IPT & EGVE Workshop (2005)
R. Blach, E. Kjems (Editors)

Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU

J. Mosegaard [1] & T. S. Sørensen [2]

[1] Department of Computer Science, [2] Centre for Advanced Visualization and Interaction, University of Aarhus, Denmark

Abstract

Modern graphics processing units (GPUs) can be effectively used to solve physical systems. To use the GPU optimally, the discretization of the physical system is often restricted to a regular grid. When grid values represent spatial positions, a direct visualization can result in a jagged appearance. In this paper we propose to decouple computation and visualization of such systems. We define mappings that enable the deformation of a high-resolution surface based on a physical simulation on a lower resolution uniform grid. More specifically we investigate new approaches for the visualization of a GPU based spring-mass simulation.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Physically based modeling I.3.7 [Computer Graphics]: Animation

1. Introduction

Recently GPUs have become programmable to a degree that makes them useful for general-purpose computation [BP04]. Specifically, the computation of physically based systems have been successfully implemented [HCSL02, MHS05, MS05, BFGS03]. The GPU is designed to work with textures, and the discretization of the physical equations often takes advantage of this by restricting nodes to a regular grid. Consequently, attention must be put into visualizing the results in detail, smoothly and continuously.

In this paper we focus on the problem of visualizing a deformable surface defined by a spring-mass system solved on the GPU. In [MHS05, MS05] we presented methods for GPU accelerated computation of spring mass based elastic deformation. In particular, a grid-based arrangement of mass-points gave a speedup of a factor of 20 to 30 compared to a standard CPU implementation. To visualize the surface, we previously used a direct mapping from the position of visualized vertices to the associated position of mass-points in the grid. This resulted in a very jagged look, see figure 2 a) and b). This limitation is removed by the work in this paper, see figure 1 and figure 2 c).

We present three generally applicable methods for the visualization of a surface deformed by a set of nodal positions. The methods are not directly dependent on the calculation

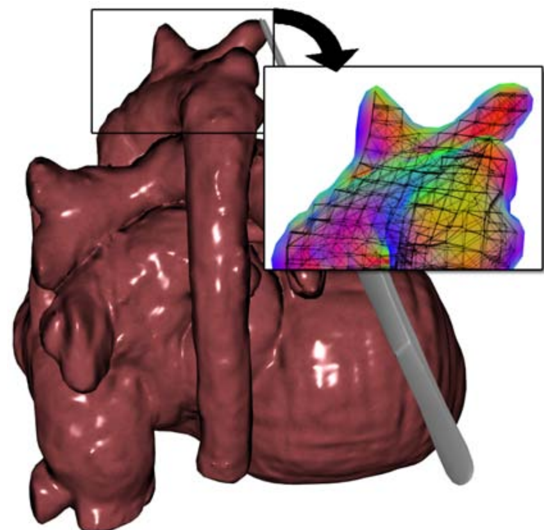
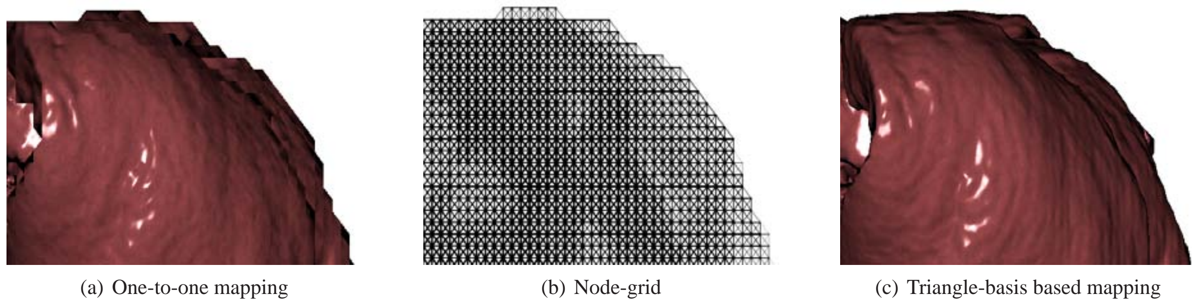


Figure 1: *A physical simulation on the GPU calculates the deformation of a grid of 20.000 particles representing the shape of a heart. A heart surface with 50.000 faces is mapped to the grid. As a result the highly detailed surface can be deformed in real-time based on the deformation of the simpler grid. In the close-up, the grid is shown on top of shading illustrating the dynamically calculated normals.*

© The Eurographics Association 2005.

delivered by
EUROGRAPHICS
DIGITAL LIBRARY
www.eg.org diglib.eg.org



(a) One-to-one mapping (b) Node-grid (c) Triangle-basis based mapping

Figure 2: *A close-up of the grid-based spring mass simulation showing the springs of the physics system in b). The visualization is done with one-to-one mapping between nodes and vertices in a) and offsets from nodes to vertices in c). An orthogonal projection matrix has been used to clearly show the grid.*

of the physical system and can be applied to other physical systems or methods of animation. First, a simple one-to-one mapping (with approximate normals) is defined between visualized vertices and mass-points. Secondly, we present two methods of deforming and visualizing a detailed surface based on the deformation of a relatively low number of nodes. The two methods both define a dynamically changing orthonormal vector basis for all nodes on the surface of the simulated volumetric grid. The surface-vertices to be visualized are expressed in this basis and consequently reflect the deformation of the simulated nodes. Tangent space bases can also be expressed in the dynamically calculated bases and enable normal-mapping of the highly detailed deforming surface geometry. Common to all the methods presented is that we construct a detailed surface mesh and render this mesh through a vertex program that defines the mapping between vertices on the surface and nodes in the simulation.

The overall goal of our project is to simulate surgery on children with congenital heart disease, supporting deformation as well as cutting of heart-tissue. The detailed visualization is needed for the surgeon to accurately recognize features of the heart. At the same time the physical system must have a resolution that allows for real-time deformation.

2. Previous work

Our simulated system can be regarded as a low-resolution geometry based on which we express and animate a highly-detailed geometry. Classical approaches to this problem on the CPU include bump mapping and displacement mapping. Bump mapping [Bli78] perturbs the surface normals at per texel basis affecting shading as if the geometry had bumps. Displacement mapping [Coo84] tessellates each triangle and displaces the newly generated vertices along the normal according to an amount read from a height-map. Displacement mapping allows for the animation of a low resolution mesh of control points that deform the high resolution geometry accordingly.

Bump mapping can easily be implemented in fragment programs on the GPU, but does not correct for the jagged appearance along the contour of the mesh. While bump mapping does not in itself solve the problem of the jagged contours, it can be combined with the presented work to visualize small-scale surface details. GPU implementations of displacement-mapping based on textures have been made possible through the texture lookup instruction in vertex programs introduced in shader model 3.0. Simple one-dimensional perturbations along the z axis of a grid of vertices have been used to visualize a dynamic heightmap [Kry05], but the method cannot directly do displacement mapping for an arbitrary 3D mesh.

Because of the large amount of fragment processing power available, several authors have introduced the idea of computing per-pixel offsets from actual geometry to a height-map represented geometry [HEGD04, KTI*01]. These techniques generally use a height-map with ray marching which is computationally expensive. Complex 3D meshes including curvature have furthermore been problematic because the current ray-marching techniques are not aware of curvature nor the mapping of heightmaps to faces. Techniques such a [WWT*03] pre-process the intersection-tests to simplify the calculations on the GPU and use lookups in the pre-processed data, at the cost of extensive memory usage. For our application the memory usage is too high. The technique furthermore requires the geometry of the simulated system to fully enclose the surface to be visualized. That requirement is too restrictive for a grid based simulation visualizing incisions that are smaller than the grid resolution.

3. Methods

In this paper we use the term *vertex* to describe a vertex in the surface mesh exclusively, and the terms *node* and *nodal* to describe a position in the lower detailed set of control points. When we wish to emphasize that a node-position is based on a physical simulation we use the term *particle*. We start out by briefly presenting our GPU spring-mass simulation in

© The Eurographics Association 2005.

section 3.1. In section 3.2 we present the simple one-to-one mapping between nodes and vertices and in section 3.3 we present two more general mappings from a set of vertices to a less detailed set of nodes.

3.1. Grid based calculation of deformation on the GPU

We briefly review the grid-based layout of particles (also called the method of implicit connections) for the calculation of deformation on the GPU as presented in [MHS05, MS05]. The method is based on a layout of particle-positions in a regular three-dimensional grid. Each particle is connected in a fixed pattern to the 18 nearest neighbours. The grid is mapped to a 2d texture for fragment processing, and the texture containing the current set of particle positions is called the position-texture. In fragment programs we calculate the forces influencing the particles and numerically integration to obtain positions. The basic linear spring force g_i with rest length l_{ij} for a node i with position p_i and neighbour positions D is calculated as:

$$\vec{g_i} = \sum_{j \in D} \frac{1}{2} k_{ij} \left(l_{ij} - \|\mathbf{p}_i - \mathbf{p}_j\| \right) \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} \tag{1}$$

This expression can be optimized for fragment processing if we assume that the spring coefficients k_{ij} are all equal to the constant k and observe that springs arranged in the grid have rest-lengths 1 and $\sqrt{2}$:

$$\vec{g_i} = \frac{1}{2} k \left\{ \sum_{j \in D_1} \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} - \sum_{j \in D_1} (\mathbf{p}_i - \mathbf{p}_j) + \sqrt{2} \sum_{j \in D_2} \frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} - \sum_{j \in D_2} (\mathbf{p}_i - \mathbf{p}_j) \right\} \tag{2}$$

D_1 is the set of neighbours with distance 1 and D_2 is the set of neighbours with distance $\sqrt{2}$. Besides saving instructions, this expression includes a sum of unit-vectors, which is used to find approximate normals after deformation.

To visualize the set of nodes we cannot simply read back the node positions to the CPU and render vertex positions accordingly, as this would be a major performance bottleneck. Instead we utilize a new feature in the shader model 3.0 design; texture lookups in vertex programs. We use vertex programs to express the calculations involved in mapping vertices to nodes. Through vertex texture fetches we transfer only nodes 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, allowing caching on the graphics card.

3.2. One-to-one mapping and approximating normals

The limited visualization method in [MHS05, MS05] is presented in this subsection to motivate the later generalization. The method is based on a simple one-to-one mapping between the position of a vertex and a surface node, simply transferring the position of the node to the corresponding vertex-position. To render the model we initially set up a display-list, which renders the surface geometry at its original rest position. We pass one texture coordinate per vertex to provide the coordinates of the corresponding node in the position texture. The vertex program then fetches the most recent node position, performs basic transformation and outputs the deformed position for further processing in the graphics pipeline.

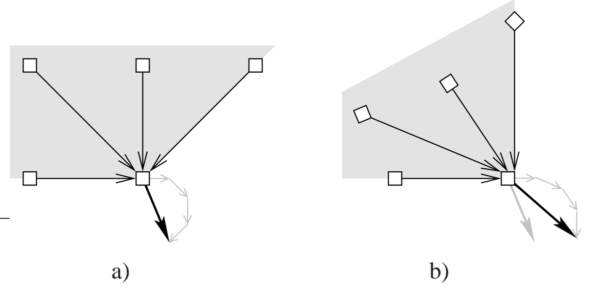


Figure 3: *A 2d example of a normal calculated on the basis of unit-vectors from neighbours. a) the original geometry. b) the deformed geometry. The normalized vectors (grey) are added to form the approximate normal (black).*

An important issue in this visualization is the calculation of surface normals used for shading of 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 the normal of the plane. In the presented GPU approach to a spring-mass system, we do not have the necessary surface information to simply reuse the CPU approach. 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. We approximate surface normals by the normalized sum of the unit vectors pointing from neighbours to the particle in question, an example is seen in figure 3:

$$\vec{N_a}(i) = normalize\left(\sum_{j \in D} normalize(\mathbf{p}_i - \mathbf{p}_j) \right) \tag{3}$$

Since we already computed the sum of the unit vectors in the force calculations in equation (2), 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 representing the particle-position. The normal is read and unpacked by the visualization vertex program and send to a fragment program for per pixel lightning.

3.3. Mapping using the triangle basis

In the previous section we presented a one-to-one mapping between a surface vertex and a node with a direct transfer of

© The Eurographics Association 2005.

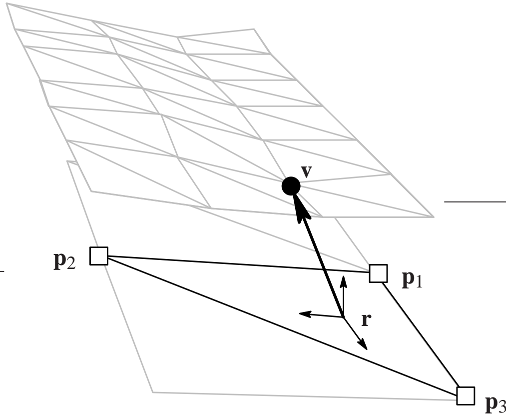


Figure 4: *a) For a high-resolution set of vertices and a low-resolution set of nodes we define for each triangle* $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ *and corresponding vertex* $\mathbf{v}$ *the reference point* $\mathbf{r}$ *and offset vector* $\vec{o} = \mathbf{v} - \mathbf{r}$.

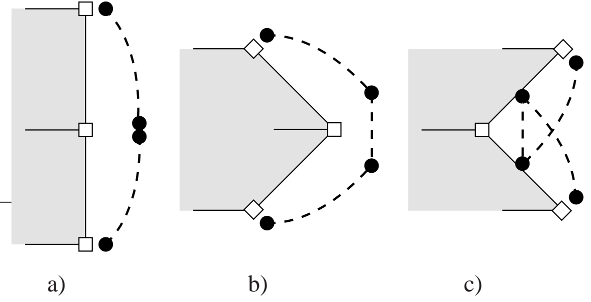


Figure 5: *Visual artifacts can occur when vertices are rotated based on per triangle information only. For simplicity we exemplify in 2D. a) is the original configuration of nodes (boxes) and vertices (dotted line and spheres). b) illustrates the problem of two vertices that are very close in the original configuration but are disproportionally far away from each other when the deformation occurs. c) illustrates how the vertices can intersect the mesh.*

nodal positions to vertex positions. For shading, an approximate normal was found based on the force calculation. The technique presented in this section enables a "many-to-one" mapping from vertices to nodes through an offset vector. The goal of this technique is to deform highly detailed geometry based on the deformation of less detailed geometry. The less detailed geometry can be as simple as a set of interconnected points.

3.3.1. Defining an offset the in triangle basis

For each vertex $\mathbf{v}$ in the highly detailed surface we associate a triangle of three nodal positions $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$, which control the position of the vertex, see figure 4. The triangle defines a space within which we will represent vertex positions and vectors from the highly detailed model. We need the positions of the vertices to be based on the location, rotation and scaling of the triangle bases. First we define a reference point $\mathbf{r}$ for each vertex. The reference point is chosen as a projection of the vertex onto the triangle base. The reference point will be represented by weights (w_1, w_2, w_3) of the nodal positions:

$$\begin{aligned} \sum_{i=1}^{3} \mathbf{p}_i w_i &= \mathbf{r} \\ \sum_{i=1}^{3} w_i &= 1 \end{aligned} \tag{4}$$

This definition allows the reference points of the vertices to adjust as the triangle scales.

The vertex $\mathbf{v}$ can be expressed as an offset vector $\vec{o}$ from the reference point $\mathbf{r}$:

$$\mathbf{v} = \mathbf{r} + \vec{o}. \tag{5}$$

This vector is expressed in world space. As a next step, we express the offset vector in a vector basis formed by the location of nodes in the associated triangle. Ideally, the offset vector should be affected by both rotation and scaling of the triangle. We simplify the definition of the offset however, taking into account only the rotation of the associated triangle. Depending on the chosen projection of vertices onto triangles, the offset vector will be close to the normal of the triangle, in which case scaling in the triangle plane has little or no effect. Per triangle we define the orthonormal basis ($\vec{T}$, $\vec{N}$, $\vec{B}$) based on $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ as the *triangle basis*:

$$\begin{aligned} \vec{B}_t &= normalize(\mathbf{p}_3 - \mathbf{p}_1) \\ \vec{T} &= normalize(\mathbf{p}_2 - \mathbf{p}_1) \\ \vec{N} &= \vec{T} \times \vec{B}_t \\ \vec{B} &= \vec{N} \times \vec{T} \end{aligned} \tag{6}$$

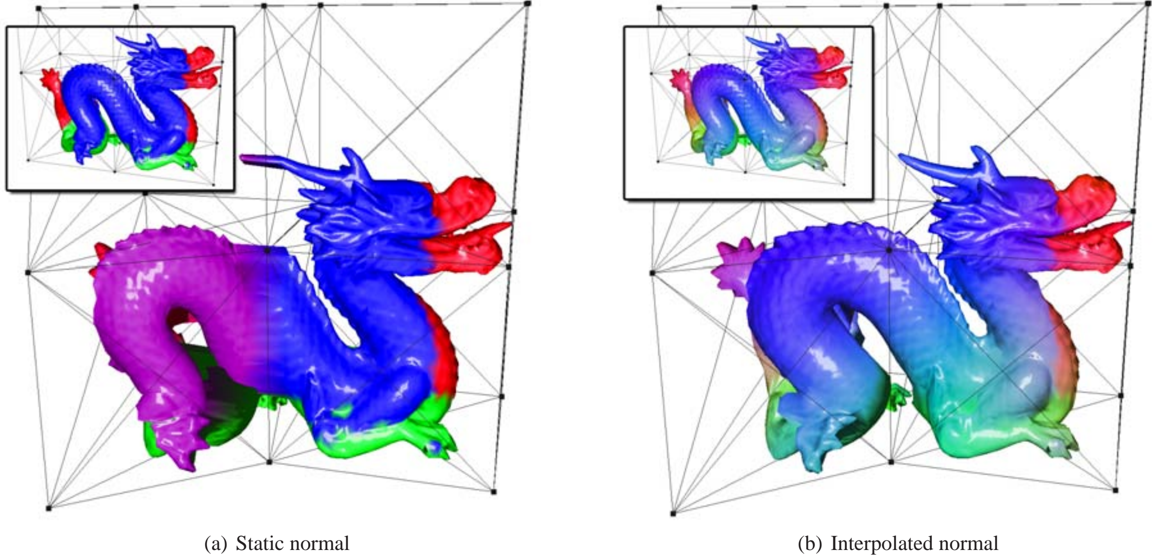
The offset vector $\vec{o^t}$ in the triangle basis is calculated as:

$$\vec{o^t} = \left(\vec{o} \cdot \vec{T}, \vec{o} \cdot \vec{N}, \vec{o} \cdot \vec{B}\right) \tag{7}$$

3.3.2. Curvature across triangles

In the previous section the triangle basis was defined as being constant across each triangle. Consequently, in some cases deforming a mesh of triangles can result in visual artifacts between vertices that are associated to different triangles, see figure 5 and figure 6 a). The deformation of vertices associated to different triangles is not in any way dependent even though they might be very close in the original configuration in world space. A better solution is to interpolate the orthonormal bases across the triangles. This means that the orthonormal basis should be defined at each node and depend on the orientation of all incident triangles. This approximates the curvature of the simulated shape more closely, not only the orientation of each triangle in isolation. The difference is comparable to the difference between flat and

© The Eurographics Association 2005.



(a) Static normal (b) Interpolated normal

Figure 6: *Visualization of the deformation of a dragon [Sta]. a) shows the deformation without correction for curvature and b) shows the deformation with correction for curvature. The shading of the dragon is a combination of normal-mapping and the colour representing the dynamic normal.*

Gouraud shading. A naive implementation would require a large amount of expensive vertex texture fetches, calculating the triangle bases of all surrounding triangles. Instead we address this issue by remembering that we already have approximate normals per node (section 3.2). We do not have a tangent and bi-tangent per triangle node though. Our solution is to assume that the approximate normal $\vec{N_a}(i)$ and the normal $\vec{N}$ (equation (6)) are close enough that we can use the per-triangle tangent $\vec{T}$ and bi-tangent $\vec{B}$ to construct a per-vertex tangent and bi-tangent based on the approximate normal. To compute the interpolated normal for a vertex associated to a given triangle of nodes $(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ we use the weights (w_1, w_2, w_3). The interpolated triangle basis $(\vec{T_c}, \vec{N_c}, \vec{B_c})$ can be expressed as:

$$\begin{aligned} \vec{N_c} &= \textstyle\sum_{i=1}^{3} w_i \vec{N_a}(i) \\ \vec{B_c} &= \vec{N_c} \times \vec{T} \\ \vec{T_c} &= \vec{B_c} \times \vec{N_c} \end{aligned} \tag{8}$$

Any vertex on the highly detailed geometry can now be expressed in the interpolated triangle basis as in equation (7).

3.3.3. Implementation

The entire highly detailed mesh is sent to the GPU for visualization through OpenGL as vertices arranged in a mesh. For all vertices we pre-calculate the nodal weights (to express the reference point) and the offset in the triangle basis on the CPU *once*. This information is given as vertex attributes to the vertex program. The nodal positions reside in texture memory; consequently we give three texture-coordinates as additional per-vertex attributes to resolve the current positions $(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3)$.

In the vertex program three texture lookups are used to retrieve the current nodal positions. Finding the vertex position in the vertex program is equivalent to the construction of weights (equation (4)), triangle-basis (equation (6) or (8)), and offset-vector (equation (7)) on the CPU. On the GPU these calculations are done "backwards" compared to the CPU; resulting in the current vertex point $\mathbf{v}'$ based on the configuration of the current positions $(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3)$:

$$\begin{aligned}
& lookup(\mathbf{p}'_1, \mathbf{p}'_2, \mathbf{p}'_3) \\
& \mathbf{r}' = \textstyle\sum_i^3 \mathbf{p}'_i w_i \\
& \\
& \vec{B_t} = normalize(\mathbf{p}'_3 - \mathbf{p}'_1) \\
& \vec{T}' = normalize(\mathbf{p}'_2 - \mathbf{p}'_1) \\
& \vec{N}' = \vec{T}' \times \vec{B_t} \\
& \vec{B}' = \vec{N}' \times \vec{T}' \\
& \\
& unpack(\vec{N'_a}(1), \vec{N'_a}(2), \vec{N'_a}(3)) \\
& \vec{N'_c} = \textstyle\sum_{i=1}^{3} w_i \vec{N'_a}(i) \\
& \vec{B'_c} = \vec{N'_c} \times \vec{T}' \\
& \vec{T'_c} = \vec{B'_c} \times \vec{N'_c} \\
& \\
& \vec{o}' = \begin{cases} (\vec{o^t_x} \cdot \vec{T'_c}, \vec{o^t_y} \cdot \vec{N'_c}, \vec{o^t_z} \cdot \vec{B'_c}) & \text{if interpolated} \\ (\vec{o^t_x} \cdot \vec{T}', \vec{o^t_y} \cdot \vec{N}', \vec{o^t_z} \cdot \vec{B}') & \text{if constant} \end{cases} \\
& \\
& \mathbf{v}' = \vec{o}' + \mathbf{r}'
\end{aligned} \tag{9}$$

© The Eurographics Association 2005.

3.3.4. Shading

Simple Gouraud shading of the highly detailed mesh depends on a per vertex normal. An optimal normal for shading is possibly different from the normal of the triangle basis. Consequently a pre-calculated normal for the initial mesh configuration is expressed in the triangle basis hereby reflecting the deformation of the nodes. Tangent-space based shading such as normal mapping or parallax mapping is possible if we express vectors defining the tangent space in the triangle basis. These additional vectors are given as per-vertex attributes.

3.4. Results

The fragment and vertex programs have been implemented on a GeForce 6800 Ultra graphics card. For shading of all the illustrated models we have used normal mapping.

The heart model in figure 1 has approximately 50.000 faces. The simulation grid consists of 20.000 nodes. The triangle-basis based visualization with correction for curvature can be visualized at 150 fps. Including 15 simulation steps per visualized frame results in 25 fps. This amounts to 6.67 milliseconds per visualization step and 2.22 milliseconds per simulation step with our current implementation. Without curvature correction each visualization of a frame takes 5.26 milliseconds (190 fps).

As a test for the mapping of a high resolution surface onto a very low resolution set of nodes we used the dragon model [Sta] consisting of 9.971 faces controlled by just 18 nodes. In figure 6 the dragon is visualized with the two variations of triangle-basis based mapping from section 3.3.

The visual difference between the one-to-one mapping of node positions to vertex positions and the triangle-basis based mapping with offsets to vertex positions can be inspected in figure 2.

3.5. Discussion and conclusion

We use the GPU hardware for both visualization and calculation of deformation. It is important to realize that the speed of both simulation *and* the visualization of the spring mass system is faster than a conventional CPU implementation. The GPU implementation of the spring mass system without visualization is 20 to 30 times faster than a similar CPU implementation [MHS05]. Secondly, the GPU can cache the highly detailed surface for visualization in the case of the GPU based simulation because the definition of vertex attributes does not change. A CPU implementation of a spring-mass system cannot use this technique because new vertex positions are calculated every frame, and these need to be sent to the GPU.

The visualization methods presented have various trade-offs that must be considered in choosing which visualization technique to use. Using triangle-basis based mapping, as presented in section 3.3, allows us to decouple visualization and simulation. This allows e.g. visualization of higher resolution or smoother appearance than the set of simulation nodes represents, see figure 2. If this is not necessary, the simpler one-to-one mapping can be used instead. If tangent-space based shading is to be used, e.g. for normal mapping, the one-to-one mapping does no directly support this since only an approximate normal is available. In such cases the triangle-basis based mapping is useful even if an offset vector is not needed.

The triangle-basis based mapping can be simplified somewhat if we assume that the offset is always along the dynamically calculated normal. In that case we do not need to create the triangle basis to express the offset vector. We still need to perform three vertex texture-lookups to interpolate the approximate normals though. Remember, the construction of the triangle basis is still necessary though if we wish express other vectors in the deforming space defined by the triangle - e.g. to use normal mapping.

The approximate normal $\vec{N}_a$ (see equation (3)) depends on certain properties of the nodes. The method is most effective if the nodes are arranged in a regular grid, since this guarantees that the neighbours are evenly distributed for calculation of the approximate normal. Very important, there must exist a neighbour that adds to the approximate normal in the direction of the desired surface normal. A thin cloth simulation would not provide correct approximate normals since nodes are only connected in two dimensions. There is no additional neighbour in the third dimension to force the approximate normals to point outwards from the cloth. In the surgical simulator this means that we cannot use the current approximate normals at parts of the organ with just one node depth. The triangle-basis based mapping that corrects for curvature through the approximate normals naturally has the same dependence on the property of nodes. The triangle-basis based mapping with a constant triangle basis across triangles can be used instead to visualize the geometry in these cases. The heart model in figure 1 is rendered with the triangle-basis based mapping without correction for curvature because parts of the blood-vessels are simulated as "one node thin" sheets.

If the offset vector in triangle-basis based mapping is sufficiently short compared to the relative resolution of the set of nodes or if the reference point is close to a node, we can leave out correction for curvature without compromising the visual appearance. In the case of the heart presented in figure 1 the correction for curvature can be left out, but in the case of the dragon in figure 6 the difference is very evident.

3.6. Future work

It would be interesting to look into the possibilities of an automatic level of detail (LOD) in the spring-mass simulation coupled with the triangle-basis based visualization. This

© The Eurographics Association 2005.

would enable a seamless change between resolutions of simulation because each level of the LOD can be mapped to the same high-resolution surface. Dynamic LOD could be implemented by exploiting hardware accelerated linear interpolation of texture lookups. Such a representation would probably include some hierarchical representation of vertices and nodes on the GPU, and could possibly also be used as an acceleration structure for intersection tests without the transfer of data to the CPU.

In cases when the triangle-basis based mapping with correction for curvature is the preferred method of visualization but the approximate normal is not well defined for all nodes, the two methods could be combined. In pre-processing on the CPU we could identify the problematic nodes and use the mapping without correction for curvature for these vertices only. The choice of method could be included as a per-vertex attribute to let the vertex program choose between the two methods on a per-vertex basis.

In a normal utilization of the graphics pipeline for drawing geometry, fragment processing is conserved through depth-buffer culling. In future work we will look into removing part of the potentially large amounts of computation done for *vertices* that are hidden by other geometry. In vertex programs we have no equivalent to depth-buffer culling of fragments though, since the information is not available until we have calculated the vertex positions. Through utilization of the low-resolution set of nodes and approximate normals, some information is available though.

Acknowledgements

Funding by the Danish Research Agency (grant #2059-03-0004).

References

[BFGS03] Bolz J., Farmer I., Grinspun E., Schröoder P.: Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3 (2003), 917–924.

[Bli78] Blinn J. F.: Simulation of wrinkled surfaces. In *Proc. SIGGRAPH'78* (1978), pp. 286–292.

[BP04] Buck I., Purcell T.: A toolkit for computation on gpus. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, ch. 37.

[Coo84] Cook R. L.: Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 223–231.

[HCSL02] Harris M. J., Coombe G., Scheuermann T., Lastra A.: Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 109–118.

[HEGD04] Hirche J., Ehlert A., Guthe S., Doggett M.: Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004), Canadian Human-Computer Communications Society, pp. 153–158.

[Kry05] Kryachko Y.: Using vertex texture displacement for realistic water rendering. In *GPU Gems 2*, Fernando R., (Ed.). Addison-Wesley, 2005, ch. 18.

[KTI*01] Kaneko T., Takahei T., Inami M., Kawakami N., Yanagida Y., Maeda T., Tachi S.: Detailed shape representation with parallax mapping. In *Proc. ICAT* (2001), pp. 205–208.

[MHS05] Mosegaard J., Herborg P., Sørensen T. S.: A gpu accelerated spring mass system for surgical simulation. In *Proc. Medicine Meets Virtual Reality 13* (2005), vol. 111, pp. 342–348.

[MS05] Mosegaard J., Sørensen T. S.: Gpu accelerated surgical simulators for complex morphology. In *Proc. Virtual Reality* (2005), pp. 147–153,323.

[Sta] Stanford University, Computer Graphics Laboratory: Dragon, the stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep`.

[WWT*03] Wang L., Wang X., Tong X., Lin S., Hu S., Guo B., Shum H.-Y.: View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3 (2003), 334–339.

© The Eurographics Association 2005.