A GPU Accelerated Spring Mass System for Surgical Simulation

Jesper MOSEGAARD[#¤], Peder HERBORG[¤], and Thomas Sangild SØRENSEN[¤]
[#]*Department of Computer Science,* [¤]*Centre for Advanced Visualization and Interaction, University of Aarhus, Denmark*

Abstract. There is a growing demand for surgical simulators to do fast and precise calculations of tissue deformation to simulate increasingly complex morphology in real-time. Unfortunately, even fast spring-mass based systems have slow convergence rates for large models. This paper presents a method to accelerate computation of a spring-mass system in order to simulate a complex organ such as the heart. This acceleration is achieved by taking advantage of modern graphics processing units (GPU).

1. Problem

In recent years simulators have been introduced in the surgical curriculum in several fields [1]. Many surgical simulators used in practice are based on spring-mass deformable models [2] due to performance reasons. The spring-mass model is considered physically based and achieves real-time visualization and fast convergence for geometry of moderate size.

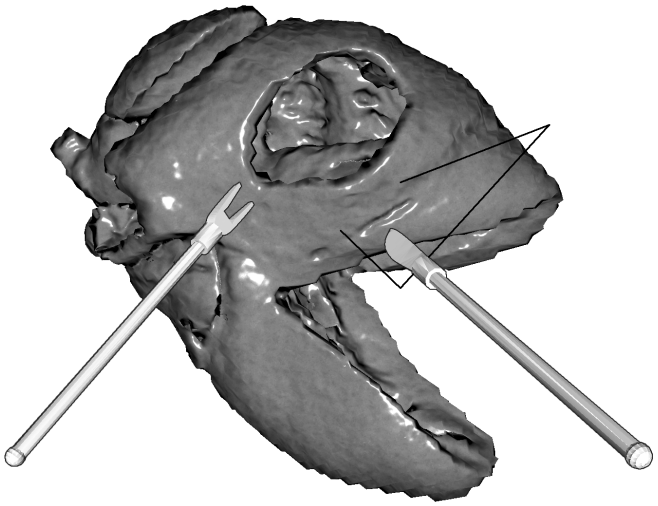


Figure 1. The Cardiac Surgery Simulator on a pig heart

In surgical simulation in general, there is a tradeoff between the costs of calculations, how realistic the tissue-deformation is reproduced, and how detailed the morphology being simulated appears. It is the goal of this paper to simulate a very high degree of morphological detail in real-time. As an example, the cardiac morphology is complex and requires a high degree of geometric detail to be modeled accurately.

We present a surgical simulator based on a spring-mass system accelerated by an implementation on the graphics processing unit (GPU). The purpose is to achieve a considerable speedup due to the parallel processing capabilities of the GPU [3]. This acceleration could be used to increase the accuracy and convergence of the numerical calculations and to increase the complexity of the simulated morphology. Previously, simple spring-mass systems have been implemented on the GPU (e.g. [4]). However, they were limited to simple shapes. Slow data transfer from the GPU to the CPU has been an

additional bottleneck when handling interaction and visualization. With the recent generation of GPUs (Geforce 6800, Nvidia, USA) simulation, interaction, and visualization of a spring-mass based surgical simulator can be accelerated on the GPU. To our knowledge we present the first implementation of a fully GPU-based surgical simulator. The driving force behind the current research is the development of a virtual training system for complex interventions in congenitally malformed hearts [5][2], see Figure 1. The approach however, is general and can be applied to other organs directly.

2. Methodology

2.1 Spring-Mass System

The GPU spring-mass implementation is based on the basic linear spring-mass formulation where each particle x_i with mass m_i is given by the following 2nd order differential equation:

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

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:

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

The differential equation can be solved with standard numerical methods, such as the verlet integration [6]:

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

2.2 GPU Pipeline

The focus of this paper is to express the calculation of the spring-mass system effectively in terms of the hardware accelerated features of the GPU. Recently, the vertex processor and fragment processor have become programmable. Both processors are parallel processors with a number of pipelines working simultaneously. Vertex and fragment computation can depend on previous iterations through texture lookups and render-to-texture functionality exposed through Pixel Buffers (PBuffers). The PBuffer can be bound as the rendering target and as a texture. Throughout this paper we will refer to the PBuffer as a texture or as the rendering target interchangeably depending on the context. Using floating point texture extensions we can do computation on IEEE 32 bit floating point numbers. These features enable general purpose computation on the GPU.

2.3 A Spring-Mass System with Implicit Connections

To calculate spring forces and perform verlet integration a fragment program was developed. The fragment processor was chosen as there are generally more fragment pipelines available than vertex pipelines. Equally important, texture lookups are more efficient in fragment programs. We associate the position of each particle with a single fragment in a PBuffer. The PBuffer is referred to as the *position-texture*. The fragment program is responsible for calculating forces affecting each particle, doing verlet integration, and outputting the calculated position to the associated fragment in the

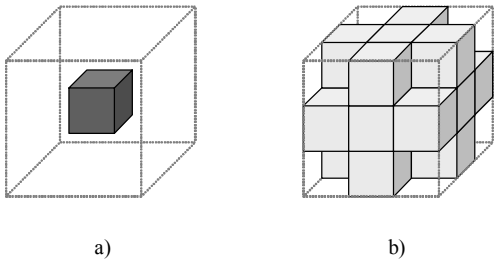


Figure 2. Particle connectivity in a 3D grid. Each particle a) is connected to 18 neighbors b) (blocking the black particle)

position-texture. Each fragment receives a texture coordinate as input, which gives the position of the associated particle through a texture lookup.

To calculate the forces affecting particles we need to fetch the position of neighboring particles connected through springs. The most important choice in our implementation and the major source of the performance we achieve is that we use only one texture lookup to obtain the position of each neighbor particle. The texture coordinates needed to lookup neighboring particles is given directly as input to the fragment program from the output of the vertex program. To avoid that the vertex processor becomes a bottleneck by rendering individual fragments as geometry, we conceptually invoke the fragment computation with a single quad covering the position-texture. Texture coordinates are specified for each vertex and interpolated automatically by the rasterizer before being received as input in the fragment programs. This means that particles must be connected in such a way that their neighbors can be fetched from per vertex interpolated texture-coordinates. That is, particles should be connected in a fixed pattern. We use a 3D grid as depicted in Figure 2 to construct a spring-mass system with eighteen springs constraining axis aligned changes as well as shearing.

The grid must be mapped to the two-dimensional position-texture to use the proposed approach. This is achieved through a derivation of the flat 3d-texture approach [7], see Figure 3. Each vertex rendered to invoke fragment computation will be given eighteen texture coordinates offset a fixed amount from the texture coordinates identifying the particle, see Figure 4. Instead of the conceptual model of rendering only one quad to invoke full fragment computation, it is necessary to render five quads with texture-coordinates constructed to take into account the border-cases of the flat 3d-texture approach.

After each iteration, the PBuffer that was rendered to is bound as a texture and used for input to subsequent iterations. The verlet integration depends on the previous two calculated positions; consequently we cycle three PBuffers containing the old, current and new positions.

As in [4] the geometry is connected in a regular grid. Unlike [4] however, we operate on a 3D grid. The grid must furthermore approximate an arbitrary geometry. Hence, it is necessary to exclude some of the particles in the grid. Conceptually we carve out the morphology in the grid of particles. Grid points are active particles in the simulation if inside the myocardium or a vessel wall and otherwise discarded with a depth-buffer based cull. See Figure 5 for an example.

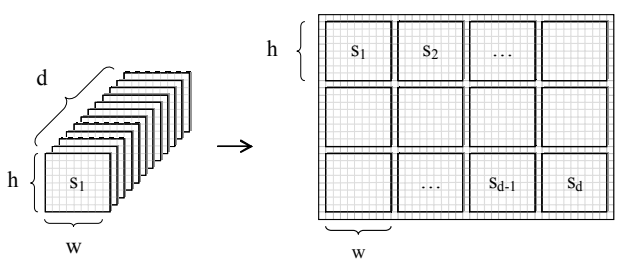


Figure 3. The flat 3d-texture approach. The 3D volume of voxels is mapped to a 2d texture by laying out each of the *d* slices of size *h·w* in the 2d texture one after another. The slices are padded with elements containing unique alpha values of zero to detect the volume borders.

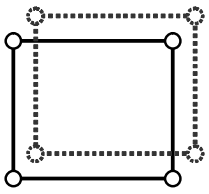


Figure 4. The solid box represents the quad drawn to invoke fragment processing. Solid spheres represent texture coordinates to the particles themselves. The dotted box and spheres represent one of the eighteen neighbors; the top left neighbor offset with texture coordinate (1,-1) in comparison to the solid box.

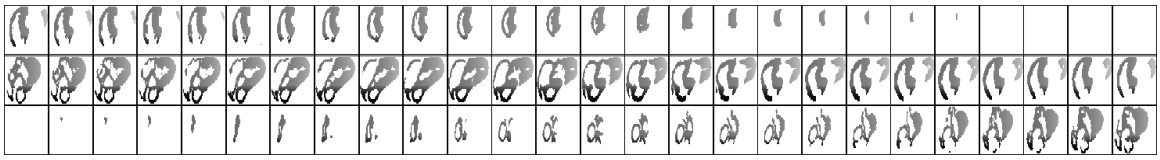


Figure 5. The position-texture of a 42.745 particle pig heart. White areas are grid points not associated with particles.

2.4 Visualization and Interaction

To visualize the calculated positions we need to define vertex positions of a surface based on position-texture values. Since a large amount of grid-points are not associated to particles, a visualization based on vertex texture fetches is advantageous compared to a transfer of the entire position-texture to either the CPU or directly to a vertex buffer. Through vertex texture fetches we transfer only particles that are part of the surface of the mesh. The geometry specified to the 3D API to visualize the current simulation-step is a static mesh where each vertex is associated to a particle through a per vertex specified texture coordinate. Through texture lookups in the vertex program we can fetch the current position of the particle. We hereby defined a mapping from one surface vertex of the visualization to one particle on the surface of the spring-mass system.

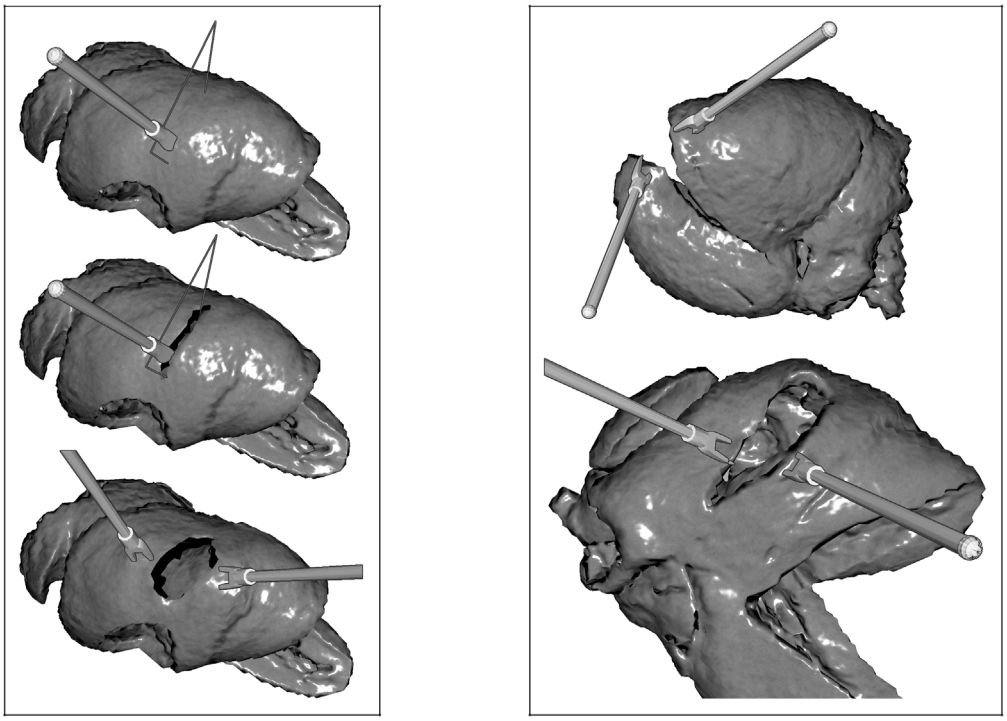
The vertex-normal (indicating the curvature of the surface) to be used for shading is approximated by the normalized sum of all normalized vectors from particle neighbors to the particle in question. This value is already calculated as part of the force computation, packed into the position-texture as the alpha component.

To handle grabbing, the collision detection is done on the CPU based on a single read-back of the position-texture when grabbing is initiated. Subsequently we render the position of grabbed particles, based on the interaction device, as geometric primitives directly into the fragments corresponding to the grabbed particles. We hereby override the simulation results.

The collision detection for cutting is also done on the CPU based on a single read-back of the position-texture. As a result of cutting, we furthermore need to change the static mesh rendered for visualization. Because the connectivity between particles is implicitly based on their location in the position texture, the smallest incision possible in the proposed model is two springs wide – by removing a particle. To support incisions as small as a single spring, we extend the proposed model. If a spring is erased, we will setup the invocation of fragment computation so that the connected particles receive invalid texture coordinates for that spring whereby the spring is considered non-existing in the fragment program doing the spring-mass computations. This means that we need to render additional geometric primitives for each particle that is missing springs. The added granularity comes at the cost of performance because additional vertex processing and fragment processing is necessary – this approach is advantageous when we only make small cuts in the morphology.

2.5 Hardware and Test-case

A Gainward CoolFX Ultra/2600 graphics card in a Pentium 4 3 GHz was used for the presented simulation and visualization. A detailed (630.000 faces) model of a pig heart was reconstructed from a CT dataset using the marching cubes algorithm [8]. Additionally, a model with lower resolution (42.745 grid points) was obtained. The spring-mass simulation was performed on the latter which was normal-mapped to visually appear as detailed as the higher resolution model (Melody, Nvidia, USA).



a) b)

Figure 6. A pig heart consisting of 42.745 particles in a regular grid reconstructed from a CT data set. We illustrate a) cutting and b) deformation by grabbing.

3. Results

The GPU spring-mass system was implemented in OpenGL, C++, Cg, NV_fragment_program2 and NV_vertex_program3 and compiled with Visual Studio C++. As a comparison for the GPU implementation a CPU implementation was implemented in C++ and compiled in Visual Studio C++. The CPU spring-mass system is a port of the GPU implementation. The resulting performances can be seen in Table 1.

We have extended the Cardiac Surgical Simulator [2] to support GPU based spring-mass systems. A real-time system simulating and visualizing the deformable heart was implemented. Screenshots are seen in Figure 6. In a setup where we do six iterations before visualizing, the simulation runs at 192 iterations per second and 32 frames visualized per second. The simulation step alone could be run at 219 iterations per second.

4. Discussion and Conclusion

We successfully expressed the spring-mass model in terms of the GPU. This is seen from Table 1 as a noticeable speedup compared to a similar CPU implementation. This speedup can be used to model and simulate morphology with a larger number of primitives than previously seen as well as faster numerical calculations. An added geometrical detail is expected to more accurately simulate complex organs such as the heart.

In the past decade, GPU performance growth has exceeded that of the CPU [9]. This

Table 1. Iterations per second (excluding visualization) with the CPU spring-mass and GPU spring-mass implementations.

Method / Nodes	CPU	GPU	GPU / CPU speedup
10.000	45,8	839,8	18,7
20.000	20,2	476,9	23,6
40.000	9,9	264,6	26,9
50.000	7,8	218,0	28,1
100.000	3,3	104,1	31,4

is expected to extrapolate well into the future. Hence the acceleration factor is expected to grow correspondingly.

With a CPU based simulation it is a potential bottleneck to visualize particle-positions since this requires a transfer of vertex attributes each frame. Visualization of the GPU based solution has the advantage that the surface-mesh is cached in video memory since there are no CPU based changes.

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.

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.

5. Future Work

The current visualization is based on a mapping from one surface particle of the simulation to one surface vertex of the visualization. A more flexible mapping would enable us to decouple the details of visualization and simulation. This would enable a smoother appearance of the proposed spring mass implementation.

Future work should also include studies on how the GPU and CPU can work more closely together. In the presented solution, the CPU is not utilized efficiently because the transfer of data from the GPU to the CPU becomes a bottleneck. When this communication becomes faster (i.e. through PCI-express), we must consider what kind of processing is suitable for the GPU and CPU.

The clinical significance of the added morphological detail as well as clinical use of the Cardiac Surgery Simulator will be examined.

Acknowledgements

We acknowledge pediatric heart surgeons Vibeke Hjortdal and Ole Kromann Hansen for their clinical feedback. For the data acquisition we acknowledge the contributions of Dr. Gerald Greil and Dr. Axel Kuettner, 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] Jesper Mosegaard. *LR-spring-mass model for cardiac surgical simulation*. Medicine Meets Virtual Reality 12, pp 256-258, 2003.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] Buck I, Purcell T. A toolkit for computation on GPUs. GPU Gems. Addison Wesley 2004.