Implementing mesh-based approaches for deformable objects on GPU.

Guido Ranzuglia, Paolo Cignoni, Fabio Ganovelli, Roberto Scopigno [†]

Abstract

These latest years witnessed an impressive improvement of graphics hardware both in terms of features and in terms of computational power. This improvement can be easily observed in computer games, where effects which, until few years ago, could only be achieved with expensive CPU computation are now shown interactively.
Although the GPU has been designed for implementing graphics effects, it is still it basically a processing unit with its own memory, and, being specialized for algebraic tasks, supplies a number of floating point operations per second which is orders of magnitude greater than the CPU.
This suggested to the graphics community that the GPU could also be used for general purpose computation and a number of papers have been published on how to hack the GPU to this target.
Following this trend we propose a framework for using GPU for implementing techniques for deformable objects represented as generic meshes. The framework only assumes than the global computation is the union of local computations, which is true for all the explicit methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture I.3.5 [Computational Geometry and Object Modeling]: Physically based modeling

1. Introduction and previous work

It is well known that modeling the behavior of a deformable object is a time consuming task, because of the amount of calculations necessary to compute the reaction of a deformable object under external forces. Without going into details (please refer to [NMK*05] for an excellent survey) the basic parts for defining a model for deformable objects are:

Forces computation. Given the current shape of the object and the force acting on it, compute the forces produced by the deformation energy stored in the body.

Discretization over space. The object must be partitioned in a, possibly large, set of simple elements. Triangle or tetrahedral meshes are popular choices.

Integration through time. Given a description of the dynamic state of the object (e.g the position and velocity of all its points), compute the description of next state after a small time step (generally smaller than 1/20 second).

For the sake of this discussion is convenient to split the existent methods in two main classes: the *explicit methods* using *explicit integration scheme* and the others, i.e. the methods requiring an *implicit scheme* either for Force Computation of for Time Integration. This distinction is fundamental because the methods of the first class share the important property that the total computation is given by the union of local (to the elements of the discretization) and independent computations.
In other words, implicit methods require the multiplication of a sparse matrix having at least as many columns and rows as three times the number of vertices of the mesh for each simulation step. With an explicit methods this matrix is block diagonal and the multiplication can be performed locally on each finite element and summed up in the vertices.
Explicit methods (and explicit integration step) are broadly used for they are easier to implement and generally more flexible. For example they do not require preprocessing and modifications to the mesh can be done on-the-fly to simulate cuts and lacerations [BMG99, GCMS00, NvdS00].
However, both explicit and implicit methods burden the CPU

[†] Visual Computing Laboratory, ISTI-CNR, email: name.surname@isti.cnr.it

```
 Alg OneStep(dt) {
     for each s  in Springs {
    f =  k_s * ( (s->m1->p - s->m0->) -
s_rl ) /
                        ||s->m1->p - s->m0-
>p|| ;
    s_1->f +=f;
    s_2->f -=f;
    }
     for each m  in Masses {
    m->velocity += m->f/m->mass * dt;
    m->p += m->velocity  * dt;
    }
 }
```

Figure 1: *Straightest implementation of a mass spring system.*

with a large amount of computation.
To evaluate the kind of algorithm involved, consider the explicit mass spring system for cloth simulation, which is by far the easiest and most straightest method. The cloth is modeled with a net of masses and springs connecting the spring to form a lattice. One step of simulation is implemented as in figure 1:

where *s_rl* is the rest length of the spring , *k_s* its elasticity constant, *s_12* the masses to which *s* is connected. This is probably the most seen and the most naive code snippet about deformable objects. Still, it useful to us because it shows three things: 1) even a simple method requires many floating point operations 2) the memory requirement is modest 3) there is no branching. Clearly there are explicit models for which these observations do not hold so strictly.

Why GPU?
Programmable Graphical Processing Units have characteristics that meet those afore mentioned the explicit methods: 1) They provide a higher number of floating point operations per second than the CPU (e.g. nVidia(tm) 7800 Gtx's pixel shader performs around 165 GFLOPS against the 8 of a Pentium IV), which is the reason because the introduction of GPUs has spawned a large number of papers pursuing the goal of using them for general purpose (see [OLG*05] for a recent survey); 2) Memory transfer is an expensive task and memory itself is limited (512 MB on nVidia 7800); 3) Branching, introduced with the latest GPU generations, it's still a costly operation.

GPUs are essentially designed for rendering, so it is not surprising that most of the work is about speeding up rendering techniques such as *ray tracing* [PBMH02, WSE04, LL04], *photon mapping* [CSKSN05,LC03,PDC*03,Hac04], *radiosity* [CHH03], just to cite a few.

A different research trend aims to develop general purpose algorithms, such as solver for linear system [GGHM05, BFGS03], for database management [GLW*04], for sorting algorithms [Gov05] and so on for a number of non graphics applications.
GPU based techniques for simulation have also been proposed. The problems that can be expressed as boundary value problems on a regular grid are particularly suitable for being solved on the GPU [GWL*03]. This is the case of Navier-Stokes equations used in fluids simulation [HCSL02, WLL04, Har04].
Fewer solutions have been proposed for simulation with meshes. In [JT05] a mass spring system is implemented in the GPU by placing particles on a regular grid and using implicit connectivity, so that each internal particle has 18 neighbors. Each particle is associated with a pixel in the PBuffer (referred as *position-buffer*) that stores the particle position. At each simulation step, force computation is performed on the pixel shader, where each particle sum up the force component running over the adjacent particles (i.e. the adjacent texels).
In order to visualize the object, a mesh is associated to the non internal particles of the mass spring system. Each vertex of the surface mesh fetches its position from the position-buffer (shader model 3.0), while the normal, needed to shade the surface, is approximated as the normalized sum of those of the adjacent particles.
The advantage of implicit connectivity is that they need only one texture look up per particle. On the other hand, the amount of texture memory is directly proportional to the accuracy of the representation. An improved version of this approach [JHT05] uses a fine grained mesh for the surface connected with a coarser mass spring system in GPU.
Irregular tetrahedral meshes are used in [GEW05]. In this work, each vertex stores, for each tetrahedron it belongs to, three reference to the other three vertices and the corresponding springs' rest lengths. For each vertex, the force contributions of all the tetrahedra in the one-ring neighborhood are summed. In this case more texture fetches are involved: for each vertex and for each tetrahedron they need three texture fetches to find out the indices of the other three vertices and the three dependent fetches to find out their position.

2. Problem's analysis

Since we want to define an as general as possible framework, instead of an ad hoc method, we will consider a mesh a set of vertices and a set of relations among vertices both vertices and relations may have a number of attributes (e.g. each vertex needs to store its velocity). The only requirement is that the relations have constant arity.

Then, we need a way to store connectivity because, as will be clear soon, a vertex needs to have a pointer to the relations

```
Alg OneStep(dt) {
    for each m  in Masses {
   m->f = m->left_s.k *
   (m->m1.p - m->p) /  ||m->m1.p - m->p|| ;
   m->f += m->right_s.k *
    (m->m2.p - m->p) / ||m->m2.p - m->p|| ;
   m->v += m->f/m->mass * dt;
   m->p += m->v * dt;
   }
}
```

Figure 2: *GPU compliant implementation of the mass spring system (chain of springs)*

that include it. For example in a tetrahedral mesh each vertex needs a pointer to the tetrahedra that include it.

3. GPU's bonds

As aforementioned the most seen snippet of code about deformable object the algorithm in figure 1 cannot directly implemented on a graphics hardware. Due to the architectural bonds that performance reasons impose to the GPU, the modern graphics devices do not expose a texture write instruction at a computed address (scatter operation). A common way to overcome this limitation (when it is possible) is to convert a scatter operation in a gather operation. For example the previous snippet of code may be translated as reported in figure 2.

The textures in a GPU's program represent the analogous of arrays in the CPU's programming paradigm. Typically in a physical simulation there are some vectorial quantities that have to be continuously updated in function of a small time step *dt*. In the code in figure 2, for example, the vertex(mass) position and the vertex speed are integrated through time. From a GPU point of view this implies the need for writing the new calculated values in textures. As previously seen the present graphics hardware does not provide a write texture instruction for a texel's index different from which that is tightly related to the vertex/mass. For this reason is necessary to provide a number of different textures; one for each vertex/mass attribute that has to be dynamically updated. In the other hand the vertex attributes that don't change during the simulation may be maintained in a single texture, using, for example, a constant number of contiguous texels.

4. Our Framework

Although each texel contains a RGBA color information from a GPGPU point of view the four coordinates of a texture's pixel may assume other meanings. In our framework a single entry of a texel may be interpreted in three different ways: a scalar value, a single vector's component, a pointer (an index) to a texel or to an entry in another texture.

As previously seen the GPU architecture create a substantial division between the attributes that remain constant during the simulation and the attributes that assume variable values (please refer to Figure 3).

The textures are mainly divided in three classes:

- *Vertex attributes textures*
- *Relation attributes textures*
- *Vertex - Relations Connectivity texture*

Both vertex and relation textures are sub-divided in two sub-classes that are dictated by the distinction between constant and variable attributes. The textures for the vertex variable attributes (*VerVar*) has all the same dimension (proportional to the vertices' number and influenced by the GPU's characteristics). In the other hand, typically, there is a single texture containing the constat vertex values(*VerConst*); the dimension of this texture with a good grade of approximation is:

$$dim(VerConst) = dim(VerVar) * (floats\_per\_vertex)/4$$[†]

.

The *VerConst* always exists because at least contains necessary two value related to the connectivity texture (*Conn*): an offset and number of relations incident in a single vertex.

In each *Conn*'s texel there are four indexes referred to the textures containing the relation's variable and constant attributes; with this information a vertex can access at the relations that influence him state.

5. A case of study: Mass - Spring System

As an example of the framework's use will consider the management of deformable tetrahedral meshes (simplicial complexes of order 3). In this case the relations are represented by the tetrahedrons that compose the mesh. The only variable attribute that we will put on the *RelVar* will be the four indexes to the vertices of the tetrahedron. In the constant attributes texture *RelConstText*, instead, there will be the six length of the springs at rest time. In this example will consider the elastic constant *k* common to each spring in the system. Using the texture coordinates defined at the four vertices of the activation quad (which size is equal to the dimension of a vertex variable attribute texture) is possible to access to each attributes in the *VerVar* [BI05]. Knowing the dimension of one of these textures and the number of constant attributes for each vertex the address of the attributes in the *VerConstTex* can be easily calculated. In this texture, other than offset and the number of relations per vertex *nelem*, we

[†] In each texel is possible to store four different scalar attributes.

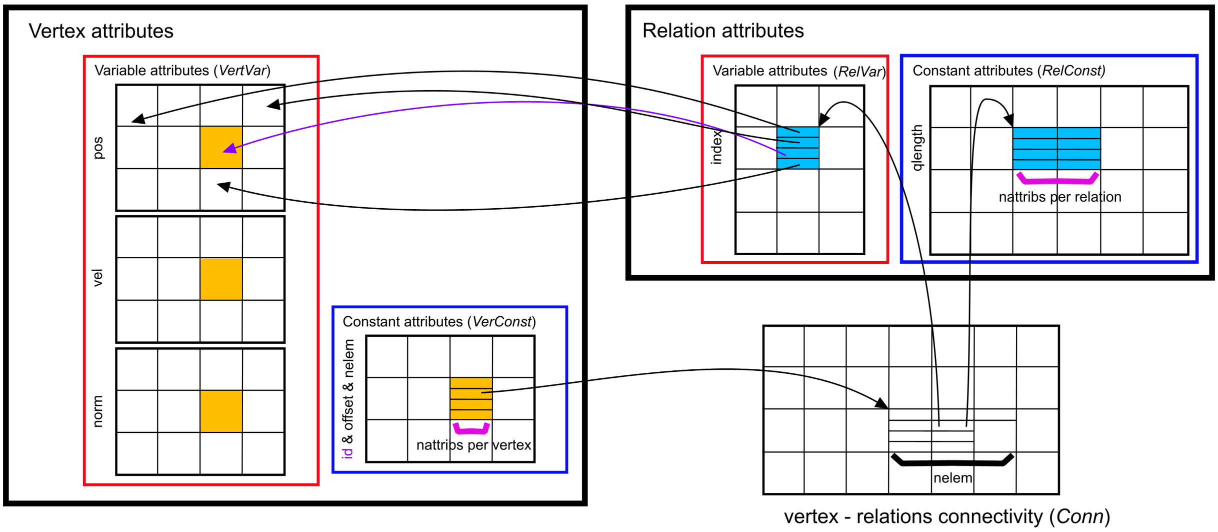
Vertex attributes
Variable attributes (VertVar)
pos
vel
norm
Constant attributes (VerConst)
id & offset & nelem
nattribs per vertex
Relation attributes
Variable attributes (RelVar)
index
Constant attributes (RelConst)
qlength
nattribs per relation
nelem
vertex - relations connectivity (Conn)

Figure 3: *Arrangement of the data in texture memory.*

will insert, for a more plain exposition, the vertex's identification serial. Using offset and nelem is than possible to directly read the attributes stored in *RelVar* and computing the address of the first texel containing the constat attributes in *RelConstTex*. As previously seen the only attributes stored in *RelVar* are the indexes of the vertices that compose the tetrahedron. One of these four indexes will refer to the vertex that is computing the total force incident on himself; in this case, comparing the index stored in the *RelVar* and the id constat attributes, we avoid to make redundant calculus.
If we had a triangular mesh instead of a tetrahedral mesh, the only change in the textures' structure is that the *RelVar* would be containing only three vertex index per texel.

6. Real-time update of a relation's attribute

The distinction between constant attributes texture and variable attributes texture permits to use the last one to update the vertices state. In the same manner in our framework is possible to change dynamically the relation's characteristic directly on GPU; without the necessity of a costly communications between the main system memory and the graphic hardware. It will be sufficient to draw a quad composed of only one fragment. The texture coordinates defined at the quad's vertices will assigned in order to refer to the particular relation that we want to update. If would be necessary to update more than one texel for each attribute's relation, as in the vertex case, we maintain the single attribute in different texture of the same dimension. If in mass-spring system we would eliminate a simplex it will be sufficient to mark the tetrahedron or the triangle in some convenient way (for example we can set the vertices indexes at -1 in the *RelVar*) in order to indicate that this element must be ignored during the physical simulation.
Using the right number of textures to store the spring's characteristics (two for a tetrahedral mesh[‡], only one for a simplicial complex of order 2) can be possible to dynamically change on GPU the elastic constant *k* of each springs in the system.

7. Results

Tables 4 and 5 shows some results when running triangle and tetrahedral meshes. From this table it can be seen that for average sized meshes the GPU implementation is twice as fast that the GPU ones on a NVidia GeForce 6800GT. This is quite a low gain considering the large gap in terms of FLOPS, but it is not too surprising, since our framework uses several dependant fetches per iteration. On the other hand, the mass spring system is somewhat the worst case because it performs few floating point operations per element (triangle, tetrahedron). Table 5 shows the fps for a ill conditioned tetrahedral mesh, where all the tetrahedra share a common vertex. As a consequence, the fragment program for this vertex will have to iterate over all the tetrahedra while all the other per fragment computation (for vertices connected to 6 tetrahedra each) will have to wait.

8. Conclusions and future work

We presented a general framework to handle irregular meshes entirely on the GPU. The framework has been tested

[‡] A tetrahedron is composed by six edges/springs so it's no possible to maintain the information in a single texel.

Triangle meshes			
masses	triangles	GPU	CPU
8321	16384	108	90
24578	49152	31	20
33025	65536	29	15
65538	131072	15	7
131585	262144	8	3

Figure 4: *GPU-CPU comparison for triangle meshes. The last two columns show fps.*

Tetrahedral meshes			
masses	tetrahedra	GPU	CPU
2783	13200	70	48
3375	16464	57	39
4096	20250	48	31
9261	48000	28	13
12167	63888	22	9
17576	93750	16	6
29791	162000	10	4

Figure 5: *GPU-CPU comparison for triangle meshes. The last two columns show fps.*

by implementing mass spring systems for simplicial complexes of order 1, 2 and 3 (chains, triangle meshes and tetrahedral meshes respectively), but it supports with no modifications other types of structure (e.g. quadrilateral or hexahedral meshes) as long as it is expressed as a set of vertices and a set of *relations* among groups of vertices, with the only assumption the all the relations connected the same number of vertices. With respect to previous approaches, the framework requires more dependent fetches, but on the other hand it saves texture memory and, differently from [GEW05] does not cause replication of data.

This will allow us to implement accurate methods that require much more memory per element that the mass spring systems, such as the so called explicit FEM [OH99], which is the next step of this work.

Worst Case			
masses	tetrahedra	GPU	CPU
2562	5120	64	84
10242	20480	42	19

Figure 6: *Worst case: all the tetrahedra share a common vertex.*

References

[BFGS03] Bolz J., Farmer I., Grinspun E., Schröder P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *Proceedings of ACM SIGGRAPH 2003* (2003), Hodgins J., Hart J. C., (Eds.), vol. 22(3) of *ACM Transactions on Graphics*, pp. 917–924.

[BI05] Buck I. Govindaraju N. K. J. L. A. P. T. W. C.: Gpgpu: General-purpose computation on graphics hardware. SIGGRAPH 2005, Course 39 Course Notes, 2005.

[BMG99] Bielser D., Maiwald V., Gross M.: Interactive cuts through 3-dimensional soft tissue. *Computer Graphics Forum (Eurographics'99 Proc.) 18*, 3 (Sept. 1999), C31–C38.

[CHH03] Carr N. A., Hall J. D., Hart J. C.: GPU algorithms for radiosity and subsurface scattering. In *Proceedings of Graphics Hardware 2003* (2003), pp. 51–59.

[CSKSN05] Czuczor S., Szirmay-Kalos L., Szecsi L., Neumann L.: Photon map gathering on the GPU. In *Proceedings of Eurographics 2005* (Sept. 2005). Short Presentations.

[GCMS00] Ganovelli F., Cignoni P., Montani C., Scopigno R.: A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum 19*, 3 (2000), ??–??

[GEW05] Georgii J., Echtler F., Westermann R.: Interactive simulation of deformable bodies on GPUs. In *SimVis* (2005), pp. 247–258.

[GGHM05] Galoppo N., Govindaraju N. K., Henson M., Manocha D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC'2005 Conference CD* (Seattle, Washington, USA, Nov. 2005), IEEE/ACM SIGARCH.

[GLW*04] Govindaraju N. K., Lloyd B., Wang W., Lin M. C., Manocha D.: Fast computation of database operations using graphics processors. In *SIGMOD Conference* (2004), pp. 215–226.

[Gov05] Govindaraju N.: Gpusort: sorting. *http://gamma.cs.unc.edu/GPUSORT/* (2005), 625–634.

[GWL*03] Goodnight N., Woolley C., Lewin G., Luebke D., Humphreys G.: A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the 2003 Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware (EGGH-03)* (Aire-la-ville, Switzerland, July 26–27 2003), Mark W., Schilling A., (Eds.), Eurographics Association, pp. 102–111.

[Hac04] HACHISUKA T.: Final gathering on GPU. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors* (Aug. 2004).

[Har04] HARRIS M.: Fast fluid simulation on the GPU. In *GPU Gems*, Fernando R., (Ed.). Addison Wesley, 2004, ch. 38, pp. 637–665.

[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.

[JHT05] J. M., HERBORG P., T.S. S.: A GPU accelerated spring mass system for surgical simulation. In *Medicine Meets Virtual Reality* (2005), pp. ???–???

[JT05] J. M., T.S. S.: Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the GPU. In *IPT and EGVE Workshop* (2005), pp. ???–???

[LC03] LARSEN B. S., CHRISTENSEN N. J.: Optimizing photon mapping using multiple photon maps for irradiance estimates. In *WSCG* (2003).

[LL04] LING .-J., LING .-J.: A graphics architecture for ray tracing and photon mapping, 2004.

[NMK*05] NEALEN A., MÜLLER M., KEISER R., BOXERMANN E., CARLSON M.: Physically based deformable models in computer graphics. 71–94.

[NvdS00] NIENHUYS H.-W., VAN DER STAPPEN A. F.: Combining finite element deformation with cutting for surgery simulations. In *EuroGraphics Short Presentations* (2000), de Sousa A., Torres J., (Eds.), pp. 43–52.

[OH99] O'BRIEN J. F., HODGINS J. K.: Graphical modeling and animation of brittle fracture. In *SIGGRAPH* (1999), pp. 137–146.

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (Aug. 2005), pp. 21–51.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *j-TOG 21*, 3 (July 2002), 703–712.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of Graphics Hardware 2003* (2003), pp. 41–50.

[WLL04] WU E., LIU Y., LIU X.: An improved study of real-time fluid simulation on GPU. *Journal of Visualization and Computer Animation 15*, 3-4 (2004), 139–146.

[WSE04] WEISKOPF D., SCHAFHITZEL T., ERTL T.: GPU-based nonlinear ray tracing. *Comput. Graph. Forum 23*, 3 (2004), 625–634.