Please see the color plate on page 323.

GPU Accelerated Surgical Simulators for Complex Morphology

Jesper Mosegaard* MSc
Department of Computer Science
University of Aarhus, Denmark

Thomas Sangild Sørensen† MSc, PhD
Centre for Advanced Visualization and Interaction
University of Aarhus, Denmark

ABSTRACT

Surgical training in virtual environments, surgical simulation in other words, has previously had difficulties in simulating deformation of complex morphology in real-time. Even fast spring-mass based systems had slow convergence rates for large models. This paper presents two methods to accelerate a spring-mass system in order to simulate a complex organ such as the heart. Computations are accelerated by taking advantage of modern graphics processing units (GPUs). Two GPU implementations are presented. They vary in their generality of spring connections and in the speedup factor they achieve.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture - Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Animation; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Physically based modeling; J.3 [Life and Medical Sciences]: Life and Medical Sciences - Health.

Keywords: Surgical simulation, Congenital heart disease, Spring mass systems, GPU, GPGPU

1 INTRODUCTION

Surgical simulators have the potential to improve the training of surgeons by presenting training scenarios in a virtual reality environment [1]. In this environment the surgeon can practice without the risks and the time pressure associated with real surgery. The surgeon is allowed to experiment and fail, learning from these experiences. A simulator is built around generalized anatomical models or alternatively it presents patient specific models for rehearsal of individual procedures. In the context of this paper we define surgical simulation as the calculation and visualization of tissue deformation in response to surgical tools interacting with parts of a virtual organ. We want to visualize this deformation in real-time, as well as make incisions in real-time.

Many methods of simulating deformation have been proposed previously [2]. One of these, the spring-mass model, is a physically based model often used in real-time surgical simulators [3]. In some areas of surgical simulation a high degree of morphological detail must be preserved to represent the involved organ(s) accurately. One such example is surgery on congenital heart defects. In spring-mass based systems such complex models have previously exhibited slow convergence to equilibrium. The simulation in these cases has been unrealistically slow.

The Graphics Processing Unit (GPU) is a programmable

*e-mail: mosegard@daimi.au.dk
† e-mail: sangild@cavi.dk

IEEE Virtual Reality 2005
March 12-16, Bonn, Germany
0-7803-8929-8/05/$20 ©2005 IEEE

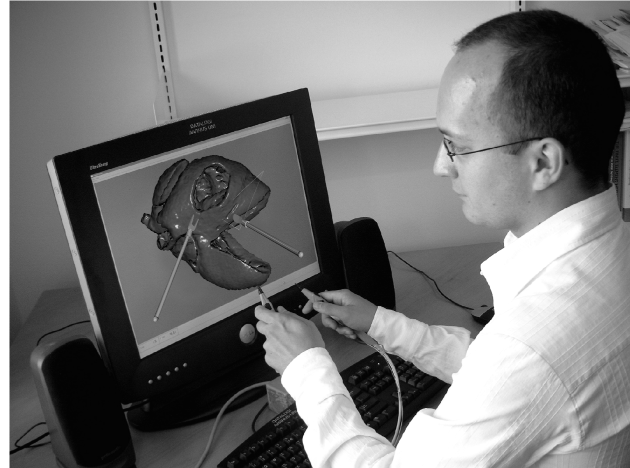

Figure 1. This picture shows the setup with styluses and the virtual environment for training surgical procedure in congenital cardiac surgery.

parallel processor capable of processing vertices and fragments in parallel. The GPU is designed to perform graphics rendering starting from geometric primitives and ending with pixel coloring. Recently the GPU has become programmable to a degree that makes it useful for general-purpose computation [4].

In this paper we present two spring-mass implementations solved entirely on the GPU. The purpose is to achieve a considerable speedup compared to existing CPU implementations due to the parallel processing capabilities of modern GPUs. Such acceleration could then be used to increase the convergence rate of the simulation and to increase the complexity of the simulated morphology.

Previously, simple spring-mass systems have been implemented on the GPU (e.g. [5]). However, they have been limited to simple shapes and primitive interaction. Necessary but slow data transfer from the GPU to the CPU has previously been a bottleneck when handling interaction and visualization. With the recent generation of GPUs (Geforce 6800, Nvidia, USA) simulation, interaction, and visualization can now be performed primarily on the GPU. The driving force of the presented implementations was the development of a surgical simulator (Figure 1) for complex interventions in congenitally malformed hearts [6][7]. Hence we apply the accelerated simulation on a pig heart derived from CT imaging data. The approach however is general and can be applied to other organs directly. The GPU used was a Geforce 6800, but the approach should work on any graphics card supporting shader model 3.0 features.

2 METHODOLOGY

2.1 The Spring-Mass Model

The spring-mass model is an often-used physically based model for surgical simulation when real-time behavior is desired. It is a discrete model of particles (with mass) and springs connecting these. Particles move in space according to Newton's second law

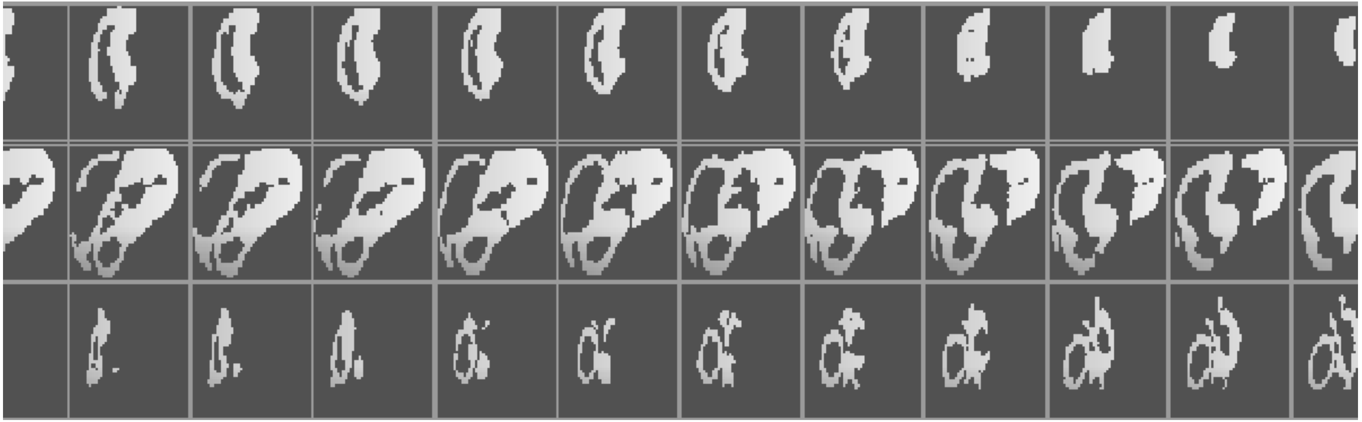

Figure 2. Part of the pig heart position texture containing 42.745 particles.

of motion constrained by springs, which connect pairs of particles. For a linear spring-mass model with damping the position of each particle x_i with mass m_i is given by the following 2nd order differential equation:

$$m_i \ddot{x}_i = -y_i \dot{x}_i + \sum_j g_{ij} + f_i \tag{1}$$

where y_i is the damping factor and f_i is the external forces. g_{ij} is the force vector defined by spring stiffness k_{ij}, spring rest length l_{ij} and particle positions x_i and x_j as:

$$g_{ij} = \frac{1}{2} k_{ij} (l_{ij} - \|x_i - x_j\|) \frac{x_i - x_j}{\|x_i - x_j\|} \tag{2}$$

Since $g_{ij} = -g_{ji}$, the computational cost can be minimized by evaluating the forces of each spring once, and adding this term to the summed forces of the connected particles. This system of differential equations can be solved by standard numerical integration schemes. One such method is Verlet integration [8] in which the particle positions at the subsequent time-step can be calculated as:

$$x(t+h) = 2x(t) - x(t-h) + \ddot{x}(t)h^2 \tag{3}$$

2.2 Parallel Computation of the Spring-Mass System

The GPU can in many ways be regarded as a shared memory parallel architecture, although with many limitations. We will begin by some considerations on the design of a simple parallel algorithm to solve a spring-mass system on a shared memory architecture. Fortunately the solution to (1) is straightforward to parallelize and solve iteratively. Processor P_i is responsible for updating information regarding particle x_i. At each time-step and for each particle the following must be done: Calculate spring forces based on the distance to neighboring particles and numerically integrate to calculate the new position of particle x_i.

Through the shared memory architecture, processor P_i can retrieve the current position of all particles for force calculation.

2.3 GPU Pipeline

When rendering a geometric primitive the vertex information is processed by a programmable vertex shader. Vertices are transformed and per vertex information is interpolated and transferred to the programmable pixel shader. The pixel shader processes this information as well as additional input in the form of textures to compute the final coloring of a fragment (a generalized pixel).

If we regard each fragment as a representation of a particle position we can design fragment programs to solve (1). An off-screen rendering buffer (PBuffer) is used to store the calculated positions. Each position in the PBuffer maps uniquely to the position of one particle. Each component of each particle position $x_i = (x, y, z)$ is represented as a 32-bit floating point value in the red, green and blue part of a pixel through OpenGL float-buffer related extensions. A PBuffer can be bound either as the rendering target or as a texture. Reading from textures and writing to the rendering target implements shared memory in fragment programs. The PBuffer containing positions will in the remaining paper be referred to as the *position texture*.

2.4 Integration loop on the GPU

The basic loop for integration is as follows. At any given time-step of the simulation, activate a dedicated fragment program by rendering a single quad covering the entire PBuffer. In this program, let the supplied texture coordinates implicitly provide particle positions through lookups in the position texture. The texture coordinates are given at the vertices of the quad and automatically interpolated throughout all fragments. A one-to-one mapping of the PBuffer between the input texture and output buffer is then established.

The choice of the numerical integration method is influenced by the properties of the fragment program's input and output. Verlet integration is well suited for our GPU implementation since calculations of future positions depend exclusively on the previous two positions. The two previous particle positions can be provided as input through textures. These textures are available from previously written PBuffers. Verlet integration result is only one vector, the position, and consequently conserves bandwidth compared to numerical integration resulting in more than one vector. Furthermore, the Verlet method is inherently stable at large time-steps, a desirable property for a real-time application.

2.5 Spring-Mass Simulation on the GPU

In this section we explain in detail two spring-mass implementations for the GPU. The first method represents spring connections explicitly, while particles in the second method are connected implicitly based on their location in a three dimensional grid. The explicit method allows the most freedom in representing spring connections and particle locations at the cost of some simulation speed compared to the implicit alternative.

2.5.1 A Spring-Mass System with Explicit Connections

In a general spring-mass model every particle can be arbitrarily connected to other particles with no special order assumed. We encode this spring connectivity in a floating point texture, in the following referred to as the *connectivity texture* (Figure 3). This connectivity texture defines for each particle p_i a list of springs to

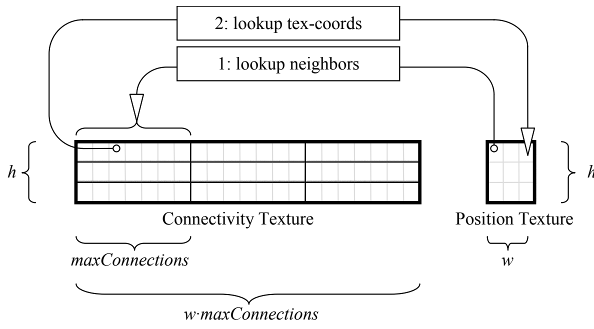

Figure 3. Layout of the connectivity texture. For a fragment *(x,y)* in the position texture, the connectivity texture contains the texture-coordinates of neighbors in coordinates *(x·maxConnections , y)* to *((x+1)· maxConnections , y)*.

particles p_j by storing 1) the texture coordinate in the position texture of each particle p_j and 2) the rest length l_{ij} and stiffness k_{ij} The connectivity texture remains constant as long as no changes in particle connectivity are made. From an element of the connectivity texture a texture lookup provides the neighboring particle position x_j. By iterating over all springs connected to particle p_i it is possible to calculate (2). If a node has less than the maximum number of neighbors, the spring rest length or stiffness can be set to 0 to indicate that a given element of the connectivity texture should not be considered.

Note that the spring forces will be calculated twice (with different sign), from each particle connected to it. We have chosen not to take advantage of the fact that $g_{ji} = -g_{ij}$, as it would require a second pass. Additional information would also have to be given to each fragment to indicate the sign of the force of a given spring.

2.5.2 A Spring-Mass System with Implicit Connections

A potentially limiting factor of the approach of explicit connections is the intense use of texture lookups to retrieve the positions of neighbor particles. What if the texture coordinates of neighbors could be given directly as input to the fragment program to enable a single texture lookup to retrieve neighbor positions? Using interpolated texture coordinates to index neighbors is possible if particles are fixed in a regular three-dimensional grid. Particles are then regarded as connected if they are neighbors in the grid. There is no explicit connectivity texture as in the previous method. We chose to connect each particle to 18 neighbors as depicted in Figure 4 to constrain both shear and structural changes.

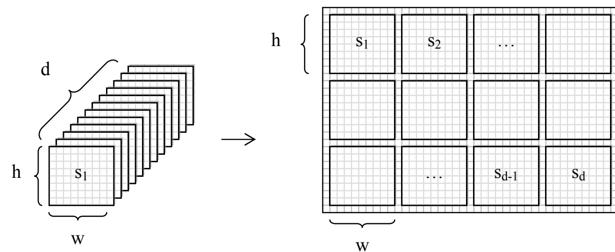

Figure 5. The flat 3d-texture approach. The 3D volume of voxels is mapped to a 2d texture by laying out each of the *d* slices of size *h·w* in the 2d texture one after another. The slices are padded with elements containing unique alpha values of zero to detect the volume borders.

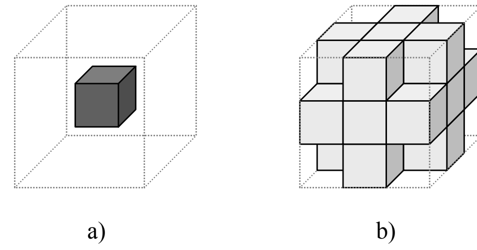

Figure 4. Particle connectivity in a 3D grid. Each particle a) is connected to 18 neighbors b).

As stated previously, fragment programs are run in response to geometry being rendered. For this to happen in the case of particles in a 3d-grid, we defined a mapping to a 2d PBuffer. This mapping is based on the flat-3d texture method [9]. Beginning with the bounding cube of the 3d-grid, we define slices through the depth of the cube. These slices are mapped consecutively to the 2d texture (Figure 5). For an example of a position texture based on the flat-3d approach see Figure 2.

Texture coordinates indicate the locations of neighbouring particles in the position texture. The coordinates are fixed offsets from the fragment positions and are given at vertices of the rendered quads. Interpolation ensures that all fragments get the correct offset to connected particles in relation to their own position. Figure 6 depicts this process. The 18 texture locations of neighbors plus the location of the particle itself are given through 8 texture coordinates (with 8·4=32 values). Identical components of neighbor locations can be reused. To handle border cases correctly when laying out in several rows, it is necessary to render five quads as illustrated in Figure 6. Texture coordinates take into account the wrapping and will ensure correct neighbor relations. Neighbors within a slice are addressed by offsetting the fragment position by ±1 along the width or height of the texture. Particles on the border of a slice are not intended to be connected to the neighbors, though. To alleviate this problem we pad the slices with inactive particles that we do not seek to process, notice the padding in Figure 5. We define inactive particle fragments to have an alpha value of zero. This value is detected in the fragment program, and the calculation skipped for the associated springs.

Neighbors between slices are addressed by furthermore offsetting with either w+1 or $-(w+1)$ to reach a neighbor in the previous and next slice respectively.

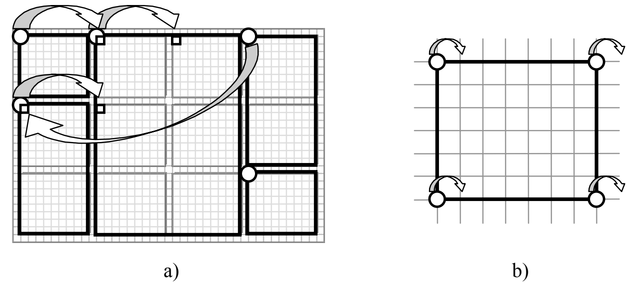

Figure 6. Vertex texture coordinates for neighbor positions. For clarity we will only show two of the 18 possible texture coordinates. a) depicts the five quads rendered to ensure correct wrapping. The texture coordinate pointing to the neighbor directly below in depth is shown for the top-left vertex. b) shows the four texture coordinates given with all quads for the right neighbor.

By this implicit connectivity g_{ij} in (2) can be determined using only 1 texture lookup per neighbor instead of 2 per neighbor as was the case in the explicit method. Furthermore we have fixed the possible rest lengths to all neighbors to either 1 or √2. We can hereby simplify the force computation somewhat if we assume all stiffness coefficients k_{ij} to be equal to the constant k:

$$\sum_j g_{ij} = \sum_j \frac{1}{2} k_{ij} \left(l_{ij} - \left\| x_i - x_j \right\| \right) \frac{x_i - x_j}{\left\| x_i - x_j \right\|}$$

$$= \frac{1}{2} k \left\{ \begin{array}{l} \left(\sum_{j \in D_1} \frac{x_i - x_j}{\left\| x_i - x_j \right\|} - \sum_{j \in D_1} x_i - x_j \right) + \\ \left(\sqrt{2} \sum_{j \in D_2} \frac{x_i - x_j}{\left\| x_i - x_j \right\|} - \sum_{j \in D_2} x_i - x_j \right) \end{array} \right\} \qquad (4)$$

D_1 is the set of neighbor particles with rest length 1 and D_2 is the set of neighbor particle with rest length √2. Using this expression as the basis for the fragment programs saves instructions. Equally important we have a sum of unit-vectors, which we can later use to calculate normals after deformation.

Naturally, a cube of connected particles does not constitute a flexible shape. To support complex morphology we recognize that arbitrary fragments can be inactive particles by setting their alpha value to zero (as in the case of the padding). Within the resolution of the grid, any morphology can be modeled in this fashion. Unfortunately the size of the position texture has at least the number of elements as the bounding cube surrounding the model. A vast majority of fragments are potentially inactive particles. They are processed automatically however, since they are positioned within a rendered quad. Even though we can detect these particles in the fragment programs, there is some overhead involved in doing so. To overcome this problem we initially "fuse" an image of the model in the OpenGL depth buffer *once*, and set up a depth-buffer test to fully eliminate processing inactive particles.

2.6 Visualization and Approximate Normals

In the previous sections we have presented two methods to calculate the particle positions on the GPU. The next step is to visualize the result. This has two aspects: How to construct and render a surface model of the morphology, and how to deform this surface based on the newly calculated particle positions.

A surface model is reconstructed with the marching cubes algorithm [10]. Next we define a mapping from each surface vertex to one particle in the spring-mass system. With this mapping the issue of deforming the surface is reduced to transferring the calculated particle positions to the related surface vertices. We cannot simply read back the particle positions to the CPU and set vertex positions accordingly, as this would be a major performance bottleneck. Instead we utilize a new feature in the Vertex Shader 3.0 design: texture lookups in vertex programs. The position of each surface vertex to be rendered is found by a texture lookup in the position texture. To render the model we initially set up a display-list, which renders the surface geometry at its original rest positions. We pass one texture coordinate per vertex to provide the coordinates to lookup in the position texture. A vertex program then fetches the most recent particle position, does basic transformation and outputs the deformed position.

An important issue in this visualization is the calculation of surface normals when shading the surface. In a conventional CPU application one would often approximate vertex normals by averaging the adjacent face normals. Face normals are found by reading the positions of nodes making up the face and calculating

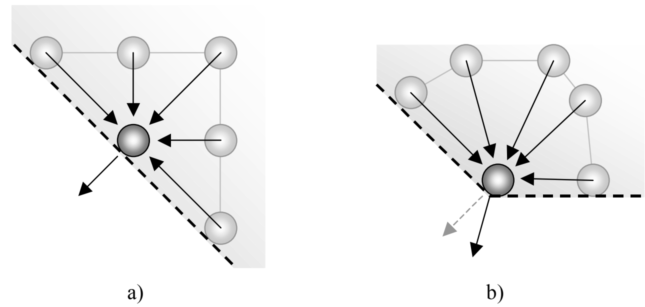

Figure 7. A 2d example of normals calculated on the basis of unit-vectors from neighbors. a) the original geometry. b) the deformed geometry.

the normal of the plane. In the presented GPU approaches to a spring-mass system we do not have the necessary surface information to simply reuse the CPU approach. Still it is not an option either to read back the position texture each frame due to performance reasons; the calculation of normals must take place on the GPU. In [11] a general method for the calculation of normals for discrete surfaces is presented. Our approach is approximate but simple and fits elegantly with the way we calculate forces. We approximate the normals by the normalized sum of the unit vectors pointing from neighbors to the particle in question (Figure 7):

$$n_i = normalize\left(\sum_j \frac{x_i - x_j}{\left\| x_i - x_j \right\|} \right) \qquad (5)$$

Since we already computed the sum of the unit vectors in the force calculations, we only need to normalize this vector and save it. This approximation gives us well behaving normals almost for free. We pack the 3-tuple normal into the 32 bit alpha channel of the fragment. The normal is read by the visualization vertex program and send to the fragment program for per pixel lightning. To add shading details we normal-map the rendered geometry based on a reconstructed model with a very high level of detail (Melody, Nvidia, USA).

2.7 Interaction

To interact with the simulation the system uses two Polhemus Fastrak styluses with two buttons each (Figure 1). Each stylus provides position and orientation of the instruments in space. We support three modes of interaction; probing, grabbing and cutting. The main issue in all these interaction methods is to avoid communication between the CPU and GPU every frame, as this will introduce a bottleneck in the application.

2.7.1 Probing

When probing, all particles seek to remove themselves from the volume of space covered by the interaction instrument. Currently our probing tools are represented as spheres. The fragment program calculates intersection and intersection response. Every fragment simply checks if the particle is inside a globally defined set of spheres. If this is the case the particle is projected to the sphere surface.

2.7.2 Grabbing

All particles have a state indicating whether they are grabbed or not. When the grab-button is pressed all particles are checked for intersection with a bounding sphere of the grabbing tool. If a particle is intersected the state of the particle is changed to 'grabbed'. When particles are grabbed, they will set their position

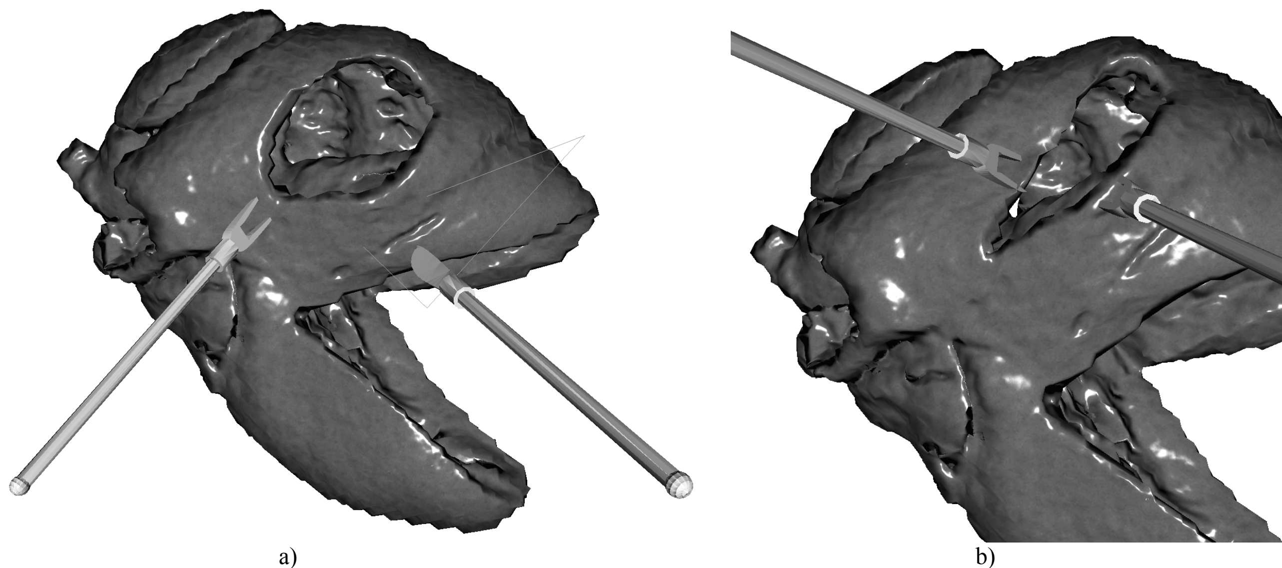

Figure 8. A Pig heart consisting of 42.745 particles in a regular grid reconstructed from a CT data set. a) original geometry b) deformed by grabbing.

relative to the center of the grabbing device for as long as the grab-button is active.

We have chosen to resolve the grab state of particles on the CPU. When the grab-button is pressed we do a read-back of the position texture *once*. The CPU knows which particles are grabbed and what their positions should be relative to the interaction device. This information is rendered back into the position texture *after* the integration, overwriting that calculated position, and *before* cycling of the PBuffers. When the grab-button is released the additional rendering is no longer necessary. Since the position texture is only read back at the beginning of a grab there is no noticeable slowdown in the simulation. The final result is that grabbed particles will move relative to the interaction device and neighboring particles will follow due to the springs connecting these particles.

It is possible to handle grabbing entirely on the GPU. This leads to more fragment operations per simulation iteration however, and results in lower simulation rates.

2.7.3 Cutting

A cutting action defines an incision in the spring-particle connections, removing or altering connections so that there are no structural constraints across the incision. The geometry should furthermore follow the incision as closely as possible.

Since the surface faces are not directly represented in the fragment programs for the spring-mass computations, updating the surface geometry is done on the CPU with up-to-date nodal positions. Before each cutting procedure the position texture is read back to the CPU once. Collision detection can be done on the CPU or GPU [12] but collision response in the form of structural changes is handled on the CPU.

When using explicit connections, cutting is simply a transfer of CPU based cutting schemes [13] since we represent both particles and springs. Structural changes can be propagated to the GPU by simply writing into the position texture and connectivity texture. When using implicit connections we have no representation of springs and we cannot insert particles between other particles. A simple cutting scheme would erase particles and hereby the springs connecting the particle to its neighbors. This can be accomplished by writing an alpha value of zero into the fragments corresponding to the particles we wish to erase. This will mark the fragments as an inactive particles. The incision would then be at least as wide as the length of two relaxed springs. We propose instead a method of cutting in the implicit model that improves the granularity; we instead erase individual springs. To erase a spring we render a special fragment-sized quad at the position of the two associated fragments. The fragment program is unchanged. We erase springs by giving the value of zero for those texture coordinates (out of the 18) related to erased springs. A zero texture coordinate results in reading the texture padding and consequently indicates that there is no connection. This approach of cutting results in a growing number of quads rendered proportional to the number of erased springs. In many simulations the number of quads will not grow excessively however.

3 RESULTS

This section presents performance measures of the described GPU spring-mass systems and compares them to a CPU implementation. The GPU based spring-mass systems were implemented in CG, OpenGL arbfp1 (including fragment_program2 features) as well as Visual Studio .net 2003 C++. The CPU spring-mass system used for comparison is a pure CPU implementation of the spring mass system with implicit connections. The system was implemented in Visual Studio .net 2003 C++ and compiled with all optimization flags set and optimized for speed. The tests were run on a Pentium IV 3GHz machine with a Gainward Geforce 6800 Ultra graphics card.

As a case study for the GPU-based surgical simulator we extended the Cardiac Surgery Simulator [7] to support the GPU spring-mass model (Figure 8). The morphology is based on CT-images of a pig heart. The CT-data consists of 400x400x395 voxels in isotropic resolution of 0.6 mm in all axes. From the high-resolution dataset we resampled a reduced resolution dataset of 100x100x100 voxels. The low resolution dataset was used in a marching cubes reconstruction of one surface geometry and to create a regular grid of particles for the spring-mass simulation. If a voxel in the low resolution dataset indicated tissue, we created a particle at that location in the regular 3d-grid of particles. In total, this provided 42.745 particles (Figure 2). The surface reconstructed with the marching cubes algorithm consists of 31.320 faces. To add further visual detail from the high resolution CT-dataset we calculated a normal-map from a 630.000 polygon reconstruction of the high resolution CT-dataset. The simulator

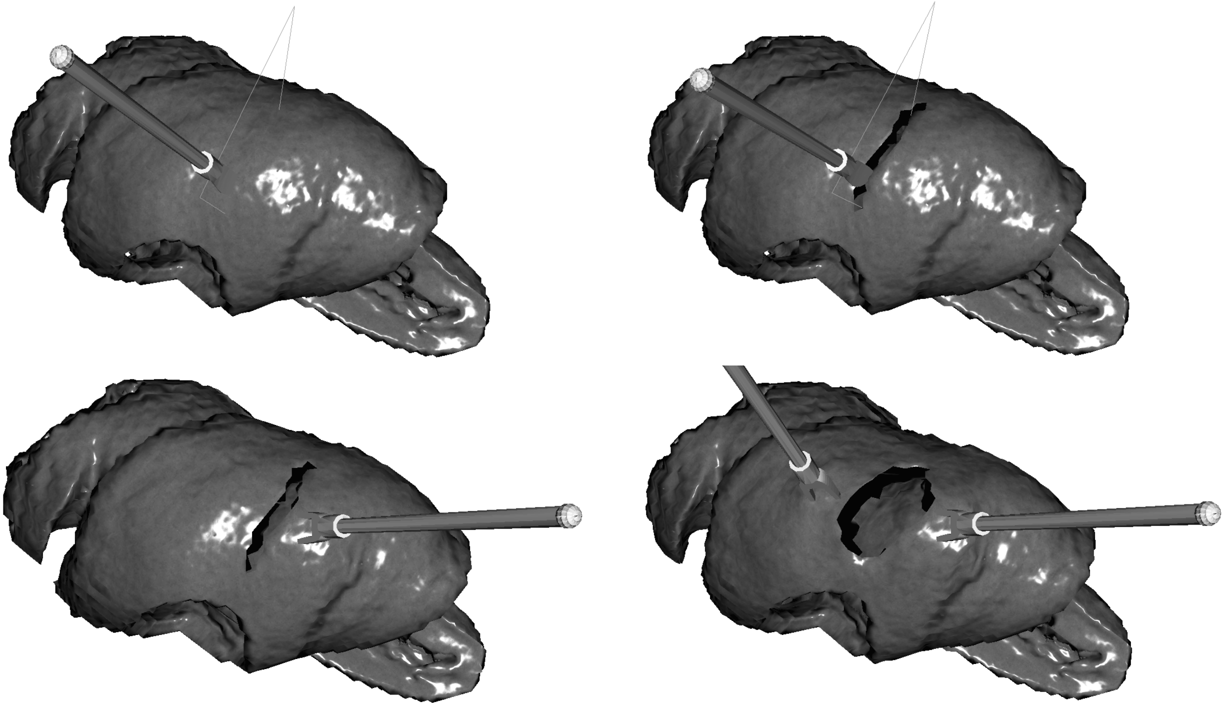

Figure 9. Cutting in the implicit model. The incision is as wide as the rest length of the springs.

was set-up to do four simulation iterations per visualization frame. In this setup the simulator was able to run at 47 frames per second. With four simulation iterations per frame this corresponds to 188 iterations of the simulation per second. Figure 8 shows deformation of the morphology while Figure 9 shows cutting. We furthermore refer to the accompanying video for a demonstration of the system.

Performance results for different sizes of GPU spring-mass systems in comparison with the CPU implementation are reported in Figure 10 and Table 1. Performance results for the implicit GPU cutting are reported in Figure 11.

4 DISCUSSION

As can be seen from Table 1 and Figure 10 the GPU clearly outperforms the CPU in computation of a spring-mass system.

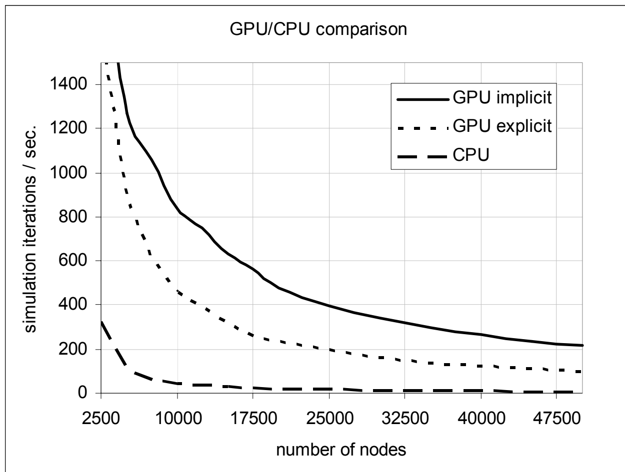

Figure 10. The GPU spring-mass systems in comparison to a CPU implementation.

Comparing the implicit GPU method to the CPU implementation we have a speedup of up to a factor 30. The GPU achieves this computational speedup since we successfully expressed the spring-mass algorithm in the language of the very specialized fragment processor. The GPU method with explicit connections runs at about half the speed of the implicit method. There is clearly a trade-off between speed and the generality of connectivity in the spring-mass system.

With a GPU implementation of the spring-mass system we use the GPU hardware for both visualization and calculation of deformation. It is important to realize that this will not slow down visualization overall compared to a CPU implementation. On the contrary it might be faster; The GPU implementations render the surface geometry through display lists allowing for GPU caching of the geometry. A CPU implementation cannot use this technique because new vertex positions are calculated every frame, and these need to be send to the GPU.

With the increased number of iterations available per second, we can achieve faster convergence when simulating physically based deformable geometry. We can iterate several times in the simulation loop before rendering the result to the screen, still in

Table 1. Performance comparison for the CPU, Explicit Connections GPU and Implicit Connections GPU. In the 2nd , 3rd and 4th column we present iterations per second. GPU/CPU columns present the GPU speedup in comparison to the CPU.

Nodes / method	CPU	Implicit GPU	Implicit GPU / CPU	Explicit GPU	Explicit GPU / CPU
10.000	45,8	839,8	18,7	457,1	10,2
20.000	20,2	476,9	23,6	234,4	11,6
40.000	9,9	264,6	26,9	121,0	12,3
50.000	7,8	218,0	28,1	99,0	12,7
100.000	3,3	104,1	31,4	48,5	14,6

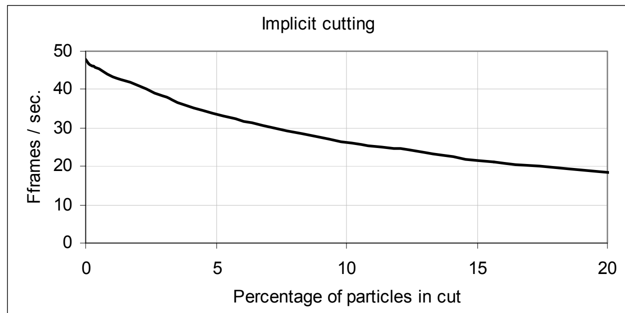

Figure 11. For the implicit method we proposed a cutting scheme that rendered additional quads to exclude individual springs. This graph shows the relationship between the size of the cut as percentage of the whole geometry of 42.745 particles and the frame rate. The frame-rate is for rendering the pig heart morphology with four iterations of simulation per visualization.

real-time. Furthermore we can achieve greater stability and precision of the numerical methods by using smaller time-steps than previously. Finally, the added computational power could also be used to process larger geometries with added degree of detail, enabling more realistic surgical simulations.

If we consider a standard spring-mass implementation, there are many improvements that can accelerate the simulation on the CPU. These might, however, not be easily ported to the GPU. Hence, the presented speedup is not to be interpreted as a speedup compared to the fastest CPU implementation available.

The performance of the Cardiac Surgery Simulator in the reported case study resulted in real-time visualization and fast convergence of the deformed tissue. The geometry was furthermore detailed enough to represent the complex morphology to a higher degree than previously. For this kind of surgical simulation the growing number of quads rendered to support incisions is a minor performance issue since using few and small incisions is a primary goal. The granularity of cuts in the implicit method is not a big problem for the incisions in the heart since these are large compared to the detail of the spring-mass system.

4.1 Perspectives

As a next step in the range of interaction modalities, future research will investigate how the GPU implementation of the spring-mass algorithm can effectively support suturing.

In this paper we defined a simple mapping between surface geometry and the underlying spring-mass simulation. When texture lookups in vertex programs eventually support interpolation of the texture values, more advanced mappings are possible. The visible geometrical surface might then become more detailed than the underlying simulation but still deform based on the simulated particles. Additionally, when surface vertices become detached from the particle positions, we can also split the cutting operation in two: The cutting of the particle system and the cutting of the surface geometry. This would enable us to visualize very detailed cuts controlled by a less detailed particle system.

In cases where the spring-mass model is not considered adequate, other physically based models of deformation could be ported to the GPU following the principles of this paper. A Finite Element Model could be implemented as a dynamic simulation in much the same way as the spring mass model, but requiring a larger number of neighbor lookups as well as neighbor dependent stiffness coefficients.

It would also be very interesting to look into the possibilities of automatic level of detail in the spring-mass simulation using hardware accelerated linear interpolation of texture lookups. Such a representation would probably include some hierarchical representation of particles on the GPU, and could possibly be used as an acceleration structure for intersection tests without the transfer of data to the CPU.

We have successfully implemented an accelerated surgical simulation on the GPU. Running the GPU based simulation does not utilize the CPU fully though. When the communication between the CPU and GPU becomes faster, i.e. through the PCI-express standard, the CPU and GPU should work more closely together on the computational problems. It will become important to identify which problems are best suited for the graphics processing unit and which are well suited for the central processing unit.

4.2 Acknowledgements

We acknowledge Vibeke Hjortdal, MD and Ole Kromann Hansen, MD for clinical support and guidance throughout the development of the surgical simulation. For the data acquisition we acknowledge the contributions of Dr. Gerald Greil, University of Tübingen, Germany as well as Dr. T. Flohr and Dr. I. Wolf.

References

[1] Richard M. Satava. *Accomplishments and challenges of surgical simulation*. Surg Endosc;, 15(3), pp 232-41, 2001.

[2] Sarah F. F. Gibson and Brian Mirtich. *A Survey of Deformable Modeling in Computer Graphics*, MERL Technical Report, TR97-19, 1997

[3] Kevin Montgomery, et al. *Spring: A General Framework for Collaborative, Real-time Surgical Simulation*. Medicine Meets Virtual Reality 11, pp 23-26, 2002.

[4] Chris J. Thompson , Sahngyun Hahn and Mark Oskin. *Using modern graphics architectures for general-purpose computing: a framework and analysis*. Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture., pp 306—317, 2002.

[5] Simon Green. *OpenGL Shader Tricks*. Game Developers Conference, 2003.

[6] Thomas S. Sørensen, Erik M. Pedersen, Ole K. Hansen, Keld Sørensen. *Visualization of morphological details in congenitally malformed hearts*. Cardiol Young; 13(5), pp 451-60, 2003.

[7] Jesper Mosegaard. *LR-spring-mass model for cardiac surgical simulation*. Medicine Meets Virtual Reality 12, pp 256-258, 2003.

[8] Loup Verlet. *Computer Experiments on Classical Fluids. I. Thermodynamical. Properties of Lennard-Jones Molecules*. Physical Review, Vol. 159, pp 98–103, 1967.

[9] Mark J. Harris, William V. Baxter III, Thorsten Scheuermann and Anselmo Lastra. *Simulation of Cloud Dynamics on Graphics Hardware*. Proceedings of Graphics Hardware, pp 92-101, 2003.

[10] William E. Lorensen and Harvey E. Cline. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics (Proceedings of SIGGRAPH '87), Vol. 21, No. 4, pp. 163-169, 1987.

[11] Grit Thürmer & Charles A. Wüthrich. *Normal Computation for Discrete Surfaces in 3D Space*. Eurographics 97. Volume 16 (1997), Number 3, pp 15-26.

[12] Naga Govindaraju, Stephane Redon, Ming C. Lin and Dinesh Manocha. *CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware* , ACM SIGGRAPH/Eurographics Graphics Hardware, pp 25-32, 2003.

[13] Han-Wen Nienhuys and A. Frank van der Stappen. *A Surgery Simulation Supporting Cuts and Finite Element Deformation*, Medical Image Computing and Computer-Assisted Intervention, pp 153-160, 2001.