International Scholarly Research Network
ISRN Computer Graphics
Volume 2012, Article ID 936315, 8 pages
doi:10.5402/2012/936315

Research Article

A Novel GPU-Based Deformation Pipeline

Muhammad Mobeen Movania and Lin Feng

Division of Information Systems, School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798

Correspondence should be addressed to Muhammad Mobeen Movania, mova0002@e.ntu.edu.sg

Received 17 August 2011; Accepted 24 September 2011

Academic Editor: C.-M. Wang

We present a new deformation pipeline that is independent of the integration solver used and allows fast rendering of deformable soft bodies on the GPU. The proposed method exploits the transform feedback mechanism of the modern GPU to bypass the CPU read-back, thus, reusing the modified positions and/or velocities of the deformable object in a single pass in real time. The whole process is being carried out on the GPU. Prior approaches have resorted to CPU read-back along with the GPGPU mechanism. In contrast, our approach does not require these steps thus saving the GPU bandwidth for other tasks. We describe our algorithm along with implementation details on the modern GPU and finally conclude with a look at the experimental results. We show how easy it is to integrate any existing integration solver into the proposed pipeline by implementing explicit Euler integration in the vertex shader on the GPU.

1. Introduction

Physically based deformation is an actively sought out area of research. Real-time deformations of 3D volumes as well as polygonal objects have been used in the modeling of real-time surgery simulators in the past with approaches such as FEM-based models [1, 2], ray deflectors [3, 4], mass spring models [5, 6], and chain mail algorithm [7]. In this paper, we limit our discussion to the mass spring models since they are among physically based methods of deformation. This restriction is for convenience only and it is easy to apply the proposed ideas to any of the existing deformation methods cited earlier since all of the models follow the same physical laws. How they follow the laws is subjective. Our main contribution is the development of a new pipeline on the GPU using the transform feedback mechanism for physically based deformation that is independent of the integration solver and the deformation method used.

Prior approaches are based on the GPGPU approach that requires rendering of positions and velocities into separate offscreen buffers using multiple passes. Moreover, these approaches are using the fragment pipeline only which results in an imbalanced utilization of the programmable graphics pipeline. In contrast, our approach performs the deformation calculation in the vertex shader in a single pass. In addition, the fragment shader may be utilized for other rendering tasks, for example, volume rendering of the deformed elements. Since the transform feedback mechanism allows points to be deformed directly on the GPU without a CPU transfer, this saves the GPU bandwidth significantly, preventing the stalls which might have taken place due to the CPU-GPU transfer. Utilizing the proposed pipeline, we can obtain a much more concise implementation as we will see using the cloth simulation as an example.

2. Previous Work

Deformable models allow us to portray the accurate real world behavior of a 3D object be it a polygonal geometry or a 3D volume. Without these models, the 3D world would seem characterless. In the case of the surgical simulators, this realism is particularly important since this dynamic behavior adds to the visual cues which play a significant role during the surgical training. There has been a considerable amount of research on physical deformable models in computer graphics. Rather than listing out all of the approaches one by one, we refer the readers to the following survey [8].

Numerous deformation models have been proposed. One such model, the ray deflectors method [3, 4], involves bending of viewing rays using translate, rotate, scale, and discontinuous operators for deformation. It should be noted that the operators are limited to spheres only; thus, it is difficult to approximate more complex topologies without a very large number of spheres of varying radii. This in turn requires significant resources which makes this method non-real time.

With the introduction of the 3D texture mapping capability in hardware, new and innovative uses of texture memory were proposed. This includes applications ranging from volumetric renderers to volume deformers. Typically, this approach [9, 10] converts the volume into a collection of tetrahedra. Following this conversion, the volume deformation is applied in either model space or volume space using an affine 4×4 matrix. Along with the transformation matrix, several constraints are also imposed [10] to maintain continuity. Unfortunately, this model requires considerable additional processing time during conversion of the volume into its tetrahedral representation. Moreover, continuity constraints have to be met in order to avoid cracks between the tetrahedra during deformation. To circumvent this problem, the skeleton trees method [11] was proposed that uses the skeleton representation of the data to aid as bones for deformation. The same disadvantage as in the texture mapping approach is that the extraction of the skeleton representation requires considerable amount of preprocessing time which is dependent on the dataset resolution.

Similar to the requirement of the intermediate representation for the 3D texture mapping and the skeleton trees approach, the displacement-driven model [12] generates deformations by extracting a high-resolution triangular surface mesh. It constructs a deformed node index table (DNIT) to model the deformation propagation driven by the displacement at a surface contact point (SCP). Surface nodes that undergo deformation are added to the current DNIT according to the triangular mesh topology. It sounds promising for real-time deformation, but this approach is limited to modeling of isotropic elastic deformations only.

The deformations are not just restricted to the spatial domain. One frequency domain approach, the scheduled Fourier method [13], uses the Fourier transforms of the volume to morph from one volume to another. Similarly, the wavelet-based morphing method [14] uses the wavelet domain by first decomposing the volume into a set of frequency bands. These bands are then smoothly interpolated and finally the morphed output is generated from the smoothly interpolated bands. In both of the frequency domain approaches, high frequencies may be introduced during the interpolation step which results in a sequential crystallization of the data. This unfortunately restricts the application of the method.

The quest for real-time deformation led to the development of a new breed of fast algorithms that includes the 3D chain mail algorithm [7]. It approximates the deformation by linking elements as chain rings. The deforming elements only propagate deformation to their immediate neighbors; hence this method of deformation is very fast. However, the deformation is restricted using several constraints and, notably, the underlying model for deformation is not physically based.

With the advent of the GPU, its immense computational power has been used to extend and enhance the deformation models such as those for tetrahedral [2] and multigrid FEM [1]. Noticeably, mass-spring models have been studied heavily for deformable body simulations [5, 6] as well as for surgical simulations [15, 16]. Two of these approaches in [12, 17] try to conserve volume during deformation. The approach in [12] uses special support springs to represent the inner matter of a volumetric object that aids in volume conservation. Unfortunately, a significantly large number of support springs are needed to create realistic deformations. A different approach in [17] uses the Bulk Modulus to restore volume by applying volumetric stresses using circumcenter alignment method. Like the texture mapping algorithm, this method also requires extraction of tetrahedral elements for deformation propagation which adds significant preprocessing time.

The GPU-based approaches in [15, 18, 19] have resorted to the general-purpose GPU, or GPGPU, techniques for evaluating the position and/or velocity integration on the fragment shader. This involves rendering a screen-sized quad with the appropriate textures setup and then the fragment shader is invoked to solve the integration for each fragment. The output from the fragment shader is written to another texture. The GPGPU approaches require algorithms to be moulded considerably so that it is easier to apply GPGPU techniques to them. As demonstrated in [15], such compliance could be achieved by rearranging the dataset in a flat layout. In addition, conforming an algorithm to the GPGPU design requires ample skill and experience, while some GPU-based algorithms can be empowered by GPGPU programming paradigms like CUDA and OpenCL. In fact, implementation with a CUDA kernel does not necessarily accelerate a process, and it depends a lot on the algorithm at hand and how the memory access patterns are in the GPU pipeline.

Largely due to the generality of the mass spring models in physically based modeling, we are also especially interested in its application. We propose a new strategy on the GPU using the transform feedback mechanism for physically based deformation that is independent of the integration solver and its numerical integration schemes. In the following sections, we demonstrate the implementation of the explicit Euler integration scheme to show how the new pipeline is formed. We will also describe how to incorporate other numerical integration schemes in the proposed pipeline. Experimental results will be presented and the performance will be compared with prior work on the GPU.

3. Mathematical Modelling

From the point of view of mathematical modeling, there is a strong overlap among the physical deformable models specifically between the FEM-based model and the mass spring model. An elastic model is based on a 3D mesh of

virtual masses which are linked to their neighbors using massless springs in three ways [20]:

(1) structural springs that link the node to its immediate neighbor in x-, y- and z-axes only,

(2) shear springs that connect the remaining neighbors including all of the diagonal links,

(3) flexion springs that are structural springs connected to the nodes one node away.

Each of these springs is constrained by a different force; that is, under pure stress, shear springs are constrained, under pure compression/traction stress (i.e., stretching), only structural springs are constrained, and under pure flexion stresses (i.e., bending), only flexion springs are constrained. All of the connections act as linear springs which bring the voxel mesh towards equilibrium.

Each node is associated with a set of physical properties including mass ($\mathbf{m}$), position ($\mathbf{x}$), velocity ($\mathbf{v}$), and acceleration ($\mathbf{a}$). At any point in time, the system is governed by the following second order ODE:

$$m\ddot{\mathbf{x}} = -c\dot{\mathbf{x}} + \sum \mathbf{f}_{\text{int}} + \mathbf{f}_{\text{ext}}, \tag{1}$$

where c is the damping coefficient, $\mathbf{f}_{\text{int}}$ is the ith spring force, and $\mathbf{f}_{\text{ext}}$ is the external force which may be due to the user's intervention, wind, or gravity force or collision force due to collision of the object with other objects. The spring force $\mathbf{f}_i$ can be defined as

$$\mathbf{f}_{\text{int}}(t) = k_i\left(\left\|\mathbf{x}_i(t) - \mathbf{x}_j(t)\right\| - l_i\right)\frac{\mathbf{x}_i(t) - \mathbf{x}_j(t)}{\left\|\mathbf{x}_i(t) - \mathbf{x}_j(t)\right\|}, \tag{2}$$

where k_i is the spring's stiffness, l_i is the resting length of the spring, $\mathbf{x}_i$ is the spring's position, and $\mathbf{x}_j$ is the position of its neighbor.

The system in (1) may be solved using any of the numerical integration schemes for, for example, explicit Euler integration, implicit Euler integration, midpoint method (2nd-order Runge Kutta), Verlet integration, or 4th-order Runge Kutta method. Whatever method we use, the acceleration ($\mathbf{a}$) may be calculated using Newton's second law of motion:

$$\mathbf{a}_i(t + \Delta t) = \frac{\mathbf{f}_i(t + \Delta t)}{m_i}. \tag{3}$$

If the explicit Euler integration [6, 20] is used, the velocity ($\mathbf{v}$) and position ($\mathbf{x}$) are updated separately using the following equations:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t\mathbf{a}_i(t + \Delta t), \tag{4}$$

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t). \tag{5}$$

In the case of the Verlet integration, there is no need to calculate and store velocity ($\mathbf{v}$) since the new position ($\mathbf{x}$) is obtained from the current and the previous position, using the following numerical operations:

$$\mathbf{x}_i(t + \Delta t) = 2\mathbf{x}_i(t) - \mathbf{x}_i(t - \Delta t) + \mathbf{a}_i(t)\Delta t^2. \tag{6}$$

For this to work, both the current and the previous positions are needed. When the implicit Euler integration is used, the new position is given as

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t + \Delta t). \tag{7}$$

Note that in (7), the new velocity is used immediately to obtain the new position whereas in (5), the old velocity is used.

In the case of the midpoint Euler method (2nd-order Runge Kutta), the new velocity and the new position are given as

$$\begin{aligned}\mathbf{v}_i(t + \Delta t) &= \mathbf{v}_i(t) + \Delta t\mathbf{a}_i\left(t + \frac{\Delta t}{2}\right),\\ \mathbf{x}_i(t + \Delta t) &= \mathbf{x}_i(t) + \Delta t\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right).\end{aligned} \tag{8}$$

Note that in (8), both the acceleration and the velocity are evaluated at the midpoint between t and $(t + \Delta t)$, that is, $(t + \Delta t/2)$. Likewise, Verlet integration may be refined by evaluating the acceleration ($\mathbf{a}$) and the previous position $\mathbf{x}(t - \Delta t)$ at the midpoint as follows:

$$\mathbf{x}_i(t + \Delta t) = 2\mathbf{x}_i(t) - \mathbf{x}_i\left(t - \frac{\Delta t}{2}\right) + \mathbf{a}_i\left(t + \frac{\Delta t}{2}\right)\Delta t^2. \tag{9}$$

Finally, for the 4th-order Runge Kutta method, the new velocities are first obtained using the following set of operations:

$$\begin{aligned}\mathbf{v}_i(t + \Delta t) &= \mathbf{v}_i(t) + \frac{1}{6}(\mathbf{F}_1 + 2(\mathbf{F}_2 + \mathbf{F}_3) + \mathbf{F}_4),\\ \mathbf{F}_1 &= \frac{\Delta t}{2}\mathbf{a}_i(t + \Delta t),\\ \mathbf{F}_2 &= \frac{\Delta t}{2}\frac{\mathbf{F}_1}{m_i},\\ \mathbf{F}_3 &= \Delta t\frac{\mathbf{F}_2}{m_i},\\ \mathbf{F}_4 &= \Delta t\frac{\mathbf{F}_3}{m_i}.\end{aligned} \tag{10}$$

The new positions are then obtained by the following set of operations:

$$\begin{aligned}\mathbf{x}_i(t + \Delta t) &= \mathbf{x}_i(t) + \frac{1}{6}(\mathbf{k}_1 + 2(\mathbf{k}_2 + \mathbf{k}_3) + \mathbf{k}_4),\\ \mathbf{k}_1 &= \frac{\Delta t}{2}\mathbf{a}_i(t + \Delta t),\\ \mathbf{k}_2 &= \frac{\Delta t}{2}\mathbf{k}_1,\\ \mathbf{k}_3 &= \Delta t\mathbf{k}_2,\\ \mathbf{k}_4 &= \Delta t\mathbf{k}_3.\end{aligned} \tag{11}$$

In an iterative algorithm, setting of the time step (Δt) value is critical. For stability, Courant condition should be met; that is, Δt must be inversely proportional to the square root of elasticity (k) [1, 19].

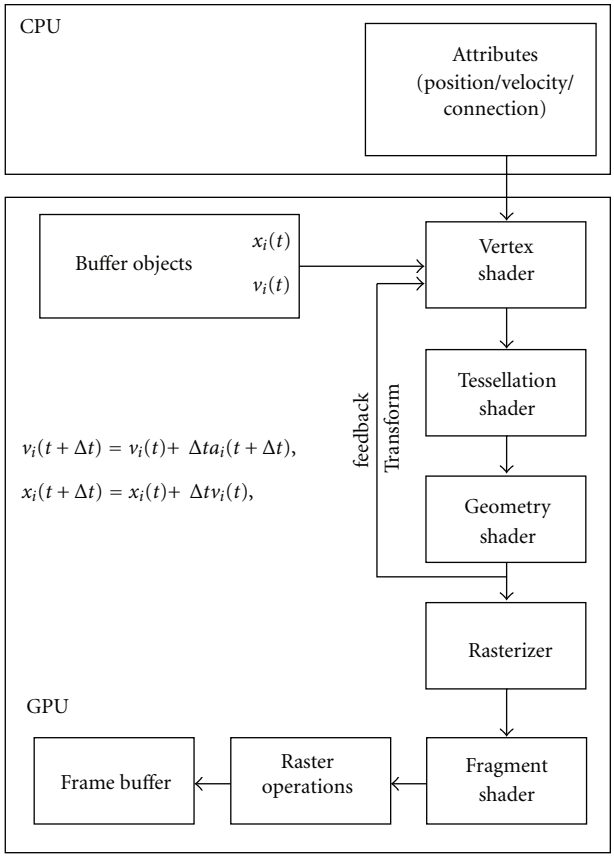

Figure 1: Proposed deformation pipeline using transform feedback.

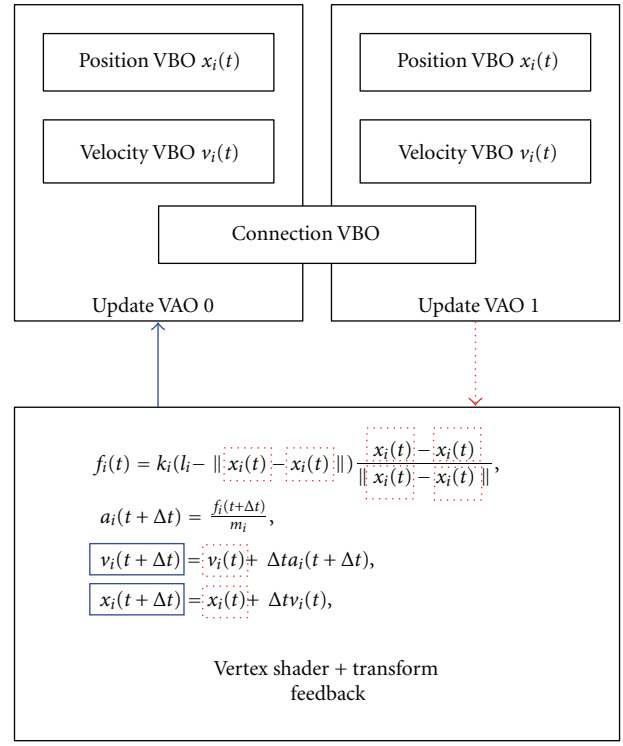

Figure 2: The VAO and VBO setup for transform feedback: the blue solid rectangles show the attributes written to a VAO; the red dotted rectangles show the attributes being read simultaneously from another VAO.

4. The Transform Feedback Pipeline

Prior algorithms like in [15, 18, 19] have resorted to the GPGPU-based techniques for evaluating the position and/or velocity integration on the fragment shader. This involves rendering a screen-sized quad with the appropriate textures setup and then the fragment shader is invoked to solve the integration for each fragment. The output from the fragment shader is written to another texture. On the contrary, we adopt a different approach in this paper (see Figure 1). We implement the mass spring deformation by using the transform feedback mechanism of the modern GPU. This mechanism allows us to push as many vertices as the GPU may handle for deformation.

To understand how the different steps of the algorithm work, for the rest of this discussion, we will be discussing the steps needed to implement the explicit Euler integration as an example. We do the integration calculation on the vertex shader. Then, using transform feedback, we direct the new positions and velocities to a set of vertex array objects (VAOs).

We have two VAOs for updating the physics and two more VAOs for rendering of the resulting positions. Referring to Figure 2 for the following, each VAO stores a set of vertex buffer objects (VBOs) for position and velocity. An additional VBO is required to store the connection information. The connection VBO is also bound to the texture buffer target so that the connectivity information could be fetched in the vertex shader on demand. The usage flags for the position and velocity VBOs are set as dynamic (GL_DYNAMIC_COPY in OpenGL) since the data will be dynamically modified using the shaders. This gives an additional hint to the GPU so that it may put the buffers in the fastest accessible memory.

The application pushes a set of positions (each element is a float4 with xyz in the first three components and mass in the fourth component), velocities (each element is a float3), and one connection texture buffer object (each element is an ivec4) to the GPU. The connection texture buffer object stores the neighborhood information for each mass so that the neighbor's position could be retrieved. The reason we use a set of VBOs for positions and velocities is so that we may use the ping pong strategy to read from a set of position/velocity while we write to another set using the transform feedback approach since we may not write to a transform feedback attribute when we are reading from it.

The vertex shader receives the positions, velocities, and connections as input attributes. The damping coefficient (c) and the spring stiffness (k) are given as shader uniforms. The vertex shader calculates the acceleration by running a loop through all of the neighbors. For fixed masses, a special sentinel value (-1) is used. Thus, if the current position's x component is -1, the point is fixed. In the loop, the neighbor's position and its resting length are obtained. These

values are then used to obtain the current spring's force $\mathbf{f}_i$. This force is added along with the external forces such as wind or gravity. This loop continues for all neighbors. Once the external force is calculated, the acceleration is obtained. Finally, the acceleration is used to get the new velocity. This in turn allows us to obtain the new position.

The new positions then applied certain constraints like the positivity constraint to prevent the masses from falling under the ground plane. The positivity constraint is given as

$$\mathbf{x}_i \cdot y = \begin{cases} \mathbf{x}_{i+1} \cdot y, & \text{if } \mathbf{x}_{i+1} \cdot y > 0, \\ 0, & \text{else,} \end{cases} \tag{12}$$

where $\mathbf{x}_i \cdot y$ is the y component of the position $\mathbf{x}$ assuming that the $\mathbf{y}$-axis is the world up axis. Likewise, other constraints like collision of the mass with an arbitrary polygon may be implemented very easily in the vertex shader. For instance, we consider a constraint on collision of the masses with a sphere. Assuming that we have a sphere having a center ($\mathbf{C}$) and a radius (r), we have a mass at position ($\mathbf{x}_i$) and it is transformed to a new position ($\mathbf{x}_{i+1}$). The collision constraint is given as

$$\mathbf{x}_{i+1} = \begin{cases} \mathbf{C} + \dfrac{(\mathbf{x}_i - C) \cdot r}{|\mathbf{x}_i - C|}, & \text{if } |\mathbf{x}_i - C| < r, \\ \mathbf{x}_i, & \text{else.} \end{cases} \tag{13}$$

5. Implementation and Performance Assessment

We have implemented the proposed pipeline using the GLSL shading language. Setting up of the VAOs and VBOs for transform feedback is described in the previous section. For instant rendering of the deformed objects, we require a pair of VAOs for updating the position and velocity.

Referring to Figure 3, for each rendering cycle, we swap between the two buffers to alternate the read/write pathways. Before the transform feedback could proceed, we need to bind the update VAOs to the current render device so that the appropriate buffer objects can be set up for writing data to. Once the update VAO is bound, we bind the appropriate VBOs for reading the current positions and velocities to the current transform feedback buffer base (by issuing a call to glBindBufferBase OpenGL function). The rasterizer is disabled to prevent the execution of the rest of the programmable pipeline. The draw point call is issued to allow writing vertices to the VBO. The transform feedback is then disabled. The amount of primitives transformed could be queried by issuing a query. Following the transform feedback, the rasterizer is enabled and then the points are drawn. This time, the render VAOs are bound. This renders the deformed points on screen.

This process is repeated until the difference between the current and the previous position of the mass is below a threshold. The threshold value is dependent on the simulation accuracy required. In practice, a value of 0.001 is found to be a good compromise. A recent extension GL_ARB_transform_feedback2 in the core OpenGL 4.0 has eased the transform feedback along with the handling of its buffer object/s. It provides specific states that allow transform feedback to store references to the VBO. The extension also adds some functions that allow rendering of primitives without the need to query the number of primitives written through the transform feedback.

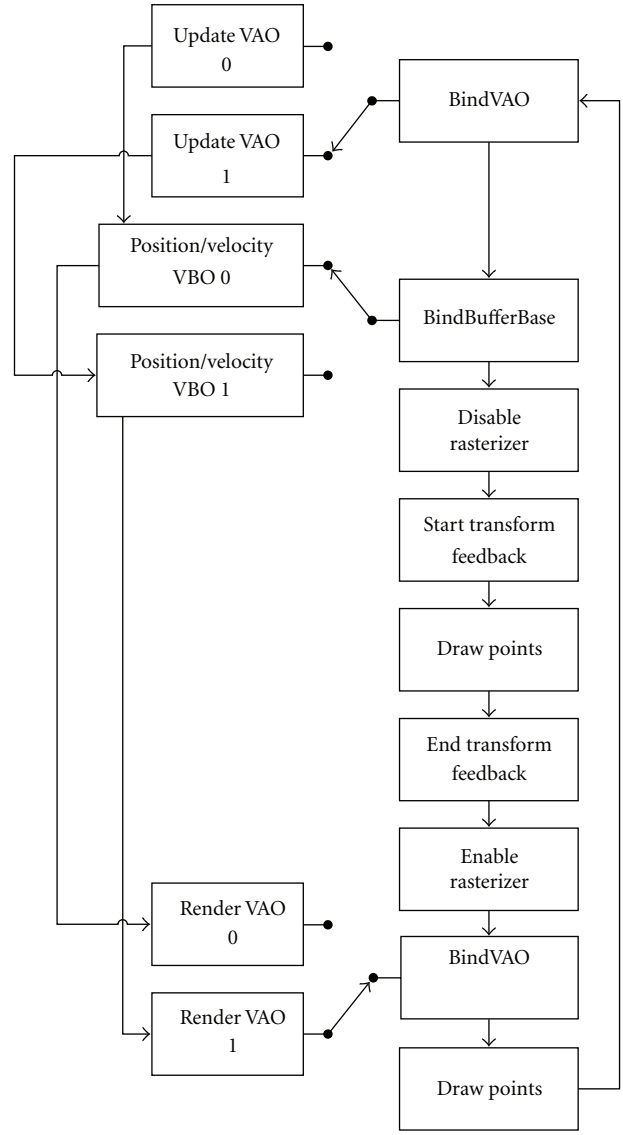

Figure 3: The transform feedback data flow for the update and render cycle.

We have applied the new deformation pipeline on a *Dell Precision* T7500 workstation (*Windows* 7 64-bit) with a 2.27 GHz *Intel Xeon* CPU with 4 GB of RAM. The machine is equipped with a Quadro FX 5800 GPU card with 4096 MB of dedicated video memory. The output resolution for all of the experiments is 1024 × 1024 pixels.

For comparisons of performance, we rendered a deforming grid of points ranging from the size of 64 × 64 grid points to 2048 × 2048 grid points, on both CPU and GPU, as shown in Figure 4. In the experiment, the user arbitrarily

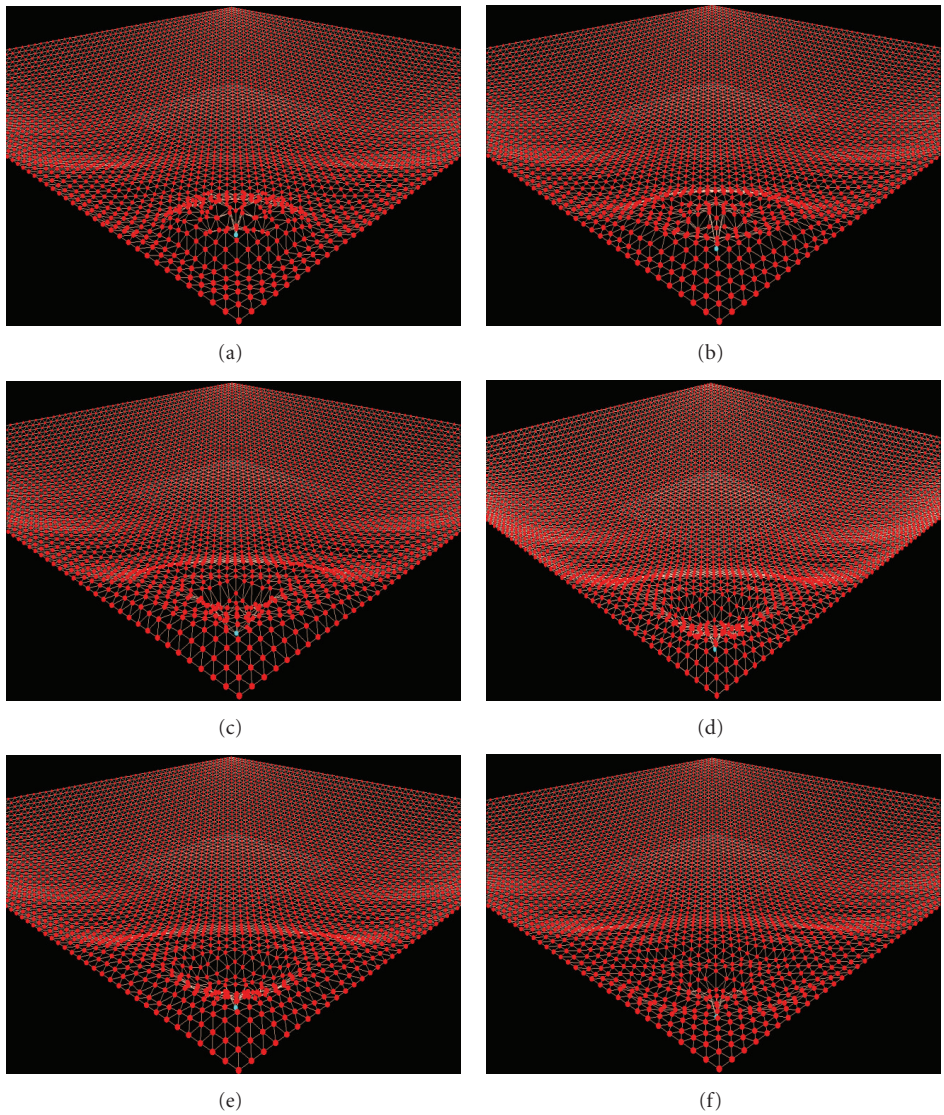

Figure 4: Real-time rendering of deformation of a grid of masses using explicit Euler integration using transform feedback.

picks up a point and moves the point in 3D space. The grid deforms accordingly to bring the grid mesh towards equilibrium. The explicit Euler integration was used as the integration solver for this experiment. Execution time was recorded for both deforming operations and rendering of the updated grid points for a single frame.

For fair comparisons, we optimized the CPU code by utilizing the OpenMP (column *Optimized CPU (b)* in Table 1 and column *CPU (a)* in Table 2). The force accumulation and the spring constraint phase is greatly aided by the OpenMP pragma to accelerate the code by issuing parallel threads. This helps to reduce the execution time for rendering a single frame as shown in Table 1. In addition, an unoptimized CUDA version using global memory (column *GPU-CUDA-UOP (b)* in Table 2) and an optimized CUDA version using the shared memory (column *GPU-CUDA-OP (c)* in Table 2) are also implemented (using the same integration scheme and the grid sizes) followed by our transform feedback-based code (column *GPU-TF (d)* in Table 2). The performance of the four programs is summarized in Table 2. The CUDA versions were tested with different execution configurations, and after various experiments, we found the execution

Table 1: Comparison of unoptimized and optimized CPU code.

Grid size	Time for rendering a single frame (in msecs)		
	Unoptimized CPU (a)	Optimized CPU (b)	Speedup (b/a)
64 × 64	1408.45	99.90	14.09
128 × 128	5263.16	368.83	14.27
256 × 256	16666.67	1515.15	11.00
512 × 512	N/A	5000.04	N/A
1024 × 1024	N/A	200,000.00	N/A
2048 × 2048	N/A	N/A	N/A

Table 2: Comparison of execution performance.

Grid size	Time for rendering a single frame (in msecs)				Speedup (a/d)	Speedup (b/d)	Speedup (c/d)
	CPU (a)	GPU-CUDA-UOP (b)	GPU-CUDA-OP (c)	GPU-TF (d)			
64×64	99.90	4.05	2.557	1.07	93.36	3.79	2.39
128×128	396.82	4.92	3.126	1.47	269.94	3.35	2.13
256×256	1,515.15	6.99	5.742	3.34	453.63	2.09	1.72
512×512	5,000.00	31.06	9.861	11.65	429.18	2.66	0.85
1024×1024	200,000.00	133.69	38.957	54.76	3652.30	2.44	0.71
2048×2048	N/A	699.31	172.408	265.25	N/A	2.63	0.65

configuration of 8×8 threads per block to perform best on our hardware.

As expected, for small grid sizes, even the optimized CPU implementation can also catch up with the requirement of real-time rendering. However, as the grid size increases, the amount of calculations increases which drops the performance of CPU program sharply. The drops in performance for the three versions of GPU implementation also vary. Our proposed GPU program using transform feedback is about 2 to 3 times faster as compared to a naive CUDA implementation (see column *GPU-CUDA-UOP (b)*). The main reason for the slowdown of the naive CUDA program seems to be due to the noncoalesce memory accesses needed to obtain the neighbour node positions making the kernel memory bound. Another reason may be the OpenGL interop that is needed to copy the data to/from the CUDA device from/to the OpenGL API. Since memory read is the most expensive operation and our proposed pipeline using transform feedback reads directly from the GPU memory, we attain significant speedup as compared to a naive CUDA implementation. However, this memory access overhead becomes significant for large grid sizes (compare column *GPU-TF (d)* to column *GPU-CUDA-OP (b)* for large grid size). This is where the versatility of CUDA comes into play since using shared memory and careful memory access patterns, many sequential memory transactions could be coalesced into a single transaction. This feature is unfortunately not available through OpenGL and transform feedback. Thus, an optimized CUDA code using the shared memory performs better for larger grid sizes due to the greater flexibility of memory access provided in CUDA.

To assess the effectiveness of the transform feedback for different integration solvers, we conducted another experiment. We compared the performance of the discussed integration schemes, namely, explicit Euler, implicit Euler, Verlet integration, and midpoint Euler (2nd order Runge Kutta) and 4th order Runge Kutta methods on the proposed GPU pipeline. Table 3 shows the time measured for a single frame including deformation updates and rendering for the same set of grid configurations as were used for the previous experiment.

As can be seen, the overall performances of these integration solvers are close to each other in the transform feedback pipeline. This ensures the stability and scalability of the new algorithm in applications. The Verlet integration has reduced memory requirement (no velocity storage required) and performs well; it is also stable, with an approximation error on the order of $O(n^4)$. From the point of view of stability, the 4th-order Runge Kutta method is found to be the best whereas the explicit Euler method is the worst. Thanks to the flexibility of the proposed pipeline, we have enjoyed minimal efforts to add support for all of the integration schemes.

Table 3: Performance of different integration schemes.

Grid size	Time for rendering a single frame (in msecs)				
	Explicit Euler	Implicit Euler	Midpoint Euler	4th-order R. K.	Verlet
64×64	1.07	0.867	0.905	1.151	0.853
128×128	1.47	1.817	1.806	1.895	1.793
256×256	3.34	3.409	3.464	3.678	3.107
512×512	11.65	11.588	11.088	11.957	11.027
1024×1024	54.76	54.288	54.436	55.157	53.705
2048×2048	265.25	274.725	271.739	327.868	258.397

6. Conclusion

We have presented a novel GPU-based deformation pipeline. Our approach is based on the mechanism of transform feedback available in the new-generation GPUs. To the best of our knowledge, this is the first ever proposal of a pipeline that is using transform feedback for deformation entirely on the GPU. We are confident on the results obtained from the algorithm and would like to expand the model to address specific areas like biomedical modeling and simulation [21, 22].

As expected, when comparing our implementation to an optimized CUDA implementation, the performance of the CUDA implementation is better. We can think of two reasons for this; the first is the ability in CUDA to write to any memory region directly (the scattered writes) and the second is the ability in CUDA to control the shared memory the efficient use of which may allow contiguous memory accesses. This feature is unfortunately unavailable in shader APIs and so clearly the performance suffers specially in case of larger resolutions.

To circumvent such a performance loss, we may use a CUDA kernel to do scattered writes, alongside a GLSL shader. This way CUDA may be used for scattered data writes as

well as for processing the more computationally demanding steps. This hybrid scheme however requires a more rigorous treatment and that will possibly be a future research topic.

Acknowledgment

This work is partially supported by a research grant from the Institute of Media Innovation, Nanyang Technological University, Singapore.

References

[1] J. Georgii and R. Westermann, "A multi-grid frame-work for real-time simulation of deformable volumes," in *Proceedings of the VRI-PHYS Workshop on Virtual Reality Interactions and Physical Simulations*, 2005.

[2] J. Georgii and R. Westermann, "A generic and scalable pipeline for GPU tetrahedral grid rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1345–1352, 2006.

[3] Y. Kurzion and R. Yagel, "Space deformation using ray deflectors," in *Proceedings of the 6th Eurographics Workshop on Rendering*, pp. 21–30, 1995.

[4] Y. Kurzion and R. Yagel, "Interactive space deformation with hardware-assisted rendering," *IEEE Computer Graphics and Applications*, vol. 17, no. 5, pp. 66–77, 1997.

[5] T. Vassilev and B. Spanlang, "A mass-spring model for real time deformable solids," in *Proceedings of the East-West Vision*, pp. 149–154, 2002.

[6] T. Vassilev and R. Rousev, "Algorithm and data structures for implementing a mass-spring deformable model on GPU," Biomedical Physics Papers, Research and Laboratory University Ruse, 2008.

[7] S. F. F. Gibson, "3D ChainMail: a fast algorithm for deforming volumetric objects," in *Proceedings of the Symposium on Interactive 3D Graphics (I3D '97)*, pp. 149–154, April 1997.

[8] A. Nealen, M. Mueller, R. Keiser, E. Boxerman, and M. Carlson, "Physically based deformable models in computer graphic," *STAR Report Eurographics*, vol. 25, no. 4, pp. 809–836, 2005.

[9] S. Fang, R. Srinivasan, S. Huang, and R. Raghavan, "Deformable volume rendering by 3D texture mapping and octree encoding," in *Proceedings of the IEEE Visualization Conference*, pp. 73–80, November 1996.

[10] R. Westermann and C. Rezk-Salama, "Real-time volume deformations," *Computer Graphics Forum*, vol. 20, no. 3, pp. 443–451, 2001.

[11] N. Gagvani, D. Kenchammana-Hosekote, and D. Silver, "Volume animation using the skeleton tree," in *Proceedings of the IEEE Symposium on Volume Visualization*, pp. 47–53, 1998.

[12] P. Chen, K. E. Barner, and K. V. Steiner, "A displacement driven real-time deformable model for haptic surgery simulation," in *Proceedings of the 14th Symposium on Haptics Interfaces for Virtual Environment and Teleoperator Systems*, pp. 499–505, March 2006.

[13] J. F. Hughes, "Scheduled Fourier volume morphing," *Computer Graphics*, vol. 26, no. 2, pp. 43–46, 1992.

[14] T. He, S. Wang, and A. Kaufman, "Wavelet-based volume morphing," in *Proceedings of the IEEE Visualization Conference*, pp. 85–92, October 1994.

[15] J. Mosegaard, "A GPU accelerated spring mass system for surgical simulation," *Studies in Health Technology and Informatics*, vol. 111, pp. 342–348, 2005.

[16] C. A. D. Leon, S. Eliuk, and H. T. Gomez, "Simulating soft tissues using a GPU approach of the mass-spring model," in *Proceedings of the IEEE Virtual Reality (VR '10)*, pp. 261–262, March 2010.

[17] Y. Shen, X. Zhou, N. Zhang, K. Tamma, and R. Sweet, "Realistic soft tissue deformation strategies for real time surgery simulation," Tech. Rep. UMSI 2009/13, University of Minnesota Supercomputing Institute, 2009.

[18] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on GPUs," in *Proceedings of the Simulation and Visualization*, pp. 247–258, 2005.

[19] J. Georgii and R. Westermann, "Mass-spring systems on the GPU," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 693–702, 2005.

[20] Y. Chen and Q.-H. Zhu, "Physically based animation of volumetric objects," Tech. Rep. CVC-980209, 1998.

[21] F. Lin, H. S. Seah, and Y. T. Lee, "Deformable volumetric model and isosurface: exploring a new approach for surface boundary construction," *Computers and Graphics*, vol. 20, no. 1, pp. 33–40, 1996.

[22] F. Lin, H. S. Seah, Z. Wu, and D. Ma, "Voxelization and fabrication of freeform models," *Virtual and Physical Prototyping*, vol. 2, no. 2, pp. 65–73, 2007.