon [Sim04].

Among the integration methods considered for this thesis were the method by Verlet [Ver67], which is based on the Euler method, and the well-known

Runge-Kutta methods, which are listed here in order of ascending precision. In this context, increased precision also means that larger timesteps are possible without the simulation becoming unstable. However, this comes at a price as the calculations involved become gradually more complex. Runge-Kutta methods, for instance, require several evaluations of the right side of the equation, which is an expensive operation.

In the paper from Michael Teschner [Tes04], the question for the most performant method is examined from a quite clever point of view - the quotient between duration of the calculation for one integration step and largest possible timestep is compared.

method	timestep [ms]	comp. time [ms]	ratio [7]
Verlet	3.1	7.3	0.427
Leap-frog	3.1	7.3	0.426
Runge-Kutta 2nd Order	4.9	14.3	0.342
velocity Verlet	2.5	7.3	0.341
Beeman	2.5	7.4	0.337
Heun	4.2	18.4	0.229
explicit Euler	1.5	7.3	0.205
Runge-Kutta 4th Order	6.3	33.0	0.191

algorithm comparision table (taken from [Tes04])

A larger value of this quotient indicates a better performance of the algorithm, and surprisingly, the simple Verlet method (a modified Euler method) performs best.

Keeping these discoveries in mind, let's now tackle equation 2.6 again, making the following assumptions:

- only a short time interval of length Δt, starting at t_0 and ending at $t_1 = t_0 + \Delta t$ is observed
- the vectors $\vec{A}(t_0)$ and $\vec{X}(t_0)$ are known as well as $\vec{X}(t_0 - \Delta t)$
- the acceleration vector is assumed to remain constant in that interval, as no further information is available

To simulate the motion of the mass point, it is now necessary to calculate $\vec{X}(t_0 + \Delta t)$ and $\vec{A}(t_0 + \Delta t)$ from $\vec{X}(t_0)$ and $\vec{A}(t_0)$. As initial values for $t = 0$,

[7] $ratio = \frac{timestep}{computationtime}$

a known position vector $\vec{X}(0) = \vec{X}_0$, a velocity vector $\vec{V}(0) = \vec{0}$ and an acceleration vector equal to the gravity vector $\vec{A}(0) = \vec{g}$ are assumed. First, the acceleration vector is integrated over Δt to obtain the new velocity vector:

$$\vec{V}(t_0 + \Delta t) = \int_{t_0}^{t_1} \vec{A}(t)dt + \vec{V}(t_0) = \vec{A}(t_0)\Delta t + \vec{V}(t_0) \tag{2.8}$$

A second integration step yields the position of the mass point at time t_1. However, even though $\vec{A}(t)$ is considered constant,

$$\vec{V}(t) = (t - t_0) \cdot \vec{A}(t_0) + \vec{V}(t_0) \tag{2.9}$$

now changes linearly between t_0 and t_1. The new position is therefore

$$\begin{aligned} \vec{X}(t_0 + \Delta t) &= \int_{t_0}^{t_1} \vec{V}(t)dt + \vec{X}(t_0) \\ &= \int_{t_0}^{t_1} (t - t_0) \cdot \vec{A}(t_0) + \vec{V}(t_0)dt + \vec{X}(t_0) \\ &= \tfrac{1}{2}\vec{A}(t_0)\Delta t^2 + \vec{V}(t_0)\Delta t + \vec{X}(t_0) \end{aligned} \tag{2.10}$$

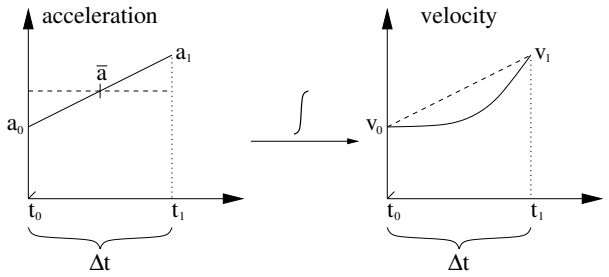

Figure 2.4: velocity approximation

The velocity $\vec{V}(t_0)$ remains as the last unknown in this equation. This unknown variable now has to be re-expressed by means of other, known variables. As the velocity at any given time depends on the previous velocity and acceleration, the following approximation is appropriate (see figure 2.4 - for a time difference of Δt, the mean acceleration value can be integrated for equal results), assuming that the acceleration changes linearly in the previous time interval:[8]

$$\vec{V}(t_0) = \vec{V}(t_0 - \Delta t) + \frac{1}{2}(\vec{A}(t_0 - \Delta t) + \vec{A}(t_0))\Delta t \tag{2.11}$$

[8]Of course, for the current interval, the acceleration was presumed to be constant. However, when we assume for the moment that more information is available about the previous interval, a refined approximation can be used.

By inserting this into equation 2.10 (and, to reduce notational clutter, assuming $t_{-1} = t_0 - \Delta t$), we get:

$$\begin{aligned} \vec{X}(t_1) &= \tfrac{1}{2}\vec{A}(t_0)\Delta t^2 + \Delta t(\vec{V}(t_{-1}) + \tfrac{1}{2}(\vec{A}(t_{-1}) + \vec{A}(t_0))\Delta t) + \vec{X}(t_0) \\ &= \vec{A}(t_0)\Delta t^2 + \tfrac{1}{2}\vec{A}(t_{-1})\Delta t^2 + \vec{V}(t_{-1})\Delta t + \vec{X}(t_0) \\ &= \vec{A}(t_0)\Delta t^2 + \tfrac{1}{2}\vec{A}(t_{-1})\Delta t^2 + \vec{V}(t_{-1})\Delta t + \vec{X}(t_{-1}) - \vec{X}(t_{-1}) + \vec{X}(t_0) \end{aligned} \tag{2.12}$$

The second, third and fourth component of this equation can now be re-expressed as $\vec{X}(t_0)$ (see also equation 2.10).

$$\vec{X}(t_0 + \Delta t) = \vec{A}(t_0)\Delta t^2 + \vec{X}(t_0) - \vec{X}(t_{-1}) + \vec{X}(t_0) \tag{2.13}$$

All put together, the result now looks as follows:

$$\vec{X}(t_0 + \Delta t) = \vec{A}(t_0)\Delta t^2 + 2\vec{X}(t_0) - \vec{X}(t_0 - \Delta t) \tag{2.14}$$

This is exactly the formula known as Verlet integration, developed by Loup Verlet [Ver67] and cited, e.g., by Thomas Jakobsen [Jak01] and countless others in the simulation world. The above approach has the advantage of showing how the Verlet equation can be derived from Newton's laws.

Verlet himself, however, has derived this formula using Taylor series. This approach has the advantage of also yielding an upper error boundary. The position, $\vec{X}(t)$, is expanded both forward and backward in time ($\vec{B}(t)$ is the third time derivative of $\vec{X}(t)$):

$$\begin{aligned} \vec{X}(t + \Delta t) &= \vec{X}(t) + \vec{V}(t)\Delta t + \tfrac{1}{2}\vec{A}(t)\Delta t^2 + \tfrac{1}{6}\vec{B}(t)\Delta t^3 + O(\Delta t^4) \\ \vec{X}(t - \Delta t) &= \vec{X}(t) - \vec{V}(t)\Delta t + \tfrac{1}{2}\vec{A}(t)\Delta t^2 - \tfrac{1}{6}\vec{B}(t)\Delta t^3 + O(\Delta t^4) \end{aligned} \tag{2.15}$$

Adding the two expressions results in

$$\vec{X}(t + \Delta t) = 2\vec{X}(t) - \vec{X}(t - \Delta t) + \vec{A}(t)\Delta t^2 + O(\Delta t^4) \tag{2.16}$$

which is equivalent to equation 2.14, except for the upper error boundary $O(\Delta t^4)$.

It is worth noting that the acceleration vector is assumed to be known, though the velocity vector is not. The velocity is simply not needed for the calculation of new positions, however, as it influences the friction force and is therefore necessary to determine the final acceleration, it needs to be explicitly calculated. This can simply be done by assuming $\vec{V}(t_0) =$

$\frac{1}{\Delta t}(\vec{X}(t_0) - \vec{X}(t_0 - \Delta t))$.[9] A variant of this approach is also known as 'velocity Verlet', as presented by Swope [Swo82] (for a comparision, see also figure 2.5).

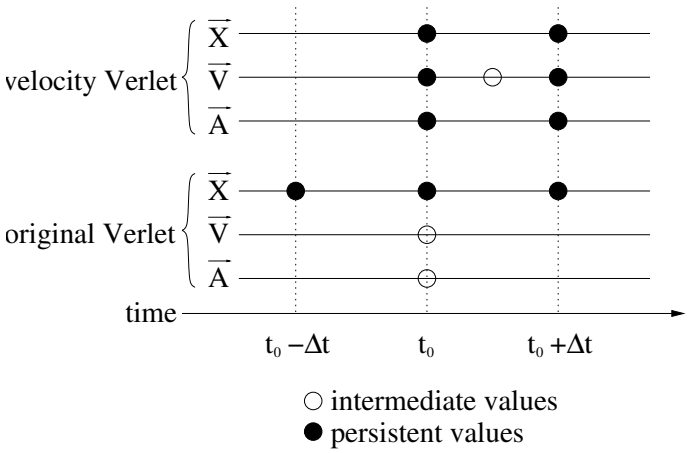

Figure 2.5: Verlet integration

While the velocity Verlet method has the advantage of a more precise velocity calculation, it also has the severe disadvantage of requiring the calculation *and* storage of the acceleration, velocity and position vectors for the next timestep. In contrast, the original Verlet method requires only an acceleration vector and the current and previous position vectors. This has the advantage that the position vectors can be stored in a ring buffer, so that almost no effort save a pointer swap is necessary to obtain the current and previous values.

[9] This differs from equation 2.11, but as only $\vec{X}(t_0)$ and $\vec{X}(t_0 - \Delta t))$ are available at this point in time, this simpler approximation must be sufficient.

Chapter 3

Theory of Operation

3.1 Representation of Vectors

As powerful as the graphics hardware is in terms of vector processing, it still does not offer the same flexibility one is used to from CPU-based programming. Even when using fragment and vertex shaders, one can not reprogram the entire GPU (see figure 2.1). For example, rasterization still occurs between vertex and fragment processing, which makes it necessary to operate on pixels as basic data element. Fortunately, the ATI systems used in this thesis support pixels consisting of four floating-point values, which fits nicely with three-dimensional spatial vectors and leaves room for an additional value.

As stated previously, the algorithm needs to operate on all vectors of one type simultaneously, e.g. on the vector $\vec{X}$ which consists of all n vectors $\vec{x}$ and has dimension $3n$. The first possible representation that comes to mind is, of course, a one-dimensional concatenation of pixels. However, as the 'pixel vector' has to be used as a texture to be read by the fragment shader, it soon hits the usual texture size limit of 2048 texels.[1]

The obvious solution is to go two-dimensional, as the texture units are optimized towards two-dimensional textures anyway. Now, the maximum texture size is 2048^2 texels, which is sufficient for slightly more than four millions of vectors. The width and height of a texture are required to be powers of two, which has the side effect that usually the texture is a bit larger than the actual vector and contains unused texels which are updated with new values in every step, but do not contribute to any other part of the calculation.

[1] texture pixels

3.2 Vector Operations

Any mathematical operation which involves basic vector arithmetic can now, thanks to the capabilities of Superbuffers (see also section 2.2), simply be executed by binding[2] an appropriate shader and binding[3] the buffers which represent the involved vectors as textures. The vector receiving the end result is then bound as render target[4] and the actual calculation is triggered by simply rendering one rectangle (in OpenGL, a `GL_QUAD`) which exactly fills the target buffer. This causes the fragment shader to be executed once for every target pixel/vector and its output to be written to the target buffer. If the previous value of the result vector is one of the input variables, it is necessary to use an intermediate buffer to store the result and afterwards swap the buffer pointers, as the current OpenGL implementation does not allow the same physical Superbuffer to be used for reading and writing.

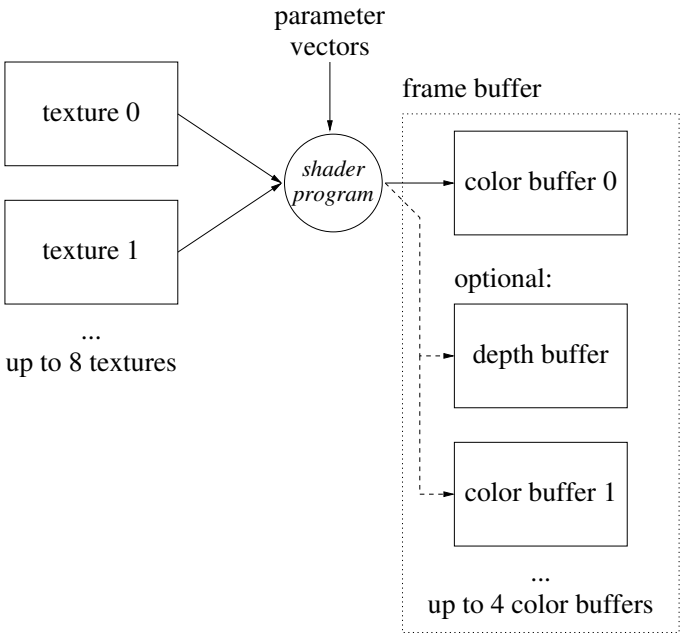

Figure 3.1: operation of a fragment shader

[2] preparing for execution
[3] attaching to the rendering pipeline
[4] a rectangular pixel buffer which receives the final output from the render pipeline

3.3 Topology Storage and Force Calculation

As described in Section 2.3, the topology of the simulated object is stored as a set of connected tetrahedra. A tetrahedron consists of a total of six springs, and if a certain edge within the object is shared between several tetrahedra, it will consist of several independent springs. This does create some redundancy, as certain edge lengths and spring forces will be calculated several times, but is necessary for a tetrahedron-based data model. To calculate the force acting on one mass point, it is therefore necessary to sum up the forces which the individual tetrahedra that this point is part of generate.

A straightforward approach would take the coordinates of the four corner points of the tetrahedron and, by using their distances, the volume and the respective rest and hardness values, calculate a force vector for every point, which is afterwards summed up by looping over all points.

However, it now becomes necessary to take certain peculiarities of the graphics hardware itself into account. As all GPUs which can be used for this thesis (Radeon 9700/9800) have a 128 bit shader-to-memory interface, only one result value from the fragment shader (which is a 4-component float vector and has therefore a size of $4 \cdot 4$ bytes = 16 bytes = 128 bits) can be written to memory at a time. A shader is allowed to write more than one result value, but this will actually cause the calculation to be performed multiple times, once for every result, thereby greatly increasing runtime.

Another potential performance sink is the so-called dependent fetch, which denotes a read operation from a texture that is based on coordinates which were calculated from an earlier texture read. This prevents the GPU from issuing all texture fetches in parallel and will degrade performance. It is therefore desirable to have as few dependent fetches as possible.

When we look at the aforementioned approach now, it is easy to see that it will require 4 (internal) rendering passes, each doing 4 dependent fetches, as the shader program will first have to read the indices of the four corner points out of one texture and then lookup the actual coordinates in another texture. This makes for a total of 16 dependent fetches per tetrahedron and simulation step.

However, this can easily be improved. When the force calculation is not executed per tetrahedron, but per mass point, only three dependent fetches are needed per point, making for a total of 12 such operations per tetrahedron and simulation step, because the index of one of the points (the one currently being looked at) is already known and does not need to be

fetched. In pseudocode, this looks as follows:

```
for every point p0
  for every tetrahedron t incident on p0

    get spring hardness hs, volume hardness hv
    get rest volume v, rest spring lengths l1-3
    get indices i1-3 of other corners of t

    // three dependent fetches follow
    get coordinates of p1-3 through i1-3
    get coordinates of p0

    calculate force on p0
    add to total force on p0

  end for
end for
```

The question that arises next is that of efficient storage. While it might seem straightforward to treat the entire topology information as a large $n \cdot n$ matrix with which the position vector is then multiplied to gain the force vector, this approach would fail to work for two reasons. First, the equation contains differences between the positions of neighbouring points (see equation 2.1), which introduces nonlinearities that can not be solved by matrix multiplication alone. Second, n vectors of size n would be required for storage, and as each vector occupies one buffer, this would require a huge amount of buffer space, far more than any graphics card can offer.

However, if every non-zero entry in the matrix is stored along with appropriate indices to the vectors it needs to be multiplied with,[5] the number of necessary vectors (and thus buffers) is reduced to the maximum number of non-zero vectors in one row of the matrix, which is equivalent to the maximum point valence with respect to incident tetrahedra. This number varies from object to object, but it is usually at least 30 times smaller than the full adjacency matrix.

Figure 3.2 shows how this data object is realized as a two-dimensional grid of buffers. Each horizontal 'slice' consists of three buffers. This is necessary because a total of 12 float values is needed to completely describe a

[5] basically, an incidence list

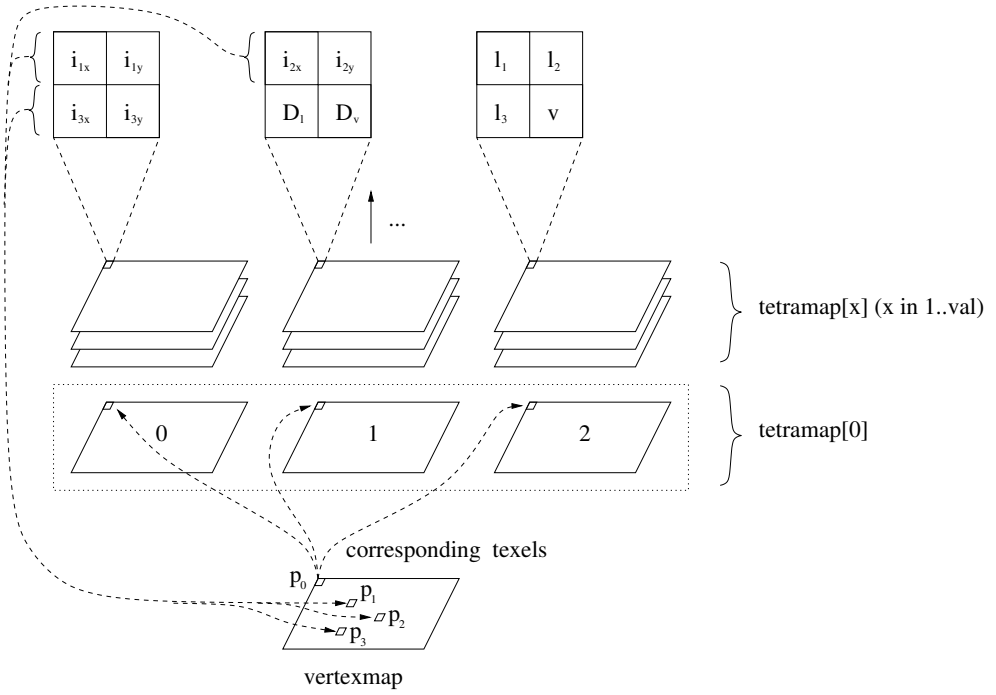

Figure 3.2: matrix texture stack

tetrahedron from the viewpoint of one of its corner points:

i_1, i_2, i_3:	indices of the other points in the form of (x,y) texture coordinates
l_1, l_2, l_3:	rest length of the springs
v:	rest volume of the tetrahedron
D_l:	spring hardness
D_v:	volume hardness

This means that in one given slice, the texels at a certain (x,y) position in each of the three buffers together describe a tetrahedron. This tetrahedron is incident to the mass point whose location is stored at the same position in the vertex map. This also means that the data for every tetrahedron is stored at four different points in this structure, once for every corner point. As a reference, see also figure 2.2.

It might be possible to compress the indices into one float each. However, as all textures used here are two-dimensional, the indices would have to be re-expanded at runtime. While this could potentially save one texture column in the stack, the additional computation overhead would far outweigh the smaller memory footprint.

One could also try to optimize the calculation by first determining all edge lengths and using them to calculate the forces afterwards. This approach would have the advantage of removing the redundancy of calculating the length of some edges multiple times, however, it would also introduce another level of indirection (positions → edge lengths → forces). Moreover, in addition to the two dependent fetches necessary to calculate each edge length, each tetrahedron would now again require a total of $4 \cdot 4 = 16$ dependent fetches for all force calculations.

The number of slices, *val*, should ideally be chosen large enough to store every tetrahedron, that is, equal to the largest point valence; however, as each slice also requires an additional expensive rendering pass, good results can still be obtained by setting *val* lower so that some tetrahedra connected to the highest-valence points get clipped, but the majority of the lower-valence points has still enough storage space for all incident tetrahedra left.

Unfortunately, this optimized data structure is still rather sparsely populated and contains roughly 80% of zero entries, i.e. entries referring to non-existent tetrahedra. The spring and volume hardness are set to zero so that these entries do not generate any phantom force, however, they still need to be calculated and consume lots of processing time.

3.4 Early-Z Test Optimization

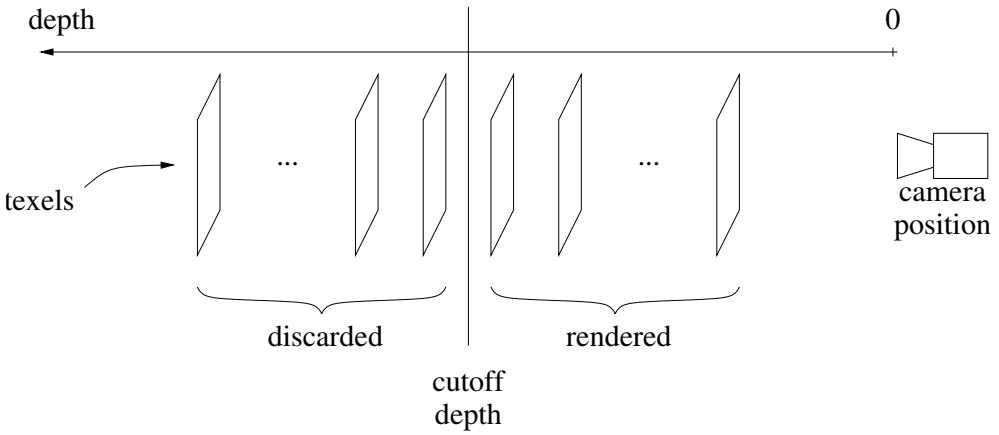

Figure 3.3: depth test optimization

The depth buffer provides an elegant solution to this problem, which is

illustrated in figure 3.3. Normally, the depth buffer is used to discard pixels on the screen that are hidden behind other pixels that are closer to the viewport. Here, it can be used to discard the unused texels in the tetrahedron stack.

The relevant tetrahedra are written into the 'lower' part of the stack, while the zero entries occupy the 'higher' part. This means that for every point, there exists now a certain cutoff depth in the texture stack at which the irrelevant entries start.

During initialization, this value is written into a depth buffer template at the appropriate pixel location. The depth buffer is then used during the force calculation process to determine the exact depth at which rendering should stop for each target pixel. To achieve this, each slice is rendered slightly more distant from the camera (this does not matter regarding the projection into the render target, as the projection is orthogonal anyway). However, the texels in every slice that lie beyond the cutoff depth are not rendered now. This is known as 'early z-test'[6] because it occurs *before* the expensive parts of rendering, such as the execution of the fragment program, and therefore provides a drastic speed-up that will be detailed in chapter 6.

3.5 Surface Rendering

To allow for efficient rendering of the object surface,[7] it is necessary that those triangles that form the surface are flagged appropriately in the object description. To render the triangulated surface with OpenGL, a list of index triplets is needed. These indices consist simply of the vertex number and are internally converted into two-dimensional texture coordinates by the GPU. For best performance, it is desirable that this list is also stored on the graphics card. At this point, the `ShortUberBuffer`, which will be introduced below, comes in handy, as it stores 16-bit `GL_SHORT` integer values that can directly be used by the `glDrawElementArrayATI` when such a buffer has been bound via `bind_array(GL_ELEMENT_ARRAY_ATI,1);`.

It is worth noting that although the buffer stores four-component vectors while a triangle needs only three indices, no space is wasted, as OpenGL correctly handles the case when the indices are packed tightly, i.e. when four triangle index triplets are stored in three texels.[8]

[6] available on most modern GPUs

[7] done via the `Simulation::draw_surface` method, see below

[8] This is also indicated by the parameter 1 in the above `bind_array` call.

3.5.1 Normal Vectors

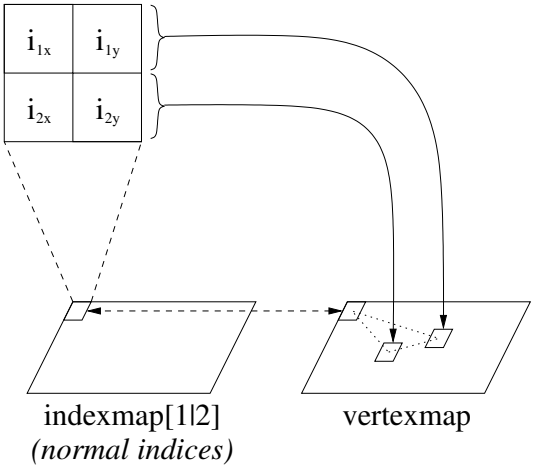

Figure 3.4: calculation of triangle normals

For a correctly rendered and lit surface, however, normal vectors are necessary, too. As the object likely will be deformed during the runtime of the simulation, it is unavoidable that the current normal vectors are re-calculated before the surface is rendered, at least if the simulation has proceeded in time since the last update.

The most accurate way to calculate the surface normal of a given surface vertex would be to first calculate the normal vectors of all triangles that are incident on this vertex, and then form the mean value of these vectors to receive the precise surface normal.

However, as the number of incident triangles varies greatly, this would again require an elaborate data structure and multiple passes to generate the exact normal vectors. Fortunately, an easy solution to this problem exists. To the human eye, there is no noticeable difference in the shading of the surface, regardless whether the normal vectors have been calculated from all relevant triangles or only two or maybe even only one per vertex.

The most simple solution would therefore be to just use the normal of one neighboring triangle for each vertex that is arbitrarily chosen when the object is loaded. While this approach requires only little computation, the surface of the object seems to vibrate when the object is in motion, especially during collisions or user interaction. This is quite easily solved by taking not one, but two arbitrary and different neighboring triangles and calculating the normalized mean value of these two normal vectors.

Again, the indices of the neighboring points of each triangle are stored as (x,y) texture coordinates in two index maps at the same location where the surface vertex in question is stored in the vertex map. This relationship is illustrated in figure 3.4.

3.5.2 Force Visualization

It is sometimes desirable to get a rough overview about which parts of an object are currently stressed by forces and which are not. The simple mass-spring method used here should not be used for precise stress calculations anyway, so a rough visualization is sufficient. As the total force upon a certain point is already available in a per-vertex map, it is easy to use the absolute length of the force vector as an interpolation value between a base color (gray) and a force color (red), which has the overall effect that stressed parts of the object light up in red.

3.6 Buffer Overview

This section gives a summary of the different buffers used in this thesis, their relations and the data stored within. The buffers can be partially empty, i.e. contain texels that are not used in any calculation.

`vertexmap/-old` stores the position of a vertex in the (x,y,z) components and its mass in the w component of the texel. `vertexmap` contains the current position, and `vertexold` the position in the previous step.

`forcemap` contains a force vector in the (x,y,z) components that is currently acting on the vertex stored at the same texture position. The w component is unused.

`depthmap/-img` contains the depth values that are used for optimization of the force calculation. `depthimg` is a floating point buffer that stores the depth value in the z component, while `depthmap` is the actual depth buffer that is updated with the values from `depthimg` before every step.

`normalmap` contains the normal vectors of the object surface in the (x,y,z) components. For those vertices that are not part of the surface, these vectors do not contain sensible data.

`indexmap[1|2]` contain indices into `vertexmap` in the form of texture coordinates. Each texel contains two indices, stored in (x,y) and (z,w). The indexed vertices form a triangle together with the third vertex that

is stored at the same coordinates as this texel. This data is used to calculate surface normals.

`colormap` contains optional colors for the object surface with RGB values stored in the (x,y,z) components. The alpha value is unused.

`trimap` contains the indices into `vertexmap` describing the surface triangles themselves. This information is already stored in `indexmap[1|2]` to a certain degree. However, it also needs to be stored in the form of single integer indices, as this format is required by OpenGL to draw a triangle list.

`tetramap` has already been extensively described in section 3.3.

3.7 Pseudocode Algorithm

The following piece of pseudocode should provide an overview over the inner workings of the simulation. The next section will explain each step in a more detailed manner.

initialization:

```
for every tetrahedron layer t
  for every point p

  if t[p] is valid
    get depth d of t
    store d in depth[p]
  end if

end for
```

simulation step:

```
// force calculation
for every tetrahedron layer t
  for every point p

    get depth d of t
    if d < depth[p]
      calculate force on p (see above)
      add to totalforce[p]
    end if

  end for
end for

// integration step
for every point p

  calculate velocity v from position[p], old_position[p]
  calculate acceleration a from totalforce[p], mass[p], v
  calculate displacement d from a, position[p], old_position[p]

  old_position[p] = position[p] + d

  if old_position[p] outside wall
    adjust old_position[p]
  end if

end for

swap position with old_position
```

surface rendering:

```
if (current_step % 5) == 0

  for every point p
    if p is surface point

      get indices i1, i2 of first triangle neighbors
      get positions p1, p2 of neighbors through i1, i2
      calculate n1 from p, p1, p2

      get indices i3, i4 of second triangle neighbors
      get positions p3, p4 of neighbors through i3, i4
      calculate n2 from p, p3, p4

      calculate mean normal n from n1,n2
      store n in normal[p]

      get force f from totalforce[p]
      calculate color c from f
      store c in color[p]

    end if
  end for

  draw surface with triangles, positions

end if
```

3.8 Data Flow

To conclude this chapter, this section illustrates the flow of point and vector data through the different shaders and auxiliary buffers. In the initialization step and in step 2 of the simulation itself, the data in the target buffer is accumulated over several rendering passes, i.e. the target buffer from pass n is reused as an additional input buffer in pass $n+1$, and a temporary buffer is used as render target (arrows marked with 1). Afterwards, the buffers are swapped for the next pass (arrows marked with 2).

3.8.1 Initialization

After a new model has been loaded, the depth buffer, as described in section 3.4, has to be recalculated. This is done with the *depth* shader and multiple rendering passes, one for each slice of `tetramap`. This shader is configured to pass through the depth value produced by rasterization plus a small offset as long as the entries in `tetramap` are valid. This offset is necessary to assure that the final force sum really consists of *all* forces. As the *collect* shader (see below) uses two force buffers that are swapped in every step, one additional pass is necessary to make sure that the final result contains the force from the uppermost tetrahedron in the stack.

As soon as an empty entry is encountered, the depth value from the previous pass is used and, as all empty entries are stored together, does not change anymore over the remaining passes. This value is then used as the cutoff depth during each iteration of the simulation, which is described in the following paragraphs.

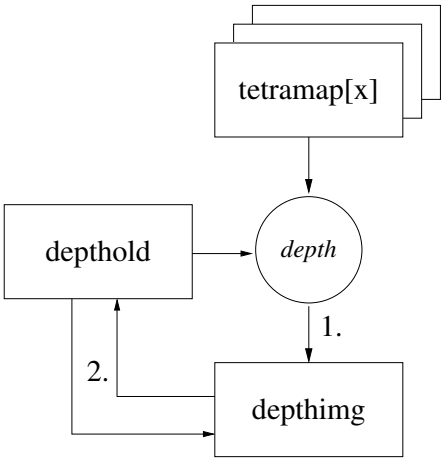

Figure 3.5: one pass of the initialization process

3.8.2 Step 1 - Load Buffers

The `forcemap` and `depthmap` buffers are initialized. It is at first not obvious why it is necessary to re-initialize the depth buffer before each iteration, as it is not written to. However, a bug in the current Superbuffer implementation prevents any other depth buffer than the pre-allocated window system depth buffer from being used, which has the side effect that when rendering the scene, the depth buffer contents are destroyed.

The forcemap is first initialized with the value of the `external_force` parameter. If the object is currently pushed by the user, the initial force is also dependent on the vertex position and the mouse parameters.

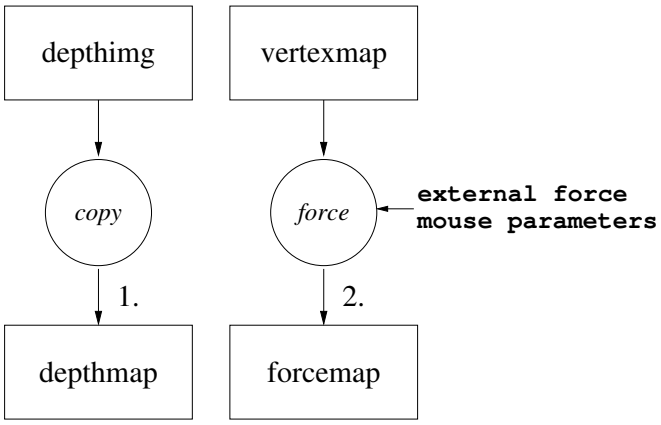

Figure 3.6: simulation step 1

3.8.3 Step 2 - Calculate Forces and Plastic Deformation

This is the most performance-intensive part of the simulation. In this step, multiple rendering passes are used to calculate the total force acting upon each vertex. In each pass, one slice of `tetramap` is processed. The vertex positions, which are necessary for calculation the spring and volume forces, are read from `vertexmap`, while `depthmap` is used to determine the exact cut-off depth. Additionally, the force value from the previous pass is read from a temporary buffer and added to the new output value. This is necessary because floating point buffers do not support blending.

As the *collect* shader is aware of the magnitude of every force acting inside the object, it can optionally be used to calculate plastic deformation of the object. This means, basically, that the rest length and volume values inside of `tetramap` have to be changed. If any of the springs in a tetrahedron exceeds its maximum allowed force (see also section 5.1, the springs rest length is adjusted to the current length. Additionally, if any of the springs in a tetrahedron changes, the rest volume is also set to the current volume, so that the current shape of the tetrahedron becomes its new rest shape.

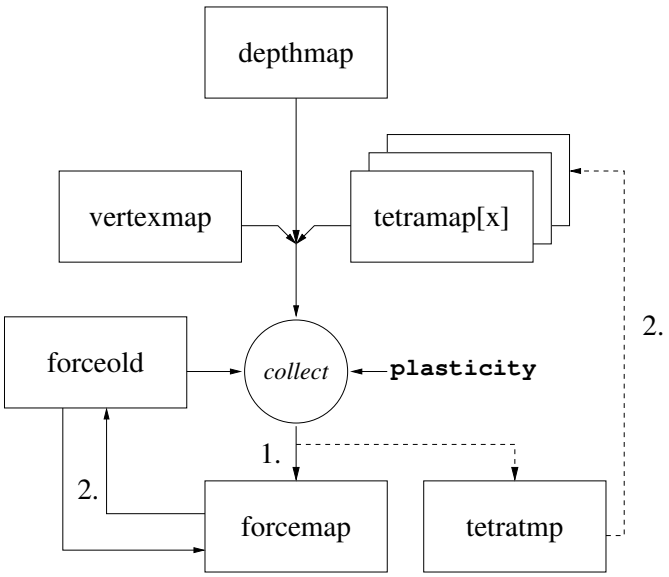

Figure 3.7: one pass of simulation step 2

3.8.4 Step 3 - Perform Integration Step and Collision Detection

This step performs several different calculations. It determines the actual acceleration from the force vector and mass of a vertex[9] and uses this value, along with the current (`vertexmap`) and previous (`vertexold`) position of the vertex, to calculate the future position using Verlet integration.

If the new position does in any direction exceed the bounds provided by the world box, the offending coordinate is clipped to the boundary position. This is only an approximation, as the exact collision point would be the intersection of the motion vector with the clipping plane. However, as shaders are limited by a maximum instruction count, the simpler approximation was chosen.

As the current position must be preserved to be used in the next iteration, the three vertex buffers are now rotated. The newly calculated vectors become the current vertex position, while the contents of `vertexmap` are shifted to `vertexold`. The contents of `vertexold` are moved to `vertexnew` and thereby effectively discarded, as `vertexnew` will be overwritten in step 3 of the next iteration. This step does not actually move buffer data around, as all these operations can also be achieved through pointer swapping.

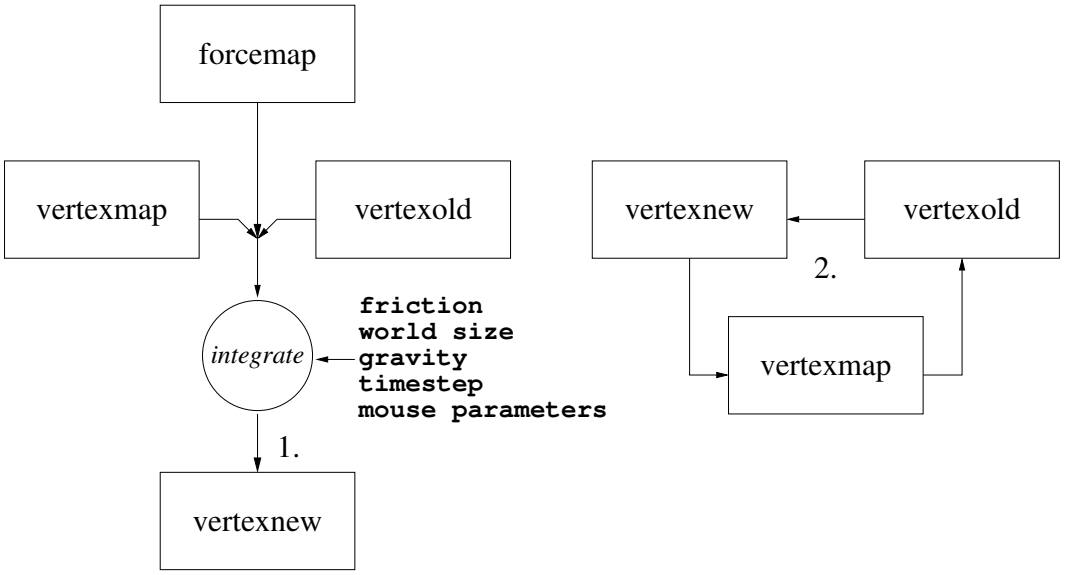

Figure 3.8: simulation step 3

[9] also read from `vertexmap`

3.8.5 Step 4 - Calculate Normals and Colors

If the current state of the model should be rendered after this iteration, it is necessary to update the surface normal vectors and colors.

The normal indices are stored in two buffers, `indexmap1` and `indexmap2`. Each texel in one of these maps describes a triangle on the surface of the model, as illustrated in figure 3.4. The corresponding texel in the vertex map provides the first corner, while the two indices contained in the texel point to the two other corners in the vertex map. From each of these triangles, the normal vector is calculated and the normalized average value of both normals is then stored in the normal map.

The surface colors are calculated directly from the forcemap. First, the absolute length of the force vector is calculated. It is then used to linearly interpolate between a base color (gray) and a force color (red).

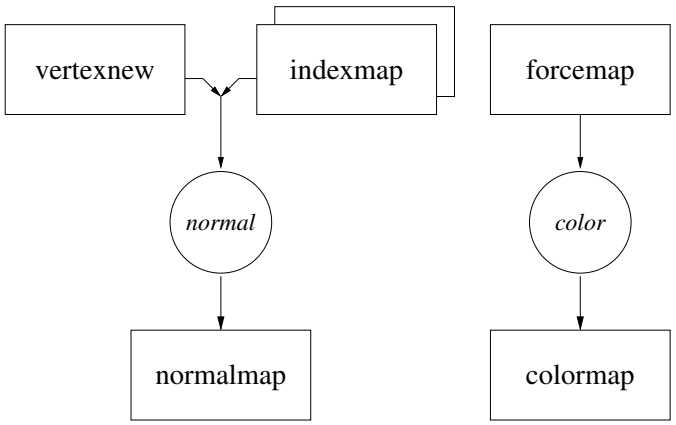

Figure 3.9: simulation step 4

Chapter 4

Class Structure

In figure 4.1, the relationships of the different classes used in this thesis are shown. The central class is `Buffer`, representing a two-dimensional data structure that can be used in OpenGL as a render target, a texture or a vertex array. This is an abstract class, and its methods are implemented in `UberBuffer` (see section 2.2) and `PBuffer`. `PBuffer`s offer some, but not all of the functionality of `UberBuffer`s, however, they are supported on almost all OpenGL implementations and therefore largely hardware-independent (and beyond the scope of this thesis).

Three other classes are derived from `UberBuffer`: `FloatUberBuffer`, `ShortUberBuffer` and `DepthUberBuffer`. The prefix specifies the data type contained in the buffer, this being four-component vectors of `GL_FLOAT` or `GL_SHORT` and single-component depth values. Each of these classes contains a subclass of `Texture`. This class is used to encapsulate an OpenGL texture object and is only used as an auxiliary when the `Buffer` object is bound to a texture. Additionally, a `FloatUberBuffer` can be treated as if it is composed from an appropriately sized array of `Vector`s, a class to encapsulate operations like scalar or cross product.

Vertex and fragment shaders are encapsulated in the `Shader` class, while the `Framebuffer` class can be used to direct rendering either to a set of `Uberbuffer`s or to the screen.

Finally, the main classes, `Simulation` and its derived class `GPUSim`, put it all together. `GPUSim` contains references to several `Shader` and `Buffer` objects and to one `Framebuffer` object.

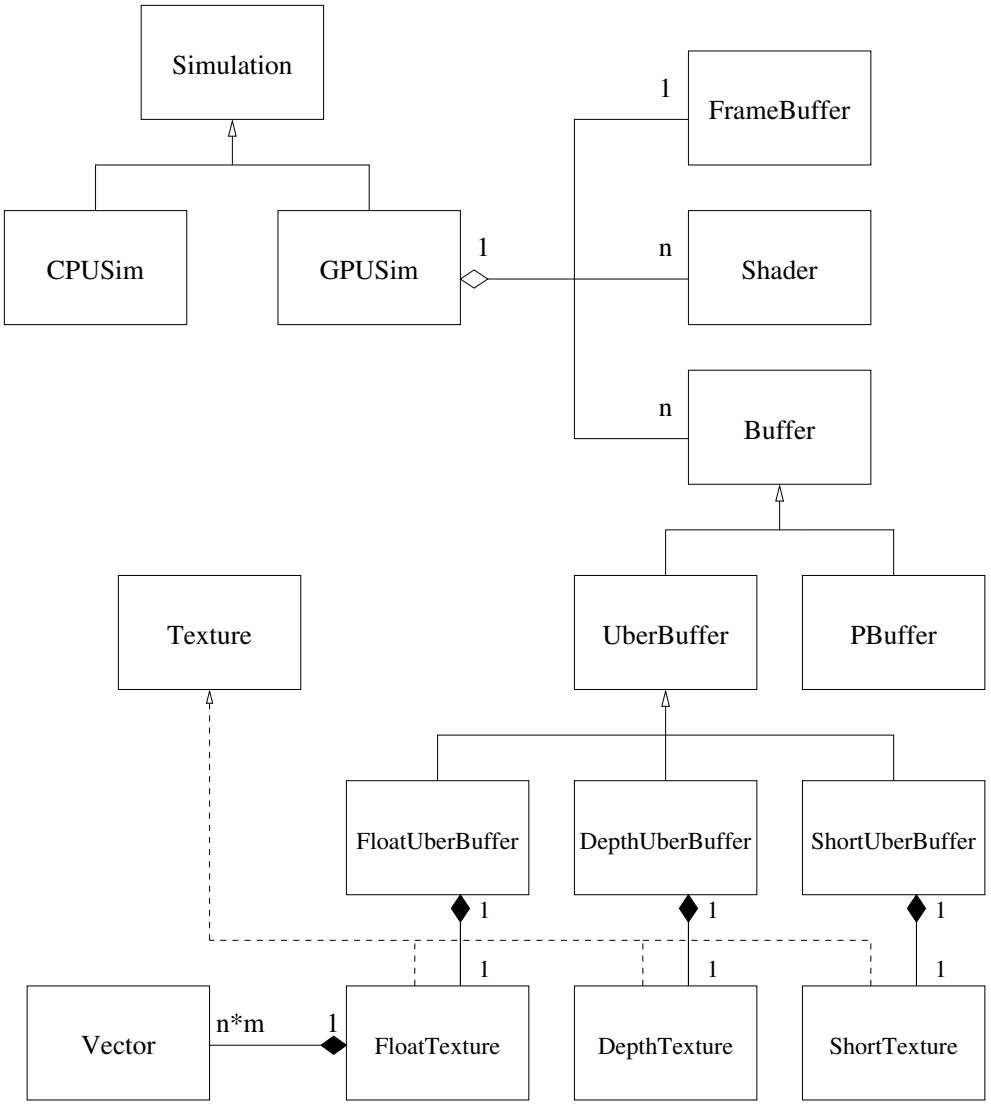

Figure 4.1: class diagram

Now let's look at the important parts in greater detail.

4.1 `Vector` class

This class offers a convenient method to manipulate four-component float vectors. Its four attributes, `x,y,z` and `w`, are made public for easy access. This contradicts usual object oriented design guidelines, however, as access to a single vector component is frequently needed, especially in the CPU-based simulation, eliminating the `get()/set()` call that would otherwise be necessary can save execution time. Unless otherwise stated, the functions and operators ignore the `w` component, as it is usually used to store a value that is not directly related to the spatial vector `(x,y,z)`.

`Vector` offers the following public methods:

- `void set(GLfloat ix, GLfloat iy, GLfloat iz, GLfloat iw)`
  Sets the entire vector at once, including the `w` component.
- `void normalize()`
  Normalizes the vector `(x,y,z)`.[1]
- `GLfloat length()`
  Returns the current length of the vector `(x,y,z)`.
- `void print()`
  Prints the vector to standard output for debugging purposes.
- `void zero()`
  Sets the `x`, `y` and `z` components to 0.0.

For convenient mathematical operations, some operators have been overloaded:

`+ - +=`	component-wise addition, subtraction and addition with assignment, requires second vector operand
`*`	either scalar multiplication with second vector operand or scaling with `GLfloat` operand
`&`	vector product (cross product) with second vector operand
`~`	unary length operator ($\sqrt{x^2+y^2+z^2}$)

[1] When the length of this vector is zero, it remains unchanged.

4.2 `Shader` class

A simple encapsulation for a vertex or fragment shader, which provides the following methods:

- `Shader(char* name, GLenum prof)` (constructor)
  Load the shader program from `name`, using the profile `prof`, which should be `GL_FRAGMENT_PROGRAM_ARB` or `GL_VERTEX_PROGRAM_ARB`.
- `GLuint get()`
  Return the shader's OpenGL name (an identification number).
- `void bind()`
  Activate this shader for its profile (only one shader can be active for each profile).
- `void release()`
  Deactivate the shader.

4.3 `FrameBuffer` class

Also a simple encapsulation class, this time for a framebuffer object. Its purpose is to group one or more `Buffer` objects into a render target, where drawing operations can be directed. The interface is similar to `Shader`:

- `GLuint get()`
  Return the framebuffer's OpenGL name (an identification number).
- `void bind()`
  Activate this framebuffer as rendering target for the current OpenGL context.
- `void release()`
  Deactivate the framebuffer, thus reactivating the default framebuffer, whose contents corresponds to the visible window on the user's display.

4.4 `Simulation` class

This is the main class of the simulation program with the following public methods:

- `void load_model(char* name, Parameters parameter, Vector scale, Vector offset)`
  Load a new model from the files "`name`.ele" and "`name`.node" (see also section 5.2), using `parameters` (specifically `spring_hardness`, `volume_hardness` and `vertex_mass`). `scale` and `offset` can optionally be used to apply a linear transformation to the coordinates of the model.

- `void reset()`
  Delete all vertices and tetrahedra from the simulation.

- `void make_normals()`
  Update the normal map with an approximation of the current surface normals of the model. Entries in the normal map that do not correspond to surface vertices[2] will not contain any sensible information.

- `void make_colors()`
  Write colors to the color map, dependent on the current force acting on each vertex. The color is interpolated between gray (no force) and red (force vector of length 1 or greater). See figure 5.3 for an example.

- `void draw_surface(int grid)`
  Draw the triangulated surface of the model, shaded with the normal vectors that `make_normals()` calculates. Optionally, when `grid == 1`, also draw the triangle outlines.

- `void mouse_click(Vector pos, Vector org, GLfloat rad, GLfloat par, int mode)`
  This method should be called to start user interaction with the model (e.g. from `glutMouseFunc`[3]). `mode` can be set to either `MOUSE_MODE_DRAG`, `MOUSE_MODE_PUSH` or zero to select one of the two interaction modes or to end user interaction.

  `MOUSE_MODE_DRAG`: `pos` specifies a point on the object surface, `org` is unused in this mode. `rad` describes the radius around `pos` in which

[2] which have to be indicated in the object description
[3] a function that is called by the GLUT library on receiving a mouse click event

all vertices will be 'grabbed', while `par` specifies the speed with which the grabbed vertices will be moved.

`MOUSE_MODE_PUSH`: The line through `pos` and `org`, together with `rad`, specifies an infinitely long cylinder. A force parallel to the cylinder's axis (`pos - org`), of size `par`, will act on all vertices inside the cylinder.

`pos` and `org` can be obtained from two-dimensional window coordinates by using `gluUnProject`.

- `void mouse_drag(Vector pos, Vector org, GLfloat rad, GLfloat par)`
  This method is designed to be called from `glutMotionFunc`[4] and takes the same parameters as `mouse_click(...)`, with the exception that the mouse mode can not be changed during user interaction.

- `void step(Parameters parameter)`
  This method provides the core simulation functionality. `parameters`, as described in section 5.1, allows the user to adjust all aspects of the simulation. When called, the simulation advances in time by one step. Its duration is specified by `parameters.timestep`.

4.5 `Buffer` class

`Buffer` is the main class for the management of two-dimensional arrays of float vectors. The `Buffer` class is an abstract base class, thereby allowing different implementations to be derived.

- `void load(GLvoid* data)`
  The buffer is filled with data, starting from the pointer location. The amount of data required is dependent on the type and extents of the buffer.

- `void read(GLvoid* data)`
  The contents of the buffer are stored at `data`. Note that `*data` must provide enough space for the contents of the specific buffer.

- `void bind_array(GLenum array, GLint size)`
  The buffer's contents are used as vertex array data. `array` specifies which array is used (e.g. `GL_VERTEX_ARRAY`, `GL_NORMAL_ARRAY` or

[4] similar to `glutMouseFunc`, this function is called when the mouse is clicked and dragged

GL_ELEMENT_ARRAY_ATI), while size denotes the number of vector components that are used (usually 3).

- void bind_framebuffer(GLenum where)
  Attach the buffer as a part of the currently active framebuffer. The where parameter selects the attachment point, which is usually GL_AUXn (the n-th auxiliary color buffer) for a FloatUberBuffer or GL_DEPTH_BUFFER_ATI for a DepthUberBuffer.

- void bind_texture(GLenum where)
  Attach the buffer as a texture. where specifies the texture unit that should be used, e.g. GL_TEXTURE0_ARB.

- void render(GLint texels, GLenum where, GLfloat z)
  Fills the currently active framebuffer by rendering a quad (assuming that bind_framebuffer has been called before, which sets the projection and modelview matrices to the identity matrix). texels specifies the number of texels in the buffer that are actually used and thereby influences the size of the quad, where denotes the texture unit that receives the texture coordinates for the quad and z specifies the z coordinate at which the quad should be rendered.
  Note: the texture coordinates are mirrored at the x-axis in respect to the quad coordinates. This ensures that each of the quad corners corresponds to the same texture corner, as OpenGL assumes that the framebuffer starts with the coordinate origin in the *lower* left corner, while the origin of a texture, when stored in memory in row-major order, is assumed to lie in the *upper* left corner.

- void release()
  This method detaches the framebuffer from any attachment points (texture, array or framebuffer).

- int is_valid()
  This method returns true when the allocation of the Buffer object succeeded.

4.6 Classes Derived from Buffer

Several classes have been derived from Buffer, primarily UberBuffer, which uses ATI Superbuffers to implement the required functionality, and PBuffer, which provides a portable solution via the GLX_SGI_PBUFFER extension.

The `UberBuffer` class does not yet specify the type of the buffer object. Therefore, three other classes, `FloatUberBuffer`, `ShortUberBuffer` and `DepthUberBuffer`, are derived, which provide different constructors to instantiate a certain kind of Superbuffer.

- `FloatUberBuffer(GLint w, GLint h)` creates an Superbuffer with `w * h` entries, consisting of four `GL_FLOAT`s each. This kind of buffer is used as color render target, texture or vertex array.
- `ShortUberBuffer(GLint w, GLint h)` creates an Superbuffer with `w * h` entries, consisting of four `GL_SHORT`s each. This kind of buffer is used as element array.
- `DepthUberBuffer(GLint w, GLint h)` creates an Superbuffer with `w * h` entries, each being a single value of type `GL_DEPTH_COMPONENT`. This kind of buffer is used as depth map.

4.7 Usage Example

To conclude the class description, an usage example shall be given. The following code snippet demonstrates how a fragment shader could be used to generate vertex data in an `FloatUberBuffer` which is rendered afterwards by indexing the vertex data from a `ShortUberBuffer`.

```
FrameBuffer* framebuf;
Shader* generate_vertices;
Buffer* input;
Buffer* vertices;
Buffer* triangles;

GLshort tri_data[64][64][4];
int     tri_count;

[..]

generate_vertices = new Shader( "generate_vertices.fp",
                                GL_FRAGMENT_PROGRAM_ARB );

framebuf          = new FrameBuffer();
input             = new FloatUberBuffer(64, 64);
vertices          = new FloatUberBuffer(64, 64);
```

```
triangles         = new ShortUberBuffer(64, 64);

triangles->load(tri_data);

[..]

framebuf->bind();
generate_vertices->bind();

vertices->bind_framebuffer(GL_AUX0);
input->bind_texture(GL_TEXTURE0_ARB);

vertices->render();

input->release();
vertices->release();

generate_vertices->release();
framebuf->release();

[..]

triangles->bind_array(GL_ELEMENT_ARRAY_ATI, 1);
vertices->bind_array(GL_VERTEX_ARRAY, 3);

glDrawElementArrayATI(GL_TRIANGLES, tri_count*3);
```

Chapter 5

Implementation Details

This chapter describes how the different classes that were introduced in the previous chapter work together to form a running simulation.

5.1 Controlling the Simulation

To control the behavior of the simulation, the `Parameters` structure is used, which can be seen below. This structure is passed to the `Simulation` class when loading a model and proceeding with the simulation.

```
typedef struct {

  GLfloat volume_hardness;
  GLfloat spring_hardness;
  GLfloat vertex_mass;

  Vector  external_force;
  GLfloat plasticity;
  GLfloat timestep;
  GLfloat friction;
  Vector  boxsize;
  Vector  gravity;

} Parameters;
```

The `volume_hardness`, `spring_hardness` and `vertex_mass` fields are evaluated when a model is loaded and specify the overall stiffness and density of the object. As mentioned previously, using uniform values throughout the

object results in non-uniform behavior

All other fields are read at every simulation step and can be used to dynamically change specific aspects while the simulation is running:

- `external_force`: A `Vector` that describes a force which is applied to every vertex and can be used, e.g, to simulate wind.
- `gravity`: Similar, but the `Vector` is treated as an acceleration vector (the final force is therefore dependent on the vertex mass as well).
- `plasticity`: A factor that, if different from zero, specifies how easily the object can be deformed plastically. When the force exerted by a spring exceeds the bound $rest\_length \cdot spring\_hardness \cdot plasticity$, the rest length of the spring is set to the current length and the rest volume of the tetrahedron to the current one, thereby effectively deforming its rest shape.
- `timestep`: The amount of simulated time that passes in one integration step. Setting this value too high will most likely cause the object to explode, as the numerical integration method becomes unstable (see also fig. 2.3).
- `friction`: Specifies the amount of friction in the simulation. The friction force is calculated by multiplying the current speed of a vertex with this factor.
- `boxsize`: The x,y and z components of this `Vector` specify the extent of the world box in each of the three space directions.

5.2 Object File Format

An object to be loaded into the simulation is described by two files, "`object.node`" and "`object.ele`". The node file specifies the initial positions of the vertices, while the element file describes the tetrahedra connecting them. These file formats are taken from the TetGen mesh generator [Si04].

5.2.1 Node File

The format of the node file is quite straightforward. The first line specifies the number of vertices as an integer, followed by the dimensionality of the

points (always `3`), the number of attributes per point (always `1`, the lock flag), and the number of boundary markers (unused and therefore `0`).

The rest of the file consists of one line per vertex, containing of the following whitespace-separated values:

- the vertex number, starting from 1 (integer)
- the `x`,`y` and `z` components of the vertex coordinate (float)
- the `lock` flag (integer). If this flag is `true`, the vertex will effectively be ignored by the simulation and will not move. This can be used to simulate objects that are affixed to an immobile entity, e.g. a flag to a flagpole or a horizontal beam to a wall (see also section 6.2.1).

5.2.2 Element File

The element file also starts with the number of tetrahedra in the first line, followed by the number of points per tetrahedron (always `4`) and the number of attributes (always `1`, the surface flag). This is in turn followed by one line per tetrahedron, again with whitespace-separated values:

- the tetrahedron number, starting from 1 (integer)
- four vertex indices (integer). These indices refer to the four corners of the tetrahedron, as specified in the node file.
- the `surface` flag (integer). This flag is a four-bit value, in which a `1` signifies that the triangle *opposite* the corresponding vertex is a surface triangle.

Example: assuming the element file contains the following line,
`15 3 8 5 19 10`
this means that the tetrahedron with number 15 is formed by the vertices 3, 5, 8 and 19. The surface flag is 10 = `1010b` with bits 2 and 4 set, so the triangles opposite the second and fourth vertex are surface triangles. This tetrahedron has therefore two surface triangles, those formed by vertices 3, 5, 19 and 3, 8, 5.

Note that nothing prevents such a file from containing two or more disjunct sets of tetrahedra, thereby effectively describing two separate objects. However, as collision detection has not yet been implemented, these objects would be unable to interact with each other.

5.3 Potential Pitfalls

As the Superbuffer functionality is so far only available in a non-public beta driver from ATI, some glitches have to be expected. Most of the development in this thesis was done with the Catalyst driver version 6.14.10.4099, which unfortunately only works with Radeon 9700/9800 cards, but not with the new X800 cards. The following issues were identified:

- While it seems like a good idea to bind the default texture to a texture unit while no `UberBuffer` is bound there, this causes a segmentation fault in atioglxx.dll.
- On driver version 6.14.10.4552 (which *does* work on the X800), no `DepthUberBuffer`s can be attached to a `FrameBuffer` without causing a 'division by zero' exception in atioglxx.dll. While the depth buffer still works despite this limitation, its contents are destroyed after every rendering pass. This effectively disables the most important optimization (see section 3.4), as the depth buffer would have to be reloaded before every pass. This heavily decreases the performance.
- While the driver version 6.14.10.4099 allows the use of `DepthUberBuffer`s, it obviously supports only one physical depth buffer. This means that after the actual scene has been rendered, the depth buffer contents (that were previously used for early z-testing) have been destroyed and must be reloaded from a persistent `FloatUberBuffer` before the next simulation step. Note that in contrast to the previous limitation, this reload only needs to be done after the scene has been rendered to the screen.

Finally, I would like to mention a certain bug in Microsoft Visual Studio that claimed about a week of debugging time, as I naively had sought the bug within my own code (sarcasm intended).

From the early stages of development on, the normal calculation had shown glitches like in figure 5.1 (some normals seemed to have their sign flipped randomly, which led to incorrectly shaded spots on the surface). I had already tried several increasingly elaborate procedures to cope with this problem, unfortunately, to no avail. Imagine my surprise when I discovered that the easiest way to get rid of this bug is to switch from 'Debug' to 'Release' build in Visual Studio - nothing else. So much for debugging.

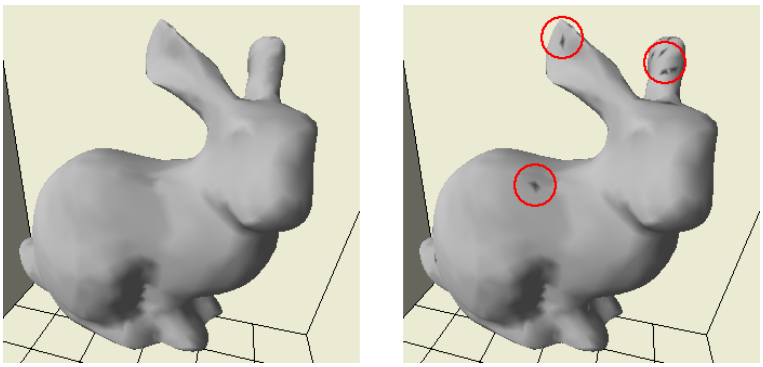

Figure 5.1: release build (left) vs. debug build (right)

5.4 User Interface

Finally, the user interface implemented in main.cc should be described here. As it is based on GLUT, the interface was not implemented as a class, because GLUT requires C-style callbacks. It consists of the following main functions:

- `void mystep(int step)`
  This function is called from the GLUT idle loop via a wrapper function and advances the simulation by one step. Additionally, it calculates and display the current frame rate and triggers a redraw of the display after a certain number of simulation steps.[1]

- `void display()`
  This function does camera setup, calculates the shadow and normal vectors if desired and finally draws the scene. Afterwards, the information display is rendered and `glutSwapBuffers` is executed.

- `void special(..)`, `void resize(..)`, `void keyboard(..)`, `void passive(..)`, `void motion(..)`, `void click(..)`
  These functions are registered as the respective GLUT callbacks and mainly change parameters or pass commands to the simulation.

- `void create_menu()`, `void initGLUT(..)`, `void initGL()`
  These setup functions initialize various aspects of the user interface and setup the OpenGL state.

[1] To the human eye, there is no difference between 60 and 300 frames per second. It is therefore sufficient to redraw the display only after every fifth simulation step.

- `int main(int argc, char* argv[])`
  The `main` method initializes the various graphics libraries, instantiates a `Simulation` object, registers the various functions with GLUT and finally starts the GLUT main loop.

The main window is shown in figure 5.2. In the upper part, the current simulation parameters are displayed, while the current simulation step (and, when the simulation is running, the rate of steps per second) is shown in the title bar.

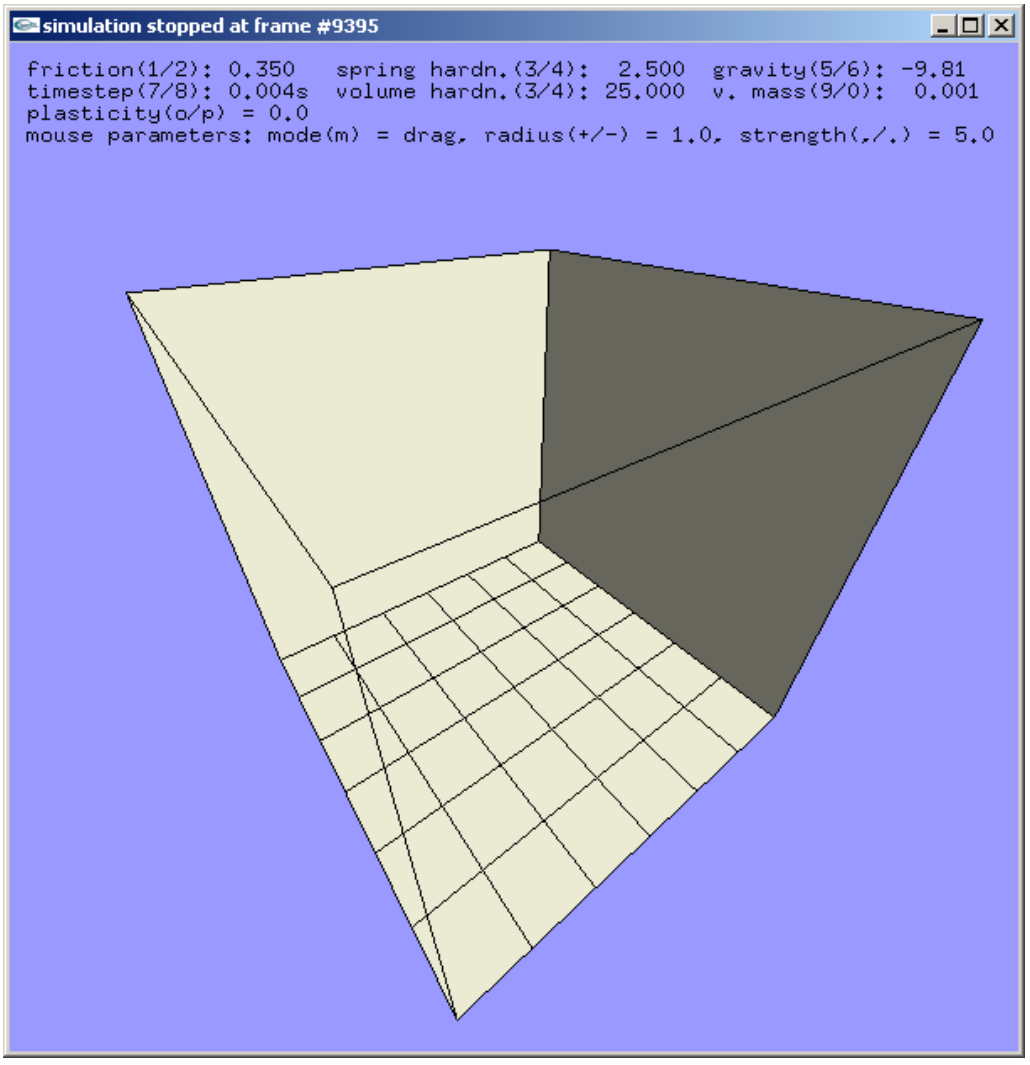

Figure 5.2: empty world with parameter listing

Most of the user interaction is done with the mouse (a three-button model is required):

Left Button allows the user to interact with the model, depending on the mouse mode that can be switched either from the menu or by pressing `m`.

- In 'drag' mode, the model can simply be grabbed by clicking, then dragged to another location. A 'rubber band' visualizes the target position. In this mode, the `radius` parameter specifies a sphere around the click point that is grabbed, while the `strength` parameter controls the velocity with which the object is pulled along the rubber band.
- In 'push' mode, the model is subjected to a force of size `strength`. This force is directed along an axis through the camera center and the click point and acts on all points inside a cylinder with the specified `radius`.

Middle Button allows the user to rotate the view by clicking and dragging. When a scroll wheel is present, it can be used to zoom in and out.[2]

Right Button opens a menu that can be used to load objects and change simulation parameters.

All simulation parameters can be changed via the keyboard. Spring and volume hardness change in parallel, with the volume hardness being always ten times the spring hardness. This has proven to be a well-suited approximation. When the information display is enabled, the respective increase/decrease keys are shown next to each parameter (see figure 5.2). Additionally, the following keybord commands are available:

`Space` starts or stops the simulation.

`s` performs a single simulation step.

`d` toggles rendering of the world.

`g` toggles rendering of the surface grid and the coordinate axes.

`h` toggles rendering of the object shadow(s).

[2]When using Windows, this requires a patched `glut32.dll`. The source code and patch have been included on the CD.

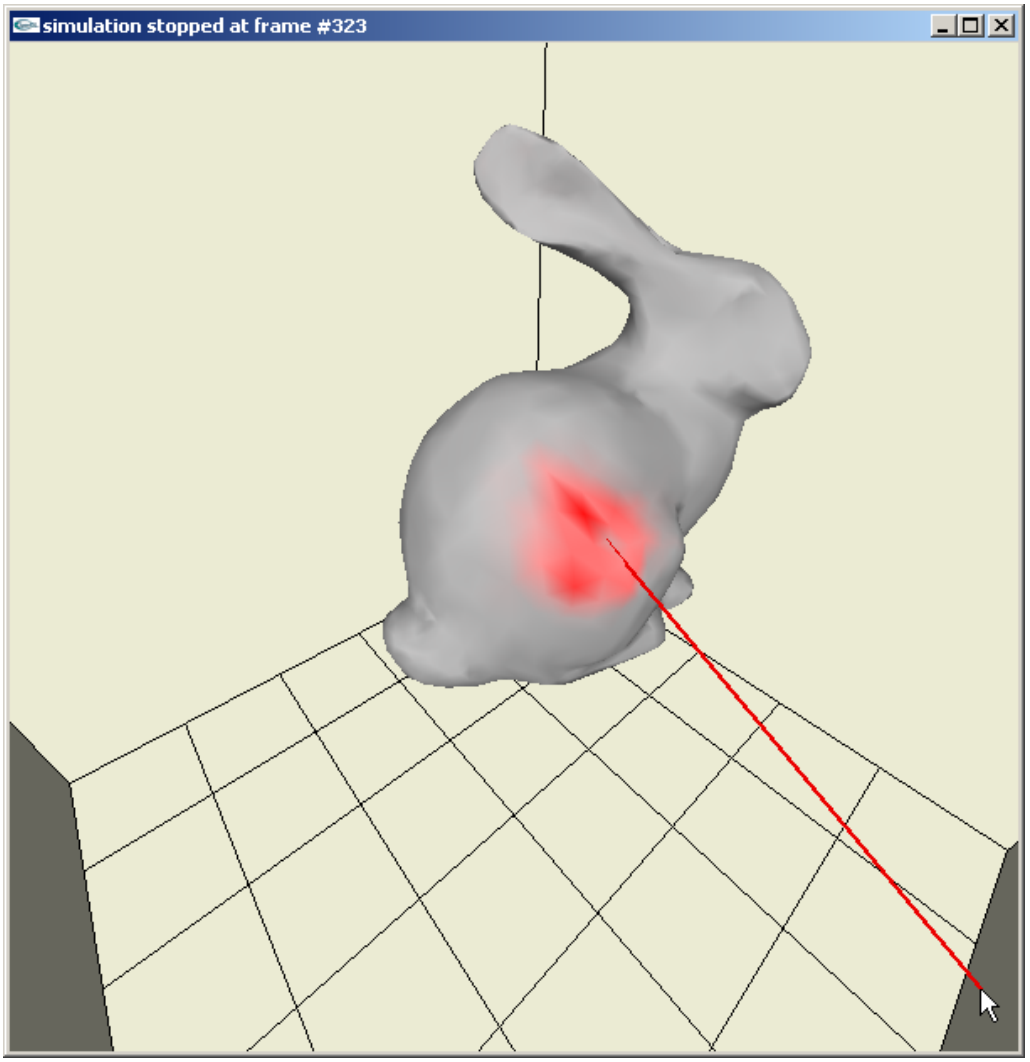

Figure 5.3: force visualization when dragging an object

`m` toggles the mouse mode between 'drag' and 'push'.

`i` toggles display of the simulation parameters.

`c` toggles display of force-dependent colors.

Chapter 6

Performance

6.1 Speed

Apart from physical plausibility, a simulated environment should ideally run in realtime. In this case, this means that if the simulation timestep is Δt ms, a framerate of at least $\frac{1000}{\Delta t}$ Hz must be reached to achieve realtime simulation.

Figure 6.1: Stanford Bunny

For most tests, a model of the venerable 'Stanford Bunny' has been used.[1] Its home can be found at [Tur04]. Note that in its original form, the bunny is only a surface description that has to be converted into the tetrahedrized volume description that this thesis expects, however, this conversion is beyond the scope of this thesis and is assumed to be already done (see also the *tetgen* program [Si04]).

[1] No bunnies were harmed during the making of this thesis.

The different models have the following dimensions:

model	*vertices*	*surface triangles*	*tetrahedra (total)*	*tetrahedra (clipped[2])*
Cuboid	1200	1390	5177	5012
Liver	1915	1992	8078	7536
Stanford Bunny	3019	4046	11206	9804
Double Bunny	6038	8092	22412	19608
'Überbunny'	19266	16184	89648	84104

The reason for the inequality between total and clipped tetrahedron count is, like described at the end of section 3.3, the necessity to reduce the maximum number of tetrahedra per vertex to a sensible value well below the highest occuring valence.

The reason that often a few vertices have very high valences about 100, while the vast majority has no higher valence than about 30, is that the models have usually been generated by volume discretization algorithms that tend to output many small tetrahedra in the interior or at the edges of the object, while other parts contain fewer, larger tetrahedra.

Fortunately, this limitation usually does not cause any visible difference in the behaviour of the object, as mostly very small tetrahedra are discarded. A simple trick to ensure that the visible surface of the object does not change at all is to re-sort the element file so that all tetrahedra that have at least one surface triangle are read first and will therefore not be clipped.

As speed measurements are not very significant without any comparision, I have again chosen the paper by Michael Teschner as a reference point.

The following parameters potentially affect the performance of the simulation:

- size (vertex and tetrahedron count) of the object
- dimensions of the Superbuffers
- size of the quad that is rendered into the Superbuffers
- maximum allowed valence = height of the texture stack
- amount of rendering to screen

Especially the last parameter might have a large impact, as the simulation as a whole is under a much heavier load when it needs to render the

[2] at a texture stack height of 30

object surface (and possibly more) in addition to the physics calculations.

The following sections compare the performance of the simulation with different objects, each time varying one of the aspects mentioned above. The tests were performed on a 2.0 GHz Pentium IV machine, equipped with an ATI Radeon 9800 Pro graphics card with 128 MB RAM. An X800 card that has 16 fragment pipelines, compared with 8 in the 9800 series GPU, was also available. It could potentially have achieved about twice the framerate of the 9800 series, unfortunately, the depth buffer optimization could not be used due to driver issues (see also section 5.3).

All measurements were acquired by running the simulation for 20 000 steps and dividing the elapsed time in seconds by 20 000. No user interaction happened during the tests. The default settings were:

- 128^2 texels buffer size
- full-size "update quad"
- texture stack height 30
- render world and object without status display and shadows
- rendering to window of 600 x 600 pixels size

Where other settings were used in the tests, they are mentioned below.

6.1.1 Superbuffer Dimensions

The measurements in figure 6.2 show the changes in performance when using different buffer sizes. The 'Double Bunny' model is missing at the smallest buffer size, as is too large (6038 vertices vs. a maximum of $64^2 = 4096$).

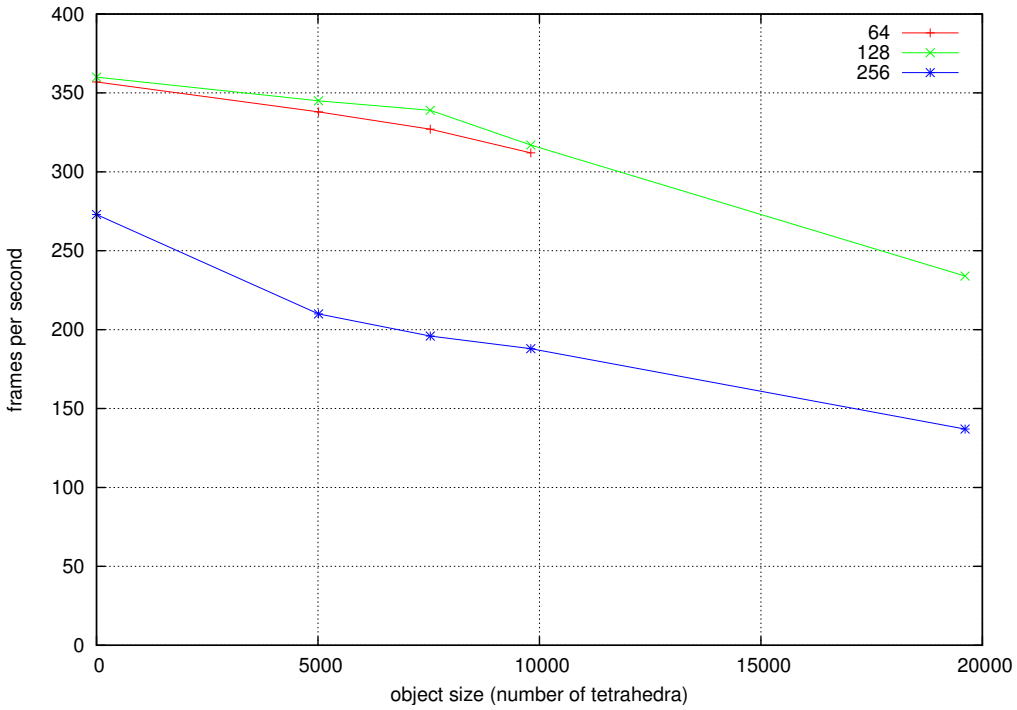

Figure 6.2: performance in relation to Superbuffer size

It is worth noting that the performance actually *increases* slightly when the buffer size is raised from 64^2 to 128^2. This is likely due to the fact that the Superbuffers are not optimized for such small textures.

When the texture size is increased to 256^2, however, the performance drops again, as now four times as much pixels have to be processed.

6.1.2 Quad Size

As the buffers often contain a significant amount of unused vectors, it might be possible to obtain a speed increase by only updating the relevant parts. This can be done by rendering a smaller quad during each pass. A performance comparision can be found in 6.3:

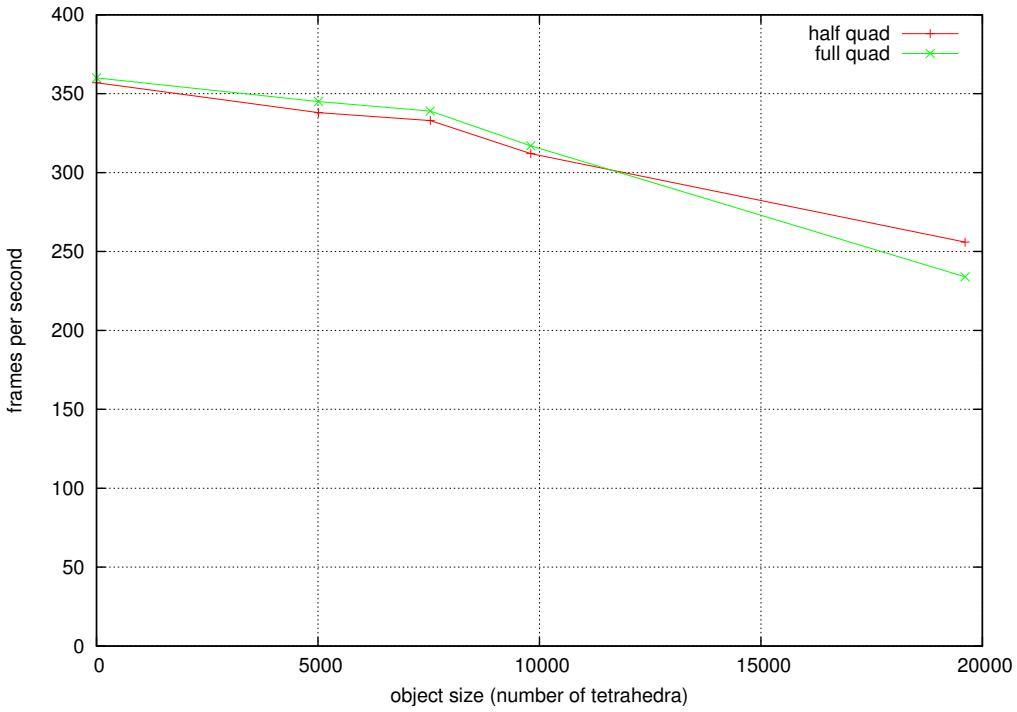

Figure 6.3: performance in relation to quad size

Again, we are in for a surprise - for most models, the performance *decreases* fractionally when the quad size is reduced. The reason is propably that the rendering pipeline is at some point optimized for the generation of full quads.

However, with larger models, the performance gain that comes from not having to render all empty texels becomes sufficient to be noticeably larger than the performance loss through not using full quads.

6.1.3 Rendering

Of course, rendering has a high impact on performance. Even simple objects usually consist of hundreds or thousands of triangles that need to be rasterized and shaded. The influence of rendering is detailed in figure 6.4.

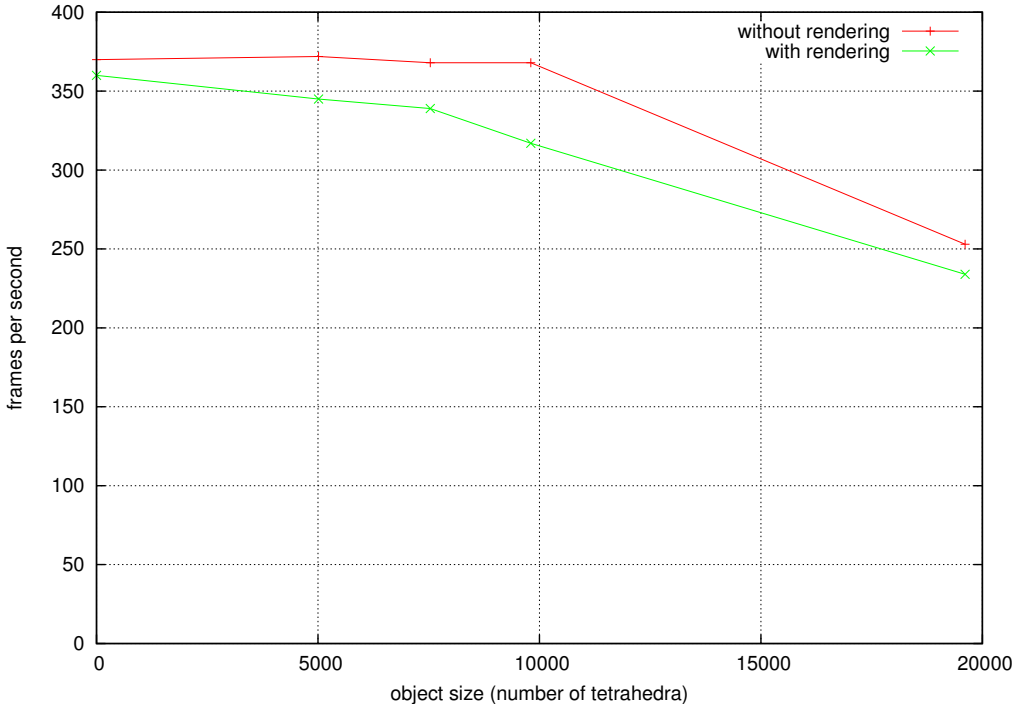

Figure 6.4: performance in relation to rendering

It is noticeable that without any drawing, the performance is capped at around 370 frames per second. Obviously, the rendering is mainly responsible for the speed differences, while the performance of the simulation itself is limited by some other factor.

6.1.4 Stack Height

The influence of the number of stack slices can be seen in figure 6.5.

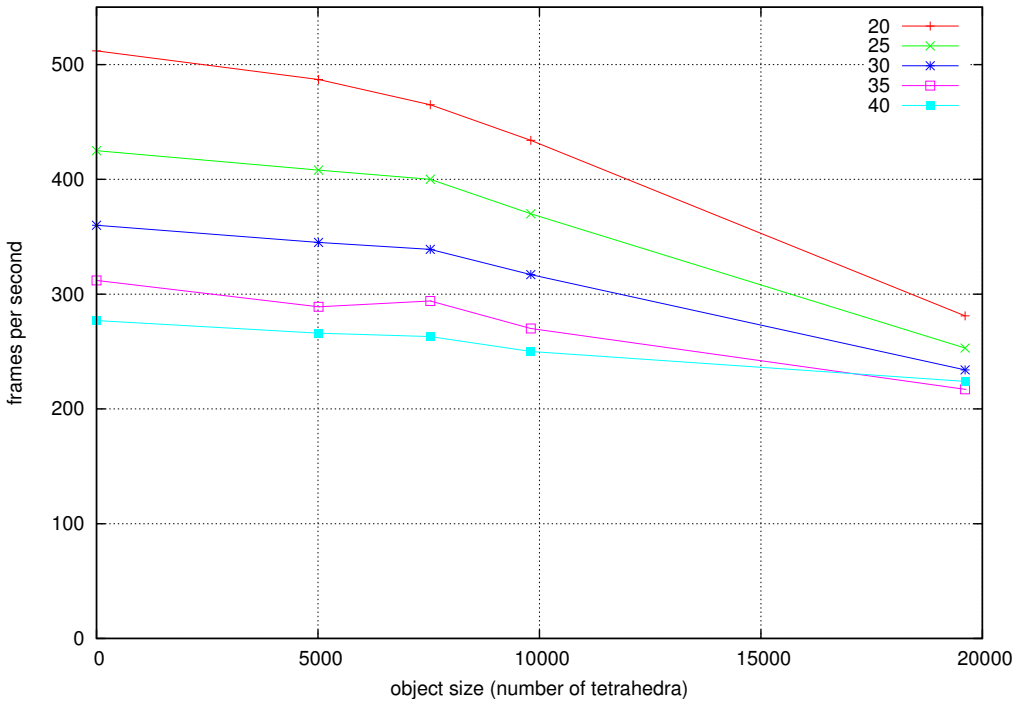

Figure 6.5: performance in relation to stack height - frames per second

Here, it is rather obvious that increasing the stack size decreases the framerate. However, as the larger stack permits more tetrahedra to be stored, the framerate alone is propably an insufficient measurement.

What is needed now is a comprehensive kind of measurement that summarizes the overall performance of a mass-spring simulation. I have decided to rate the different results in tetrahedra per second (TPS), similar to 3D engines, which are often rated in triangles per second.

The results from figure 6.5 will now be re-examined in figure 6.6.

Obviously, larger models generate a higher throughput, as the framerate drop is not linear with the number of tetrahedrons.

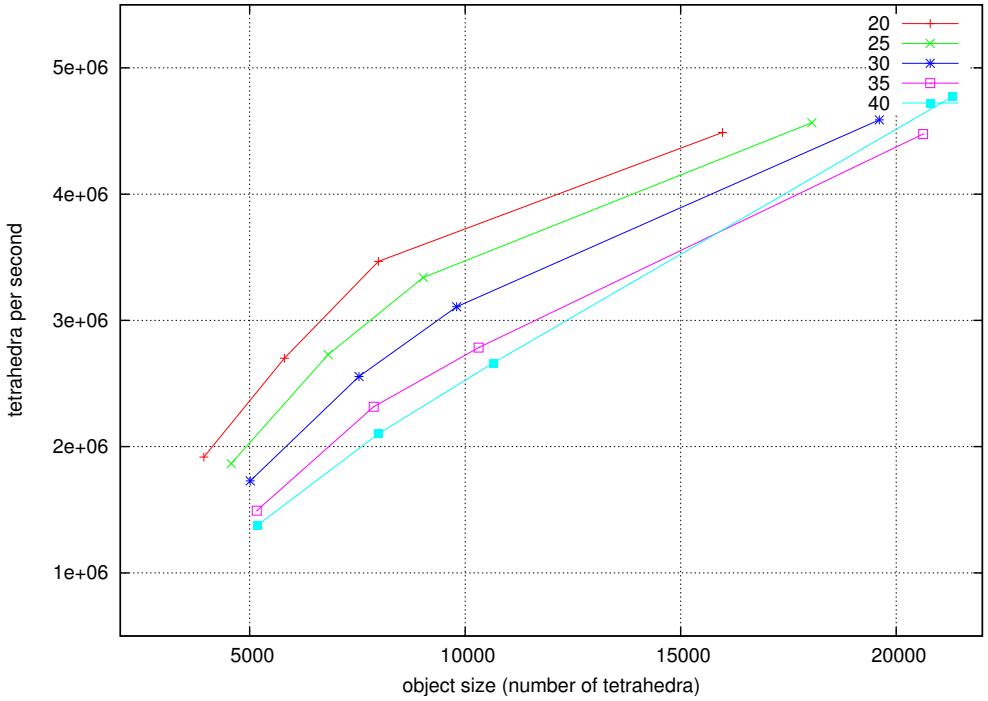

Figure 6.6: performance in relation to stack height - tetrahedra per second

6.1.5 Results

Judging from the test results, the frame rate is influenced by the different simulation aspects in the following order of importance:

1. texture stack height
2. rendering
3. Superbuffer size
4. quad size

While rendering has a large influence, it is, after all, necessary to actually draw the object. Similarly, the texture stack needs a certain minimum height to allow the object to be stored without noticeable distortions. Obviously, the setup between the different render passes takes a high performance toll.

This leaves two possible optimizations: the size of the Superbuffers and the size of the quad that is rendered into the buffers at every pass. As

Superbuffers really start to show their performance at sizes of 256^2 and above, an additional speedup might be possible by increasing the buffer size to 256^2 and additionally reducing the overhead of empty texels by decreasing the quad size.

Therefore, an adaptive quad size was implemented. The `Buffer::render()` method takes a parameter `GLint texels` that specifies how many texels in the buffer are actually filled with vectors and adjusts the quad size accordingly.

The framerates achieved with these adaptive quads are compared with the previous results in figure 6.7.

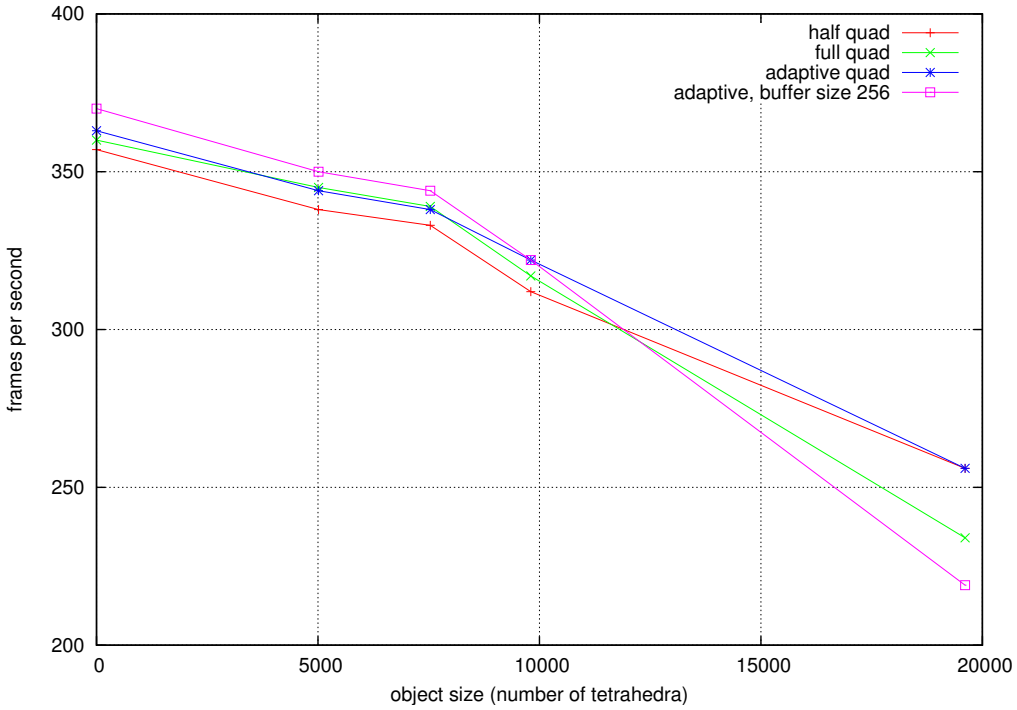

Figure 6.7: performance in relation to different quad sizes

As expected, this results in a small additional speed increase of about one percent. Obviously, the best results can in any case be acquired with adaptive quad sizes.

Unfortunately, increasing the buffer size does not seem to provide any significant speedup, even though the same amount of texels has to be rendered as with smaller buffers (thanks to the adaptive quad sizes).

6.1.6 Speed Comparision

When comparing results, only the amount of time consumed for the physics simulation should be taken into account first, as rendering can require significant time. However, as Superbuffers also provide a speedup to the rendering process itself because all vertex data is already present in graphics memory, a second comparision of the performance including rendering is advisable.

Additionally, in the case of Teschner's simulation [Tes04], only a certain percentage of time was actually used for the simulation process, while the rest was spent on collision detection. As the simulation model presented in this thesis only implements very basic collision detection with the walls of the world cuboid that does not consume any significant amount of time, it is necessary to use only the specified percentage of the total time from Teschner's paper. These results were acquired on a Pentium IV 2.8 GHz with 1 GB RAM and a GeForce 4 Ti 4600.

model	*tetrahedra*	*computation time [ms]*	*TPS rating*
Snakes	1764	4.19	420240
Pitbull	700	1.43	491021
Cows	2916	4.79	608298
Dragon	834	1.55	536576
average	1554	2.99	514034

CPU simulation performance values from [Tes04] (without rendering)

model	*tetrahedra*	*computation & rendering time [ms]*	*TPS rating*
Snakes	1764	23.60	74755
Pitbull	700	6.94	100865
Cows	2916	9.39	310421
Dragon	834	4.17	200000
average	1554	11.03	171510

CPU simulation performance values from [Tes04] (with rendering)

While the paper presents two additional models and measurements, these objects also utilize area-preserving forces and are therefore not directly comparable to the results from this thesis, which are presented in the following tables.

model	*tetrahedra*	*computation time [ms]*	*TPS rating*
Cuboid	5012	2.60	1924608
Liver	7536	2.60	2893824
Stanford Bunny	9804	2.60	3764736
Double Bunny	19608	3.61	5431416
'Überbunny'	84104	16.67	5046240
average	25213	5.61	3812165

GPU simulation performance values (without rendering)

model	*tetrahedra*	*computation & rendering time [ms]*	*TPS rating*
Cuboid	5012	2.90	1724128
Liver	7536	2.95	2547168
Stanford Bunny	9804	3.10	3156888
Double Bunny	19608	3.90	5019648
'Überbunny'	84104	24.39	3448264
average	25213	7.45	3179219

GPU simulation performance values (with rendering)

When comparing the results, the speed advantage of the GPU simulation is apparent. In a real-world scenario, simulation with additional rendering, the GPU is in some cases able to simulate over *14* times as much tetrahedra as the CPU solution, while still satisfying the realtime requirements.

The next apparent fact is that the throughput of the GPU simulation scales almost linearly with the size of the simulated object, while the computation time remains largely constant, at least with the smaller models.

For a graphical comparision, see figure 6.8.

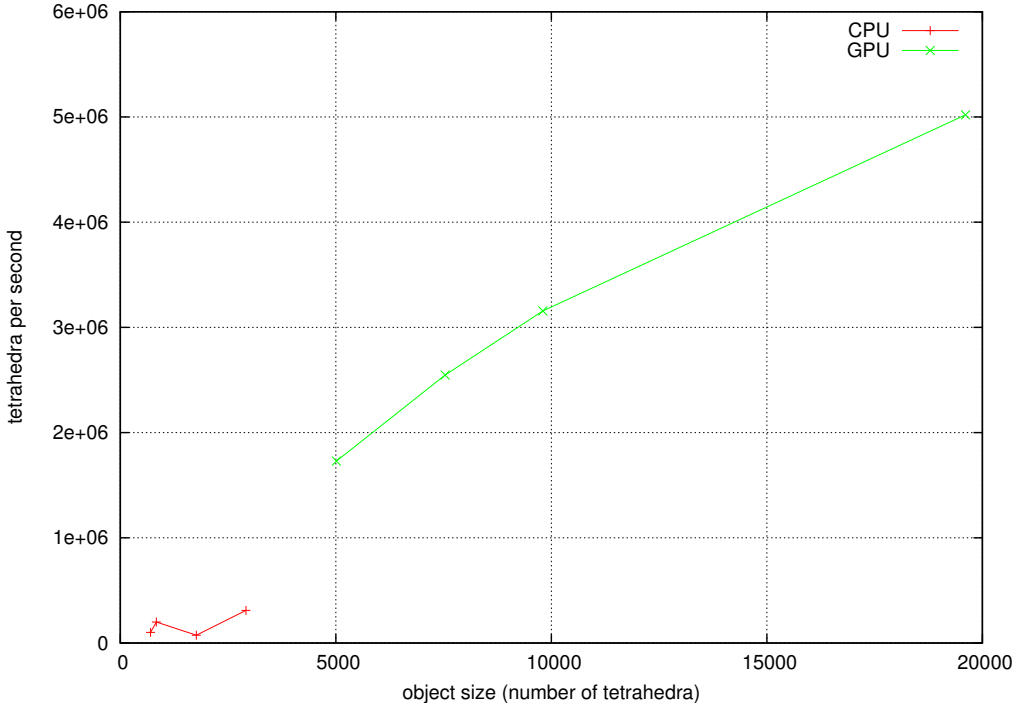

Figure 6.8: graphical comparision of CPU and GPU simulation

6.2 Precision and Stability Tests

The final question that remains to be examined is whether the GPU simulation can muster enough computational precision to run with an acceptable degree of realism.

First of all, the difference between precision and stability should be clarified.

Stability is a property of the numerical integration algorithm (see section 2.4). It is primarily dependent on the duration of the timestep, the overall stiffness of the system and the resulting size of the internal forces. Stability is basically a boolean value - the system is either stable, or it is not (see also figure 2.3).

Precision is a property of the particular implementation of the algorithm, and is dependent on the capabilities of the hardware and, to a certain degree, the order of operations. In cases where an exact result can be calculated analytically, the precision can be measured in deviation between the simulated and the pre-calculated value.

6.2.1 Precision

In this regard, it is important to remember that the fragment processor on ATI graphics cards internally works with a precision of only 24 bits (floating point values with 16 bit mantissa, 7 bit exponent and one sign bit), while externally 32-bit floats according to IEEE standard 784 are used.

A simple measurement for the accuracy of the Verlet integration is the simulation of a free-falling object. When no friction is present, the time for an object to fall from a height h at the acceleration of gravity $g = 9.81m/s$ is $t = \sqrt{\frac{2h}{g}}$.

height	timestep	fall time		deviation
		simulated	analytic	
5.0 m	2 ms	n/a	1.01 s	n/a
5.0 m	3 ms	1.37 s	1.01 s	35.1 %
5.0 m	4 ms	1.16 s	1.01 s	15.2 %
10.0 m	2 ms	2.12 s	1.43 s	48.2 %
10.0 m	3 ms	1.59 s	1.43 s	11.2 %
10.0 m	4 ms	1.52 s	1.43 s	6.3 %
15.0 m	2 ms	1.90 s	1.75 s	8.6 %
15.0 m	3 ms	1.81 s	1.75 s	3.2 %
15.0 m	4 ms	1.78 s	1.75 s	1.7 %
20.0 m	2 ms	2.14 s	2.01 s	6.4 %
20.0 m	3 ms	2.04 s	2.01 s	1.6 %
20.0 m	4 ms	2.04 s	2.01 s	1.3 %

Quite unexpectedly at first, a small timestep seems to lower precision significantly. However, this does not seem so surprising anymore when one considers that a smaller timestep also means smaller position displacements in each step, and therefore possibly increased rounding errors. When using the default acceleration of gravity ($9.81\frac{m}{s}$), this effect is obvious at a timestep of 1 ms or smaller: the object simply does not fall, but hovers in place indefinitely. It can still be moved by user interaction, but the initial displacement by gravity alone is small enough to be rounded to zero internally.

It also looks like the number of steps has an influence upon the overall error. Therefore, several free fall simulations with different gravity values were conducted:

height	gravity	timestep	steps	fall time		deviation
				simulated	analytic	
15.0 m	5.00 m/s	4 ms	634	2.54 s	2.45 s	3.5 %
15.0 m	9.81 m/s	4 ms	445	1.78 s	1.75 s	1.7 %
15.0 m	10.00 m/s	4 ms	440	1.76 s	1.73 s	1.6 %
15.0 m	15.00 m/s	4 ms	357	1.43 s	1.41 s	0.9 %
15.0 m	20.00 m/s	4 ms	308	1.23 s	1.22 s	0.6 %

Obviously, the error increases with the number of steps, which is also to be expected, as the results of step n are reused in step $n+1$, thereby adding a slowly increasing total error to the result.

So far, only dynamic precision has been tested. As a test of static precision, an elastic bar that is fixed at both ends and deforms under the influence of gravity is used (see figure 6.9).

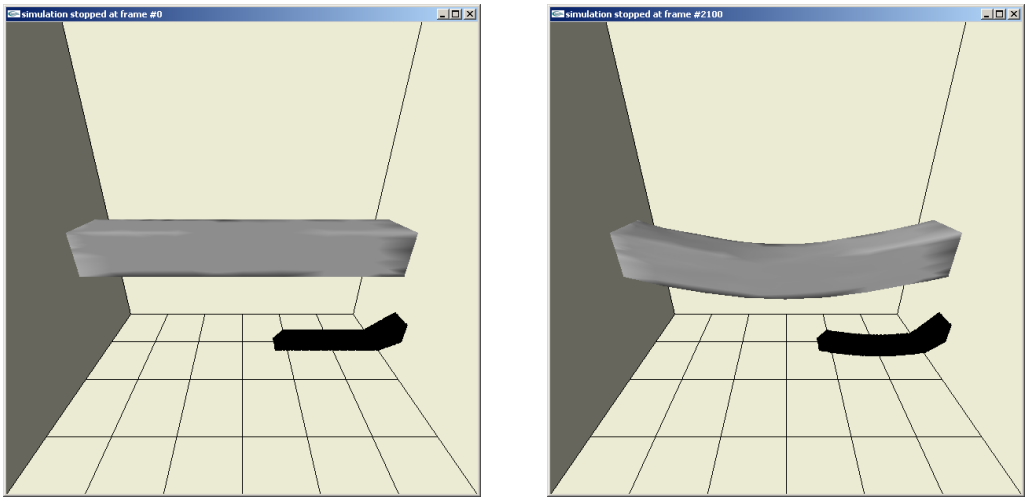

Figure 6.9: bar under influence of gravity

The parameters for this bar were as mentioned above:

- size: 14m x 2m x 2m
- weight: 1.2 kg, roughly uniform distribution
- linear hardness: 2.5 N/m
- volume hardness: 25.0 N/m^3

For the analytical calculation of a deformation, a variant of the so-called beam equation can be used:

$$\frac{d^4v}{d^4x} = -\frac{q}{EI} \tag{6.1}$$

with $v(x)$ being the displacement at position x, q the load upon the beam (in this case,its own weight), E the modulus of elasticity and I the area moment of inertia.

The beam equation is now integrated four times:

$$\begin{aligned} v(x) &= -\tfrac{q}{EI}\tfrac{x^4}{24} + D_1\tfrac{x^3}{6} + D_2\tfrac{x^2}{2} + D_3x + D_4 \\ v'(x) &= -\tfrac{q}{EI}\tfrac{x^3}{6} + D_1\tfrac{x^2}{2} + D_2x + D_3 \end{aligned} \tag{6.2}$$

To solve for the unknown values in this equation, four boundary conditions are needed. As the bar is fixed at both ends, the conditions $v(0) = 0, v(L) = 0, v'(0) = 0$ and $v'(L) = 0$ can be used, where L is the total length of the beam. The beam equation now looks as follows:

$$v(x) = -\frac{q}{EI}(\frac{x^4}{24} - L\frac{x^3}{6} + \frac{L^2}{3}\frac{x^2}{2}) \tag{6.3}$$

We are interested in the displacement at the middle of the beam, so $x = 7.0m$. I is dependent on the cross section of the beam. For a quadratic beam, $I = \frac{l^4}{12} = 1.33m^4$. As the load (weight) is uniformly distributed over the beam, $q = \frac{mg}{L} = 0.84\frac{kg}{s^2}$. Finally, the modulus of elasticity, or Young's Modulus, needs to be determined.

Assuming that the bar is compressed along its main axis by a length difference ΔL. The necessary force is $F = \Delta L \cdot D_s + \Delta L \cdot A \cdot D_v$ with D_s, D_v being the spring and volume hardness and $A = 4m^2$ the area over which the force is applied. Young's Modulus is then defined according to the following equation:

$$E = \frac{FL}{\Delta LA} = \frac{L}{A}(D_s + AD_v) = 358.75\frac{N}{m^2} \tag{6.4}$$

These values can now be inserted into equation 6.3, resulting in a final displacement of $v(7.0) = 1.58m$ at the center of the bar.

In the simulation, the displacement is $1.24m$, with an error of about 22%.

This error is, unfortunately, quite large. It is possibly caused by a combination of internal rounding errors, a not perfectly uniform internal structure of the simulated bar and the approximation of Young's Modulus through

spring and volume hardness values.

In conclusion, this simulation seems not particularly suited for exact calculations, but is on the other hand likely precise enough to be used in interactive environments, e.g. for surgery simulation, games or animations.

6.2.2 Stability

The stability of the simulation is mainly influenced by the length of the timestep and the hardness of the springs and volumes in the object.

In this thesis, the simulation parameters were set to the following values, unless noted otherwise (see also section 5.1):

- volume hardness: 25.0 $[\frac{N}{m^3}]$
- spring hardness: 2.5 $[\frac{N}{m}]$
- vertex mass: 1 [g]
- external force: (0.0, 0.0, 0.0) [N]
- plasticity coefficient: 0 (= no plastic deformation)
- timestep: 4 [ms]
- friction coefficient: 0.35
- boxsize: (15.0, 15.0, 15.0) [m]
- gravity: (0.0, 0.0, -9.81) $[\frac{m}{s^2}]$

To achieve simulation in realtime, the total computation and rendering time must not be larger than the timestep.

When comparing the timestep with the times given in section 6.1.6, it becomes obvious that this simulation is able to process objects with about 20 000 tetrahedra in realtime. This is about one order of magnitude faster than the CPU based solution (that additionally ran on a faster machine).

With these settings, the simulation runs stable under almost all circumstances. Stability can, unfortunately, still be lost when the forces generated through user interaction become too large.

The influence of precision on stability seems rather small, as can be seen by comparision between the GPU and the CPU simulation. Teschner does not mention whether his simulation uses single precision or double precision

floats. However, it is safe to assume that *at least* 32 bits are available and the precision is therefore higher as with the 24 bit GPU floats. The largest timestep for the CPU simulation is 7.1 ms for a comparatively flexible model. The other objects, which have a higher stiffness, also use timesteps of 4.0 ms or smaller. [Tes04]

One final observation regarding stability was that objects whose tetrahedra are regularly aligned tend to be less stable than objects that are tetrahedrized irregularly. The reason is possibly that these models are more susceptible to rounding errors, as their stability depends on the equality of the internal forces. This requirement is not present in an irregular mesh.

Chapter 7

Future Work

While this thesis has shown the considerable potential of a mass-spring simulation running on a GPU, there are lots of other related issues that lie beyond its scope.

This chapter tries to give an overview about possible extensions to the existing simulation environment.

7.1 Constraint-Based Model

Thomas Jakobsen [Jak01] has described a physics model that is also based upon Verlet integration, but does not use springs as connecting entities between mass points. Instead, this model tries to enforce distance constraints between pairs of vertices, for example, a constraint might require two points to be always at least two length units distant from each other. The vertices are then displaced to satisfy the constraints (usually, this process has to be iterated several times to achieve good results).

This model is of course not very well suited to modelling deformable objects, however, it might be more useful for cases like skeletal animation where elasticity is not required.

Of course, the ultimate goal in this respect would be a combination of rigid constraints and deformable springs in the same model or even the same object (however, the latter might prove problematic, as the constraints would propably upset the stability of the spring-based part).

7.2 Dynamically Changing Topology

While the current version of the simulation system allows plastic deformation of an object, the topology itself can not be changed. For a surgery simulation,

where the object is actually cut, this would be highly valuable.

Implementing this might seem straightforward at first - to delete a tetrahedron, its spring and volume hardness just has to be set to zero. This works without a hitch for internal tetrahedra, however, problems become apparent when tetrahedra on the surface are deleted. As the triangle mesh of the surface is stored in a different buffer, this buffer would have to be updated as well, making it necessary for every tetrahedron to store references to its surface triangles. As the `glDrawArraysATI` call expects a contiguous list of primitives, a triangle can not be deleted outright, but would need its indices remapped to some vertices outside the visible range.

Additionally, to make the cut surfaces visible, new triangles would need to be generated and inserted into the `trimap` buffer.

7.3 Particle Simulation

It would easily be possible to add a particle component to the simulation. A vertex that is not connected to any tetrahedron would move only under influence by external forces. To visualize these unconnected vertices, a second `ShortUberBuffer` with a particle index list would be needed. The entire particle cloud could then be rendered by calling `glDrawArraysATI` a second time, now with `GL_POINTS` as parameter.

However, to allow for interaction between the particles themselves, some kind of collision detection needs to be implemented. Some possible approaches will be described in the next section.

7.4 Collision Detection

While the existing system allows for multiple, independent objects and detects collisions between the objects and the world walls, it is so far not able to detect collisions between the objects themselves.

Several different approaches to implementing inter-object collision detection exist. While most seem simple to implement, they should be capable of running completely on the GPU, as other approaches would again require transferring position data over the system bus to main memory and the CPU, thereby reducing the performance gain of calculating the entire simulation on the graphics card.

Full collision detection requires an intersection test for every pair of polygons in the scene. While this algorithm (like the others) can be optimized by assuming that all objects are closed and only surface polygons need to

be checked, it still requires $\frac{n \cdot (n-1)}{2}$ tests for n polygons and has therefore quadratic complexity, which is likely impractical.

The following major approaches to efficient collision detection exist:

7.4.1 Octree-/BSP-based Method

These methods allow quite efficient collision detection by subdividing space into a tree structure and only checking for collisions with objects in the appropriate leaf. However, this approach is mainly used for collision testing with a static world (e.g. walls of connected rooms), as the contents of the tree would have to be readjusted often in a dynamic environment.

7.4.2 Cell-based Method

Cell-based collision detection first subdivides space along a regular grid into small cells, usually cuboids. The polygons are first sorted with respect to containing cell and afterwards, all polygons inside a cell are tested against each other. Usually, a second iteration is done with a so-called 'staggered grid', in which the new cells are shifted in each direction by half a cellwidth.

While this approach provides a significant speedup, the sorting process provides a considerable obstacle to a GPU implementation. One algorithm that is suitable for stream computation is bitonic sort [Bat68].

However, as the triangle and spring maps rely on indices into the vertex map, it might be necessary to undo the sorting operation after collision detection, e.g. by storing the original map indices of each vertex and applying a second sort operation by original index. This allows the other maps to be used in the next step without modification.

7.4.3 Image-space Based Method

Image-space based collision detection ([Hei04]) approaches the problem from a different point of view. Basically, the space in which collisions might occur is rendered into a series of depth buffers, a process called depth peeling. The result is a layered depth image (LDI) [Eve01].

The following algorithm has, unfortunately, not been implemented due to lack of time, but seems entirely feasible:

Generate an LDI All surface polygons in the world are rendered into an LDI. Every pixel in this LDI should contain the following information:

- face direction (front- or back-face)

- index of the nearest vertex into the vertex map

Detect Collisions When collisions occur, the normal alternating sequence of back-facing and front-facing polygons is broken, and the offending polygons appear out of order. Thus, when comparing two successive depth layers, the same face direction in both layers signifies a collision.

The fundamental problem is now that although all collisions have been detected, the information is still stored in an image-space buffer. However, to actually perform a collision response, the indices of the affected vertices in the vertexmap are needed.

At this point, the vertex map indices in the LDI become necessary. In the final result buffer from this step, vertex map indices are stored, and each result vector contains either a null index (outside the vertex map), or the index of a vertex that is part of a colliding polygon.

Generate Collision Buffer Now, a buffer of the same dimensions as the vertex map is cleared with an arbitrary color c_1. The result buffer from the previous step is then bound as a vertex array and rendered into the cleared buffer with color c_2.

This operation results in a buffer in which every vertex that is part of a colliding polygon is marked with color c_2.

Update Vertex Buffer As a final step, a new vertexmap is generated. Where the collision map contains color c_2, the vertex position from the previous step is written, otherwise, the current position is used. Assuming that this is the first step in which the collision was detected, then the collision is resolved by reverting all involved vertices to their previous positions.

Chapter 8

Conclusion

In this thesis, a new approach to the simulation of elastic objects via mass-spring systems has been evaluated. By offloading the necessary calculations to the graphics processor that is optimized towards vector operations, a significant speedup by about one order of magnitude with respect to conventional CPU-based approaches is possible. This speed gain allows large models with thousands of vertices and tens of thousands of tetrahedra to be simulated in realtime, as the simulation is able to process up to 5.5 million tetrahedra per second.

As the implementation uses Superbuffers, a recently introduced approach to graphics memory management, it is possible to avoid any kind of bus transfer between main memory and the graphics card. This prevents the previously described speedup to be consumed again by heavy bus activity.

Issues that remain are stability and accuracy of the simulation. The stability problems can, to a certain extent, be solved by choosing a sufficiently small timestep for the integration method while still satisfying the realtime condition. However, the accuracy problems are likely caused by the reduced internal precision of the graphics processor.

On the way towards a complete simulation, collision detection is still missing. However, an image-space based approach might be feasible.

Bibliography

Mathematics

[Rie03] Peter Riegler, *Kurzeinführung in die numerische Integration*, FH Braunschweig, 2003

[Jue04] Ansgar Jüngel, *Das kleine Finite-Elemente-Skript*, Uni Mainz, 2004

[Sim04] Bernd Simeon, *Numerik gewöhnlicher Differentialgleichungen*, TU München, Zentrum für Mathematik, 2004

Physics

[Ver67] Loup Verlet, *Computer 'Experiments' on Classical Fluids*, Physical Review Vol. 159, No. 1, 1967

[Swo82] William C. Swope et al., *Physical Clusters of Molecules*, Journal of Chemical Physics Vol. 76, Page 637, 1982

Physical Simulation

[Gib97] Sarah Gibson, *A Survey of Deformable Modeling in Computer Graphics*, Mitsubishi Electric Research Laboratory, 1997

[Jak01] Thomas Jakobsen, *Advanced Character Physics*, Game Developer's Conference, Proceedings, 2001

[Kip04] Peter Kipfer et al., *UberFlow: A GPU-based Particle Engine*, Graphics Hardware Conference, 2004

[Gel98] Allen Van Gelder, *Approximate Simulation of Elastic Membranes*, Journal of Graphics Tools Vol. 3, No. 2, 1998

[Tes04] Matthias Teschner et al., *A Versatile and Robust Model for Geometrically Complex Deformable Solids*, ETH Zürich, 2004

Graphics

[Mac04] Rob Mace, *OpenGL ARB Superbuffers*, Game Developer's Conference, Proceedings, 2004

[Shr04] Dave Shreiner, *The OpenGL Programming Guide, Fourth Edition*, Addison-Wesley, 2004

[Eve01] Cass Everitt, *Interactive Order-Independent Transparency*, Nvidia Corporation, 2001

[CEv01] Cass Everitt, *Hardware Shadow Mapping*, Nvidia Corporation, 2001

[Kil00] Mark Kilgard, *Avoiding 19 Common OpenGL Pitfalls*, Game Developer's Conference, Proceedings, 2000

Collision Detection

[Kol04] Andreas Kolb et al., *Hardware-based Simulation and Collision Detection for Large Particle Systems*, University of Siegen, 2004

[Hei04] Bruno Heidelberger et al., *Detection of Self-Collisions with Image-Space Techniques*, ETH Zürich, 2004

Miscellaneous

[Bat68] K. Batcher, *Sorting Networks and Their Applications*, AFIPS Proceedings, 1968

[Si04] Hang Si, *TetGen File Formats*, Weierstrass Institute for Applied Analysis and Stochastics, 2004

[Tur04] Greg Turk, *The Stanford Bunny*, Stanford University, 1994

[Sur04] Suresh Venkatasubramanian, *The Graphics Card as a Stream Computer*, AT&T Labs Research, 2004

[Pfl04] Bernhard Pflesser, *Volume Cutting*, Universitätsklinikum Hamburg-Eppendorf, 2004

Index