An Introduction to GPU Accelerated Surgical Simulation

Thomas Sangild Sørensen[1] and Jesper Mosegaard[2]

[1] Centre for Advanced Visualisation and Interaction, [2] Department of Computer Science
University of Aarhus, Denmark
`sangild@cavi.dk, mosegard@daimi.au.dk`

Abstract. Modern graphics processing units (GPUs) have recently become fully programmable. Thus a powerful and cost-efficient new computational platform for surgical simulations has emerged. A broad selection of publications has shown that scientific computations obtain a significant speedup if ported from the CPU to the GPU. To take advantage of the GPU however, one must understand the limitations inherent in its design and devise algorithms accordingly. We have observed that many researchers with experience in surgical simulation find this a significant hurdle to overcome. To facilitate the transition from CPU- to GPU-based simulations, we review the most important concepts and data structures required to realise two popular deformable models on the GPU: the finite element model and the spring-mass model.

1   Introduction

General-purpose computation using graphics hardware (or GPGPU) is a research area that has grown rapidly in recent years. By using the modern graphics card (i.e. the GPU) for computations, many computationally heavy algorithms have been accelerated significantly compared to conventional CPU-based algorithms. This includes most of the techniques currently being applied in surgical simulators. Unfortunately, the GPU is difficult to utilise efficiently. A substantial knowledge of its design, programming model, and limitations is necessary for optimal results. This paper is intended as an introductory article to GPGPU aimed specifically for researchers with experience in surgical simulation, who wish to attempt a GPU implementation of their algorithms. We review the literature introducing the most important concepts, and discuss the hardware limitations we must adhere for optimal results.

An overview of the many applications of GPGPU is best obtained by exploring the online resource [1] and the books in the GPU Gems series [2,3]. Moreover these surveys [4,5] and course material [6,7] highlight some commonly used algorithms. The survey by Strzodka et al [5] has a well-written introduction to scientific computation on the GPU. Several programming languages are available, ranging from a low level machine language [8] to high level C-like languages such as Cg [9] and GLSL [10]. Based on a general data abstraction model for parallel programming, streams, a compiler and run-time system is available [11]. A few getting started tutorials are available here [12]. This paper extends these references with a survey and

discussion of GPU accelerated techniques aimed specifically towards surgical simulation.

2   GPGPU Concepts and Performance

The standard graphics pipeline in OpenGL and DirectX contains fixed functionality vertex and pixel shaders. A basic knowledge of this pipeline is assumed in the remaining paper, and only a brief review is provided below. More information can be found in e.g. [13]. The vertex shader transforms the geometry (triangles) received from the application from local object space coordinates to window coordinates through a series of transformations. The colour and texture coordinates are also computed for each vertex. The geometric primitive is subsequently *rasterized*, a term that describes the process in which the colour of each pixel in the primitive is computed. The pixel shader is responsible for computing these colours. Based on each pixel's spatial position in the geometric primitive, the pixel shader receives the per-pixel interpolated colour and texture coordinates as input. The fixed function pixel shader then computes the output colour as a function of the input colour and the texture colours. The texture colours are found from texture lookups using the per-pixel input texture coordinates.

In the past generations of GPUs the vertex and pixel shaders have gradually become fully programmable. In GPGPU we utilise this to write pixel shaders that no longer compute colours, but instead the scalars and/or vectors involved in a general computation. Each pixel can store a 4-tuple of floating point values in up to 32 bit precision per entry. We store the computed pixels directly in a non-visible GPU memory buffer as it is no longer meaningful to visualise these pixels directly. This buffer is actually a texture that can be used as input for subsequent iterations of our computations. Hence we have established a computational model in which we can both read from and write to GPU memory. A custom vertex- and a custom pixel shader program are uploaded to the GPU and applied in the subsequent processing of primitives in parallel. Due to this parallel nature of the GPU (the Radeon X1900XT has 48 pixel pipes, a Geforce 7900GTX has 24 pixel pipes), a high throughput can be obtained. A CPU-based physical simulation typically stores data in one-, two-, and three-dimensional arrays. On the GPU, data is stored instead in one-, two-, or three-dimensional textures. As the GPU works most efficiently on two-dimensional data structures, we transform both 1D and 3D textures to 2D textures in practise [14]. Naturally, some bookkeeping is necessary to handle this transformation. When a computation is invoked, the pixel shader receives an input texture coordinate that identifies the spatial position of the corresponding pixel in the input textures. If the algorithm requires access to neighbouring pixels, this is achieved by offsetting the input texture coordinate before looking up in the respective textures. These offsets can be either "global constants" or obtained through an additional texture lookup at the current pixel. As will be explained in section 3, the type of offset depends on the underlying spatial discretization of the computational domain; whether it is structured or unstructured.
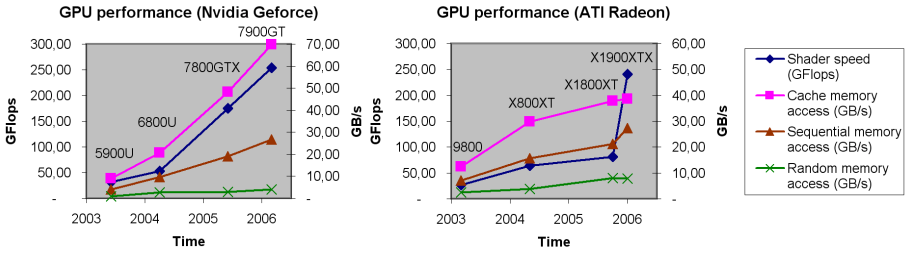
Fig. 1. *Observed* performance from the most recent generations of Nvidia and ATI GPUs. Data was obtained using GPUBench [15]. Blue diamonds represent the shader performance measured in GFlops. Cache, sequential, and random memory access measured in GB/s are depicted in the remaining graphs.

Using the benchmark test suite GPUBench [15] an overview of the performance of a system's GPU can be obtained. **Fig. 1** shows the graphs for the most recent GPUs from Nvidia and ATI. Looking at the number of floating point operations available per second (Flops) it can be observed that the current performance leaders provide roughly 250 GFlops. This number was obtained from the GPUBench test `instrissue`. This test measures the number of MAD instructions that can be executed per second on the present GPU. Since each shader operates on 4-tuples of floating point values and each MAD operation constitutes two floating point operations (a multiplication and an addition), the values reported by `instrissue` are multiplied by 8 for conversion to GFlops. Compared to the *theoretical* peak performance from a state-of-the art CPU (7.4 GFlops / 3.8 GHz Intel Xeon [16]) it should be clear why a GPU implementation of a surgical simulation can potentially boost performance. Discussing potential performance gains merely based on the shader speed reported in **Fig. 1** does not provide a fulfilling picture however, as a typical GPU implementation of a surgical simulation would not be *compute bound*. More likely it would be *memory bound* - meaning that access to GPU memory would be the limiting factor. Consequently, **Fig. 1** contains three graphs showing the observed memory bandwidth on the most recent GPUs. They are based on data obtained running GPUBench's `floatbandwidth` test on the respective GPUs [15]. It can be seen that cache memory access is significantly faster than sequential and
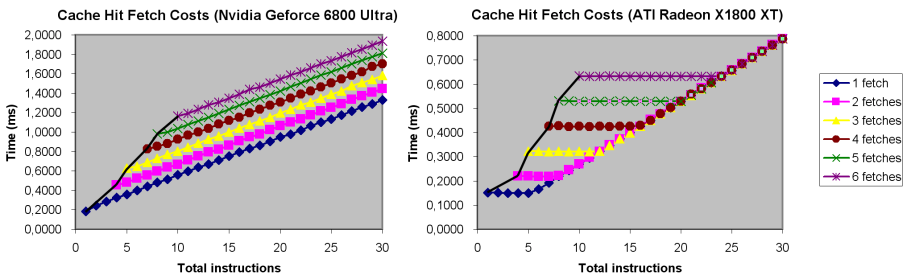

Fig. 2. Cache hit memory access costs as a function of the number of shader instructions. Data was obtained from the most recent generations of Nvidia (left) and ATI (right) GPUs using GPUBench [15]. The number of texture fetches in each test was varied from 1 to 6.

random memory access. The cache memory bandwidth constitutes an upper limit in memory bandwidth, hardly attainable in real-world applications. Depending on the memory coherence of a given application, the growth in memory bandwidth could instead follow the lines depicting the sequential or random memory access bandwidth. Thus, designing memory coherent algorithms is of outmost importance.

To discuss whether a given application is compute or memory bound, the literature (e.g. [3]) defines the *arithmetic intensity* of an application as the "amount of work" that is performed per memory access. Applications with high arithmetic intensity are most likely compute bound while low arithmetic intensity is an indication of a memory bound algorithm. To discuss this issue in more detail, we once again resort to GPUBench: The test `fetchcosts` shows the execution time of a GPU program as a function of the number of instructions. **Fig. 2** shows the results from this test on two GPUs. Note that each test is comprised of six sub-tests that perform one to six memory cache accesses each. We will discuss below the results obtained on an ATI Radeon GPU. A similar (but not entirely identical) discussion can be made for the Nvidia based GPU, but we leave this discussion for the reader to complete. First notice the horizontal line segments in the rightmost half of **Fig. 2**. They show that for each memory access, a number of "free" computations can be made without influencing the overall execution time. Only as the non-horizontal (diagonal) part of the graph is reached, there is a cost associated to issuing additional instructions. From the figure we can predict the execution time of an application consisting entirely of memory reads (solid black line). Notice that the slope of this line is much steeper than the slope of the diagonal. The diagonal constitutes the border between a memory bound and a truly compute bound application: An application with an arithmetic intensity that places it between the leftmost solid line and the diagonal is memory bound, while an application with an arithmetic intensity that places it on (or close to) the diagonal would be compute bound. As we shall see in the subsequent sections, surgical simulation algorithms implemented on the GPU are most likely memory bound, as the complexity in algebraic operations per memory access is limited. Experiments show however, that these GPU-based algorithms still significantly outperform their CPU-based counterparts.

3   Surgical Simulation on the GPU

Many computational models for deformable surfaces have been proposed in the existing literature. We refer to the surveys [17,18] for a detailed overview. We limit our description of GPU-based techniques to mesh-based deformable models (most often a mesh of triangles, tetrahedrons, or cubes) as this is the preferred approach in real-time surgical simulators that must handle arbitrary incisions and general changes in topology. For the remaining paper we refer to *nodes* as the discretized points defining a mesh. We present an overview of the required GPU-based techniques to implement the most common deformable models: finite element models and spring-mass models. The reader should subsequently be able to define custom modifications to these general models in GPU terms. The implicit linear elastic finite element

models presented by Bro-Nielsen et al in [19] are discussed in section 3.1. The explicit finite element model (tensor-mass model) presented by Cotin et al in [20] and the explicit spring-mass model (e.g. [20,21]) are discussed in section 3.2. Szekely et al used a cluster of processors in [22] to realise a laparoscopic surgery simulator. Many of the general considerations on the design of parallel algorithms for numerical computations on multiple CPUs transfer directly to the parallel processor we introduce in this paper, namely the GPU.

3.1   Implicit Finite Element Models

Using the notation from [19], finding the deformations in the implicit linear elastic finite element model reduces to solving either a static system on the form $\mathbf{Ku} = \mathbf{f}$ or a dynamic system on the form $\mathbf{M\ddot{u}} + \mathbf{C\dot{u}} + \mathbf{Ku} = \mathbf{f}$, where $\mathbf{M}$ and $\mathbf{C}$ are diagonal mass and damping matrices, $\mathbf{K}$ is a symmetric positive definite matrix representing the topology and stiffness of the discretized mass points, and $\mathbf{u}$ and $\mathbf{f}$ are the deformation and external force vectors respectively. No matter the choice of system, it can be rewritten on the form $\mathbf{\tilde{K}\tilde{u}} = \mathbf{\tilde{f}}$ following a finite difference time discretization for the dynamic system [19]. If we let n denote the total number of mass point in the finite element discretization, $\mathbf{\tilde{K}}$ has dimensions $3n$ x $3n$ and entry k_{ij} encodes the connectivity and stiffness between nodes i and j. In its most elementary form $\mathbf{\tilde{k}}$ is sparse having non-zero entries only between connected mass points.

Fig. 3. Structured (left) and unstructured (right) tetrahedral meshes. The choice of tetrahedralisation influences the layout of the resulting stiffness matrix in a linear elastic finite element model.

Depending on the choice of spatial discretization, it can either be structured (banded) or unstructured. **Fig. 3** (left) illustrates a spatial discretization that leads to a banded matrix. Boxes (possibly consisting of six tetrahedra of fixed topology) are used as the basic spatial building blocks in a regular three-dimensional grid. The rightmost tetrahedralisation in **Fig. 3** on the other hand, leads to an unstructured sparse matrix. Finally, using the condensation technique described in [19], $\mathbf{\tilde{k}}$ can be transformed to a smaller *dense* matrix of boundary nodes. We have distinguished between the different layouts of $\mathbf{\tilde{k}}$ since they each call for their own distinct representation on the GPU. The three matrix layouts are 1) sparse (banded), 2) sparse (unstructured), and 3) dense. Solving the linear system of equations involves matrix and vector algebra. We discuss common linear algebra operations below for the different representation of $\mathbf{\tilde{K}}$. This is followed by a discussion on GPU-based solutions to the linear system.

The first formulation of a general framework for numerical algebra on the GPU was published by Thompson et al in [23] in a very "machine-near" language. Inspired by the BLAS and LAPACK libraries [24,25], Krueger et al subsequently published their initial work on a GPU-based counterpart [3,26].
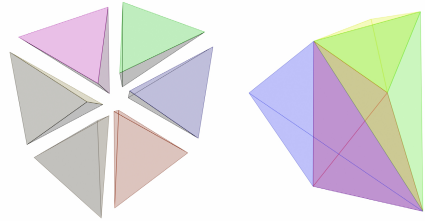
Fig. 4 (top) shows their representation of *sparse banded matrices*. Each band in an *n* x *n*–dimensional matrix **A** can be seen as a one-dimensional vector of length *n*. As explained in section 2, we convert this vector to a two-dimensional texture on the GPU. Similarly, vectors **b** and **x** of dimension *n* are stored in textures of identical dimensions to the bands of **A**. In many computations it is necessary to find the matrix-vector multiplication **x=Ab.** To achieve this we render a quad covering output texture **x** in multiple passes – one rendering pass for each band in **A**. **Fig. 4** (bottom) illustrates the three passes required for a tri-banded matrix-vector multiplication with this representation. In each pass the values in corresponding pixels in textures **A** and **b** are multiplied and
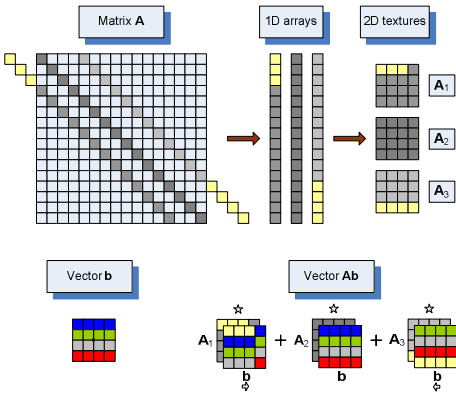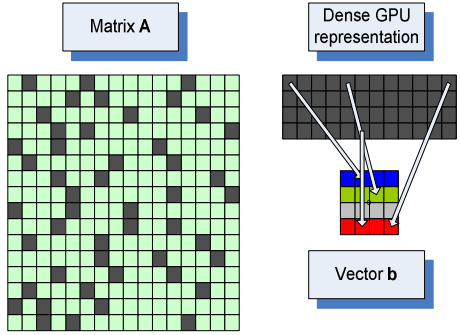

Fig. 5. Unstructured sparse matrix representation specialized from [27]. We assume exactly three non-zero entries per row in this example (grey). These values are stored in a "dense texture". For each pixel in this texture a pointer (i.e. texture coordinate) to the corresponding entry in vector **b** is stored (arrows).

added to **x**. In each pass (except the pass using the matrix diagonal) the texture coordinates used to look up pixels in **b** are shifted to account for the corresponding band position. Several optimisations to this basic scheme is possible and discussed in [3,26]. Since it is not possible to read from and write to the same texture during a pass, the accumulative writes to texture **x** must be implemented through two textures, one of which is bound as input and the other for writing in an alternating fashion. It is important to realise the parallel nature of the algorithm: For each of the three passes in the example in **Fig. 4**, the *n* entries in the result vector **x** are computed simultaneously, providing a significant speedup to CPU-based matrix-vector multiplications given a sufficient number of pixel pipes and texture memory bandwidth.

Sparse unstructured matrices are handled differently from the banded matrices above. Krueger et al [3,26] renders point based primitives to

Fig. 4. GPU representation of a sparse banded matrix **A**, a vector **b**, and the corresponding matrix-vector multiplication (adapted from [26]). Top: Each band represents a one-dimensional array which is stored in a 2D texture on the GPU (**A**$_1$-**A**$_3$). Zeroes are prepended or appended depending on the position of the band in the matrix. Bottom: A vector **b** is defined and multiplied to **A**. Each band in **A** is multiplied with **b** pixelwise. The products are added to form **Ab**. Notice that the texture coordinates used to access **b** are offset corresponding to each band.

implement such matrix-vector multiplications. We will instead describe a different approach using texture pointers however, as this relates nicely to this section's subsequent discussion of algorithm design that minimises memory bandwidth. We illustrate a specialisation of the general sparse unstructured matrix-vector multiplication by Bolz et al [27] to find the product $\mathbf{x}=\mathbf{Ab}$. We assume that a constant number of non-zero entries exist in each row of the sparse n x n dimensional matrix $\mathbf{A}$ of. In **Fig. 5** we use only three entries per row to reduce the size of the figure. We create a one-dimensional array of length $2 \cdot 3 \cdot n$ to represent $\mathbf{A}$. $3n$ entries are necessary to store the non-zeroes values in $\mathbf{A}$. Furthermore, for each value we additionally store a pointer to the corresponding entry in textures $\mathbf{b}$. As always, we represent this one-dimensional array as a two-dimensional texture on the GPU. We render a quad covering our output texture $\mathbf{x}$ to initiate the parallel computation of $\mathbf{Ab}$. For each pixel we look up the three non-zero values in the corresponding row in the texture representation of $\mathbf{A}$ from the input texture coordinate. The texture representation of $\mathbf{A}$ furthermore provides us with three pointers (texture coordinates) that are used to look up the values in $\mathbf{b}$ corresponding to the non-zero entries in $\mathbf{A}$. The results from the three multiplications are added and stored in $\mathbf{x}$. Again, it is important that the reader recognizes the parallel nature of the algorithm, in which each entry in $\mathbf{x}$ is computedin parallel.

With the understanding of sparse matrix representations on the GPU, the reader should have the prerequisites to derive representations of ***dense matrices*** as these are more straightforward than those of sparse matrices. We refer to [3,28,29] for completeness.

We now return to solving the linear system $\tilde{\mathbf{K}}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$ that was defined initially in this section. As $\tilde{\mathbf{K}}$ is symmetric and positive definite, one approach to finding $\tilde{\mathbf{u}}$ is through the conjugate gradient algorithm. Using their respective frameworks for linear algebra, both Krueger et al and Bolz et al showed how to implement the conjugate gradient algorithm on the GPU in [26,27]. An alternative approach guaranteed to converge to the right solution for arbitrary starting configurations is the Gauss-Seidel iterative process (again since $\tilde{\mathbf{K}}$ is symmetric and positive definite). Contained in [26] is a short section discussing the implementation of this algorithm on the GPU. Closely related to Gauss-Seidel's method is the Jacobi method. In contrast to Gauss-Seidel's methods, Jacobi's method is ideally suited for a parallel implementation. For this reason it has been used intensely in previous GPGPU publications, e.g. in several chapters in [2,3], and in [30,31].

The representation of banded matrices as shown in **Fig. 4** results in a minimum number of texture fetches: only the actual values needed for a matrix-vector multiplication are read from texture memory. The unstructured sparse matrix representation on the other hand requires further texture lookups to perform a matrix-vector multiplication, as pointers are stored in textures and the corresponding values only obtained through an additional texture lookup. Looking back at the discussion related to **Fig. 2** it should be clear why the banded representation performs better than the unstructured alternative. It is simply due to the lower number of texture fetches involved. Consider also the limited number of algebraic operations performed per memory access in both approaches (that is the arithmetic intensity is low). Given the memory bandwidth on current GPUs both matrix representations are memory bound, although positioned differently in the graphs in **Fig. 2**. It was Fatahalian et al who

initially reported that memory access is indeed the limiting factor for dense matrix-matrix multiplications on the GPU [28].

The reader is encouraged to examine the GPU-based surgical simulator by Wu et al [32]. They used an implicit finite element solver through the conjugate gradient algorithm and obtained a two-fold acceleration. This work was done on an Nvidia Geforce 5950 Ultra however. As is clear from **Fig. 1**, both the GPU speed and memory bandwidth have increased five- to ten-fold since on the most recent GPUs.

3.2 Explicit Models: Spring-Mass and Tensor-Mass Models

We return to the general equation of Newtonian motion: $\mathbf{M\ddot{u}} + \mathbf{C\dot{u}} + \mathbf{Ku} = \mathbf{f}$ [19,20]. We now seek to solve the system of differential equations through an explicit time integration scheme rather than by the implicit method discussed in section 3.1. A particularly well suited explicit integration scheme is Verlet integration [33], a scheme in which the position of each mesh node for the subsequent time step is calculated from its positions in the two previous iterations and from an elastic force vector (acceleration vector). No additional information, e.g. velocities, needs to be stored and calculated. The force vector is calculated locally from each node's connectivity in the

Fig. 6. Position texture, inspired from [35]. A regular 3D grid of nodes is mapped to a 2D texture. The colors of the individual pixels denote the corresponding particle's position.

mesh. We denote the force vector corresponding to a spring-mass system as $\hat{\mathbf{f}}_i$ and the force vector relating to the tensor-mass system as $\tilde{\mathbf{f}}_i$. Using the notation from [20] these forces are then defined as
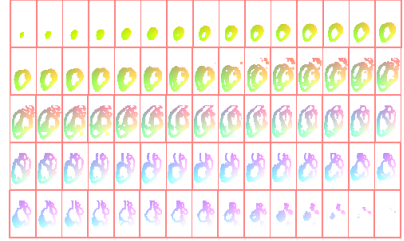
$$\hat{\mathbf{F}}_i = \sum_{j \in N(\mathbf{P}_i)} k_{ij} \left(\left\| \mathbf{P}_i \mathbf{P}_j \right\| - l_{ij}^0 \right) \frac{\mathbf{P}_i \mathbf{P}_j}{\left\| \mathbf{P}_i \mathbf{P}_j \right\|} \quad \text{and} \quad \tilde{\mathbf{F}}_i = \mathbf{K}_{ii} \mathbf{P}_i^0 \mathbf{P}^i + \sum_{j \in N(\mathbf{P}_i)} \mathbf{K}_{ij} \mathbf{P}_j^0 \mathbf{P}^j$$

where $\mathbf{P}_i$ denotes the position of node i, $\mathbf{P}_i^0$ is the initial (undeformed) position of node i, $\mathbf{P}_i \mathbf{P}_j$ is the vector between nodes i and j, l_{ij} and k_{ij} is the spring rest length and stiffness respectively between nodes i and j, and $\mathbf{K}$ is the rigidity (or stiffness) matrix of the linear elastic finite element model.

A two-dimensional GPU- and spring-mass based cloth simulation using Verlet integration was presented in [34]. It is sufficiently simple to be recommended for inexperienced GPU programmers. Mosegaard et al presented a three-dimensional, volumetric spring-mass based surgical simulator implemented on the GPU in [35]. Their paper compares two spring-mass implementations, one in which nodes were confined to a regular three-dimensional grid, and one in which node positions were unrestricted and springs explicitly represented in a *connectivity texture*. Overall, a twenty-fold acceleration over a similar CPU-based system was achieved for the first method, while the latter achieved a ten-fold acceleration. Their results were obtained on a Geforce 6800 Ultra. It is clear from **Fig. 1** that both the shader speed and memory bandwidth have increased significantly since and even better results could be

obtained on the most recent generations of GPUs. In the faster of the two methods they use a *position texture* to store the positions of the nodes in the spring-mass system. An example one such texture is depicted in **Fig. 6**. To initialise parallel computation of each time-step, a quad at the size of this texture is rendered in the output buffer. A depth test is used to prevent that calculations are wasted on the white (void) particles. The forces $\hat{\mathbf{F}}_i$ are computed for each pixel (node): The two most recent position textures are provided as input textures to the pixel shader. By adding to the input texture coordinate the fixed offset to each neighbour, each neighbouring node's position can be looked up. As the nodes are restricted to a regular grid, the individual spring rest lengths are known and need not be looked up. Furthermore, the spring stiffness is also kept constant. Thus, the only texture fetches involved are those used to obtain the connected nodes' positions. I.e. they use only the minimal number of texture fetches. This is important in the light of the discussion concerning the cost of texture fetches (**Fig. 2**). Their alternative approach uses texture lookups to fetch the texture coordinate of each neighbour. This doubles the overall number of texture fetches, consequently reducing the simulation rate by a factor of two. Replacing the spring induced forces $\hat{\mathbf{F}}_i$ with the tensor-mass forces $\tilde{\mathbf{F}}_i$ would instead solve an explicitly formulated finite element model. For each connected node we need then an additional texture lookup in the stiffness matrix to obtain $\mathbf{K}_{ij}$. Consequently we can expect the tensor-mass model to run at half the speed of the spring-mass model. Compared to a CPU-based implementation however, it is still significantly faster. The most recent performance measurements are found in Sørensen et al [36], who report simulation rates exceeding 1 kHz using a Geforce 7800 GTX on a spring-mass system consisting of 20.000 nodes connected with 18 neighbours each in a regular volumetric mesh (grid).

The methods by Mosegaard et al are simple to implement and run fast but use a significant amount of texture memory: The position texture approach wastes memory representing void particles, while the connectivity texture approach allocates memory for a constant number of neighbours per node wasting texture memory if the number of neighbours varies significantly throughout the mesh. To conserve memory, Georgii et al used a stack of *valence textures* to encode different levels of connectivity in [37]. Unfortunately the algorithm reduces performance as well due to a much more complicated rendering scheme. This led Georgii et al to develop an edge-centric data structure instead that "iterates" over springs rather than nodes [38]. This reduces the arithmetic intensity of their algorithm since each spring force is now computed only once whereas they are computed twice in the previous methods [35,37]. As all the presented algorithms are most likely memory bound (**Fig. 2**) this is not a major advantage however. We are more interested in examining the number of texture fetches used in the edge-centric approach. On the regular mesh shown in **Fig. 6** we experienced that the number of texture fetches involved in the edge-centric approach versus the position texture approach are almost identical. The edge-centric approach currently runs only in 8 bit or 16 bit precision as a necessary blending operation is not supported in 32 bit precision on any GPUs yet. Also, the vertex processor is used intensely, a potential bottleneck. It will be up to the individual application to weigh the advantage of reduced memory consumption versus these precision and vertex processing issues.

3.3   Visualisation and Interaction

Depending on the chosen simulation model, the result of each time step is either a texture of node deformations (implicit model) or a texture of node positions (explicit model). In either case a deformed surface triangle-mesh of the modelled organ can be visualised from a static display list of the initial mesh configuration through a dedicated vertex shader [35]. For each vertex in the visualised surface mesh, the application provides the required texture coordinate to look up the corresponding deformation vector or particle position. The vertex shader can thus compute the deformed vertex position for the current time step.

From both section 3.1 and 3.2 it is clear that a structured spatial discretization of the simulated volume results in the fastest algorithms due to a minimum number of texture lookups required in each time step. This can however result in a jagged (stair-like) look of the modelled morphology as illustrated in **Fig. 7**. To overcome this problem Mosegaard et al proposed in [39] to fully decouple surface visualisation from the underlying volumetric simulation. They represent each vertex on the surface model by an offset from the nearest node in the simulation mesh. **Fig. 7** shows a smooth surface drawn through this method. The offset vectors are expressed in the tangent-space of the surface, and the surface thus correctly deforms based on the deformation of the associated nodes in the simulation mesh.

Interaction with a GPU-based surgical simulator is the final issue to be discussed in this paper. The overall question is whether to resolve user interaction on the CPU or on the GPU. If the CPU is chosen one must be careful not to transfer large amounts of data from the GPU to the CPU in each

Fig. 7. Surface visualisation (circle) de-coupled from a volumetric simulation of a sphere discretized to a regular grid (grey). The green circles represent vertices on the surface mesh that can be sampled at any resolution. Each vertex is represented by an offset vector (arrow) from the nearest simulation node.

frame, as this would introduce a performance bottleneck. Consequently, interaction that involves computations on the current state of the simulation is probably best implemented on the GPU. Peripheral devices can only be communicated with through the CPU however, so a minimum amount of per-frame data transfer cannot always be avoided. Sørensen et al showed in [36] how to implement force feedback from a GPU-based simulator with limited performance penalty. Several groups have recently published algorithms for GPU accelerated collision detection [40-42].
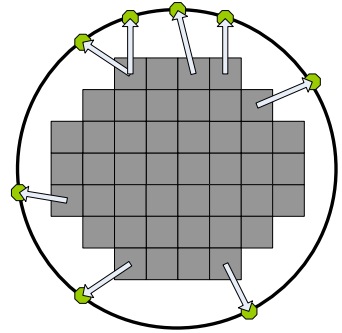
Acknowledgements

References

1. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.
2. Fernando, R. GPU Gems, Part VI. Addison-Wesley 2004.
3. Pharr, M. GPU Gems 2, Part IV-VI. Addison-Wesley 2005.
4. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., and Purcell, T.J. A Survey of General-Purpose Computation on Graphics Hardware. State of the Art Reports, Eurographics 2005:21-51.
5. Strzodka, R., Doggett, M., and Kolb, A. Scientific Computation for Simulations on Programmable Graphics Hardware. Simulation Modelling Practice and Theory 2005;13(8):667-681.
6. GPGPU course, Siggraph 05. http://www.gpgpu.org/s2005/.
7. IEEE Visualization 2005 tutorial. http://www.gpgpu.org/vis2005/.
8. OpenGL extensions specifications. ARB_vertex_program and ARB_fragment_program. http://oss.sgi.com/projects/ogl-sample/registry/index.html.
9. Fernando, R. and Kilgard, M.J. The Cg Tutorial. Addison-Wesley 2003.
10. Rost, R.J. OpenGL Shading Language. Addison-Wesley 2004.
11. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., and Hanrahan, P. Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transactions on Graphics, Siggraph 2003;23(3):777-786.
12. Göddeke, D. GPGPU::Tutorial. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html.
13. Möller, T.A. and Haines, E. Real-Time Rendering. A K Peters 2002.
14. Harris, M.J., Baxter III, V., Scheuermann, T., and Lastra, E. Simulation of Cloud Dynamics on Graphics Hardware. Siggraph/Eurographics Workshop On Graphics Hardware 2003:92-101.
15. Buck, I., Fatahalian, K., and Hanrahan, P. GPUBench: Evaluating GPU performance for Numerical and Scientific Application. General Purpose Computing on Graphics Processors, ACM Workshop 2004:C-20.
16. Dongarra, J. The Linpack Benchmark Report. http://www.netlib.org/benchmark/performance.ps.
17. Montagnat, J., Delingette, H., and Ayache, N. A review of deformable surfaces: topology, geometry and deformation. Image and Vision Computing 2001;19(14):1023-1040.
18. Gibson, S.F.F. and Mirtich, B. A Survey of Derformable Modeling in Computer Graphics. Technical Report, Mitsubishi Electric Research Lab.
19. Bro-Nielsen, M. and Cotin, S. Real-time Volumetric Deformable Models for Surgeyr Simulation using Finite Elements and Condensation. Computer Grahics Forum, Eurographics 1996;15:57-66.
20. Cotin, S., Delingette, H., and Ayache, N. A Hybrid Elastic Model Allowing Real-Time Cutting, Deformations and Force-Feedback for surgery Training and Simulation. The Visual Computer 2000;16(8):437-452.
21. Liu, A., Tendick, F., Cleary, K., and Kaufmann, C. A Survey of Surgical Simulation: Applications, Technology, and Education. Presence 2003;12(6):599-614.
22. Szekely, G., Brechbuhler, C., Hutter, R., Rhomberg, A., and Schmid, P. Modelling of Soft Tissue Deformation for Laparoscopic Surgery Simulation. Medical Image Analysis 2000;4:57-66.
23. Thompson, C., Hahn, C., and Oskin, M. Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis, 35th IEEE/ACM International Symposium on Micro Architecture 2002:306-3.

24. Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. An extended set of FORTRAN basic linear algebra subprograms. ACM Transactions on Mathematical Software 1988;14:1-17.

25. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. LAPACK Users' Guide. Society for Industrial and Applied Mathematics 1999.

26. Krueger, J. and Westermann, R. Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics, Siggraph 2003;22(3):908-916.

27. Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Transactions on Graphics, Siggraph 2003;22(3):917-924.

28. Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. ACM Siggraph/Eurographics Conference on Graphics Hardware 2004:133-137.

29. Galoppo, N., Govindaraju, N.K., Henson, M., and Manocha, D. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. ACM/IEEE conference on Supercomputing 2005:3.

30. Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. ACM Siggraph/Eurographics Conference on Graphics Hardware 2003:102-111.

31. Rumpf, M. and Strzodka, R. Using Graphics Cards for Quantized FEM Computations. IASTED Visualization, Imaging and Image Processing 2001:193-202.

32. Wu, W. and Heng, P.A. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting. Computer Animation and Virtual Worlds 2004;15:219-227.

33. Dummer, J. A Simple Time-Corrected Verlet Integration Method. http://www.gamedev.net/reference/programming/features/verlet/.

34. Green, S. Cloth Simulation on the GPU. http://developer.nvidia.com/object/demo_cloth_simulation.html.

35. Mosegaard, J. and Sørensen, T.S. GPU accelerated surgical simulators for Complex Morphology. IEEE Virtual Reality 2005:147-153.

36. Sørensen, T.S. and Mosegaard, J. Haptic Feedback for the GPU-based Surgical Simulator. Studies in Health Technology and Informatics, 14th Medicine Meets Virtual Reality 2006;119:523-528.

37. Georgii, J., Echtler, F., and Westermann, R. Interactive Simulation of Deformable Bodies on GPUs. Simulation and Visualisation 2005:247-258.

38. Georgii, J. and Westermann, R. Mass-Spring Systems on the GPU. Simulation Modelling Practice and Theory 2005;13(8):693-702.

39. Mosegaard, J. and Sørensen, T.S. Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU. Immersive Projection Technology & Eurographics Virtual Environments Workshop 2005:105-110.

40. Govindaraju, N.K., Lin, M., and Manocha, D. Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling using Graphics Hardware. IEEE Virtual Reality 2005:59-66.

41. Wong, W.S.-K. and Baciu, G. GPU-based intrinsic collision detection for deformable surfaces. Computer Animation and Virtual Worlds 2005;16:153-161.

42. Choi, Y.-J., Kim, Y.J., and Kim, M.-H. Rapid pairwise intersection tests using programmable GPUs. The Visual Computer 2006;22:80-89