Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2009)
H. Prautzsch, A. Schmitt, J. Bender, M. Teschner (Editors)

Neighboring-based Linear System for Dynamic Meshes

S. Pena Serna[1], J. Silva[1,2], A. Stork[1,3] and A. Marcos[2]

[1]Fraunhofer-IGD, Germany
[2]University of Minho, Portugal
[3]TU-Darmstadt, Germany

Abstract

A linear system is a fundamental building block for several mesh-based computer graphics applications such as simulation, shape deformation, virtual surgery, and fluid/smoke animation, among others. Nevertheless, such a system is most of the times seen as a black box and algorithms do not deal with its optimization. Depending on the number of unknowns, the linear system is often considered as an obstacle for real time application and as a building block for offline computations. We present in this paper, a neighboring-based methodology for representing a linear system. This new representation enables a compact storage of the set of equation, flexibility for ordering the unknowns and a rapid iterative solution, by means of an optimized matrix-vector multiplication. In addition, this representation facilitates the modification of part of the linear system without affecting its unchanged part and avoiding the complete rebuild of the system. This specially benefits applications dealing with dynamic meshes, where the geometry, the topology or both are constantly changed. We present the capabilities of our methodology in models with different sizes and for different operations, highlighting the dynamic characteristic of the mesh. We believe that several applications in computer graphics could benefit from our methodology, in order to improve their convergence and their performance, reducing the number of iterations and the computation time.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation I.6.3 [Simulation and Modeling]: Applications—G.1.3 [Numerical Analysis]: Numerical Linear Algebra—Linear systems

1. Introduction

Computer graphics has become a multidisciplinary community with a vast variety of research fields. Nowadays it does not only deal with graphics and interaction, it also addresses several aspects from engineering, physics, mathematics and numerical analysis, among others. Because of that, other communities have been paying attention to the evolution of computer graphics and at the same time, computer graphics researchers are adapting techniques from other disciplines to achieve more interactivity and realism. Although some of these adaptations are achieving promising results, there are other techniques which are adopted without major changes, leading to results, but neglecting a higher degree of integration and optimization. These drawbacks are encountered in techniques which are adopted as black boxes, where only inputs and outputs are considered, without analyzing the internal process.

A linear system is an example of black boxes applied in computer graphics. Several mesh-based algorithms such as simulation, shape deformation, virtual surgery, and fluid/smoke animation make use of it without major modifications. These algorithms need to traverse the mesh, in order to identify the neighborhoods edges-elements and vertices-elements, to define the dependencies of the equations and to build the linear system. The same process is done, whenever a change in the geometry and the topology of the mesh occurs, increasing the computation time and therefore affecting the overall performance. Nevertheless, there are techniques in computer graphics, which could lead to improvements in this process, by means of representing the linear system in a different form, rather than in a matrix form.

DOI: 10.2312/PE/vriphys/vriphys09/095-103

The exploration of different representation for a linear system was motivated by our aim to find an algorithm, which is able to handle the simulation of dynamic meshes (meshes with continuous geometrical and topological changes) at interactive rates. We first realized that when a local modification in the geometry of the mesh occurs, a great part of the linear system remains unchanged; however the lack of neighboring information restricts a local modification (update) of the system. Additionally if topological changes occur, the system will need to add or remove topological entities, which will imply the addition or removal of equations (rows and columns) in the system respectively. Hence, we started investigating strategies, which could easy the update of the set of equations during geometrical and topological changes of the mesh.

This investigation led us to find a neighboring-based representation of the linear system, which enables a compact storage of the equations, flexibility for ordering the unknowns and a rapid iterative solution, by means of an optimized matrix-vector multiplication. This new representation was found by studying different applications within the computer graphics community, which provided us some hints regarding their utilization of this kind of systems. We also analyzed how the discretization methods for physical problems work, as well as different solvers, aiming at understanding their requirements (section 2). This information helped us developing a methodology, which can effectively improve the build, update and solution of linear systems and which can also reduce the consumption of resources (section 3). We implemented the methodology, in order to evaluate and compare its performance. We tested the capabilities of our methodology in models with different sizes and for different operations, highlighting the dynamic characteristic of the mesh (section 4).

2. Related Work

As we stated in the previous section, we started studying different techniques in computer graphics such as deformable models ([NMK*06]), shape modeling ([ACF*06]), animation ([MSJT08]) and simulation ([BFMF06]), in order to understand how linear systems are used within this community. The study of these techniques revealed us, that linear systems are used in a classical way and that there are no optimization procedures to minimize its build time. Hence, we decided to analyze partial differential equations ([Lan03]) and discretization techniques, particularly the Finite Element Method ([Hug87] and [SG04]), in order to explore the requirements for building linear systems.

Additionally, we wanted to understand how the solution of a linear system is computed, hence we revised the literature regarding iterative solvers ([SVDV00]) and specially the conjugate gradient method ([She94]). Based on this information, we realized that we need to concentrate in a physical problem, since the generated systems could present different properties concerning symmetry, definiteness, among others, and depending on this properties, there are solvers which are better suited than others. Hence, we decided to concentrate our effort in the solid mechanics problem [Bow09] and to investigate the build and the solution of the linear system for this kind of physics.

Augarde et al. [ARS06] explained that in the linear elasticity problem, the Galerkin method causes the stiffness matrix to be symmetric and positive definite. This fact makes the Conjugate Gradient Method a suitable solver for the linear system of equations yielded in the linear elasticity problem. Saad and Vorst presented, in [SVDV00], an in-depth historical perspective of iterative solutions of linear systems and they attributed their origin to the work of Gauss in the early nineteenth century and show how the main contributions over the years led to the iterative solvers we have nowadays.

The studied related work suggested us, that the Conjugate Gradient method is currently the most appropriate solver for computing the solution of a linear system of equations in an iterative form and without using a hierarchy of discretizations or adaptivity methods. Hence, we decided to implement the Conjugate Gradient Method based on [She94], where he presented a practical explanation on how the Conjugate Gradient works and also showed the building blocks and its interconnections, i.e. the method of Steepest Descent, the method of Conjugate Directions and finally their relations within the Conjugate Gradient method.

As mentioned before, our work was motivated by our aim to find an algorithm, which is able to handle the simulation of dynamic meshes at interactive rates. The computer graphics community has not deeply dealt with the simulation of changing meshes, but there are some interesting approaches. Bro-Nielsen ([BNC96, BN96]) presented the advantages of condensed or Fast Finite Elements for deformable models in surgery simulation, where only the surface of the volumetric model is considered for the simulation. Gissler et al. [GBT07] proposed recently constraint sets for FE models, where topological changes (merging and breaking) of deformable tetrahedral meshes are supported, by means of replacing mass points by mass portions (in a constraint set) according to the number of incident tetrahedra.

Klinger et al. [KFCO06] developed a fluid animation application, which requires the remesh of the whole model after every iteration (or mesh change), leading also to a rebuild of the linear system. Von Funck et al. [vFTS06] developed an algorithm for shape deformation based on time-dependent divergence-free vector fields, for which they need to build a linear system, in order to find the path between steps. They also implemented a remesh step, since the deformation of the mesh between steps led to poor quality triangles, which affected the convergence of the linear system. Hence, they also need to rebuild the linear system after every remesh step.

There are several algorithms dealing with the simulation of physically-based deformable models. For example Mezger et al. [MTPS08] proposed a real time physically-based shape editing algorithm based on the simulation of the mechanical properties of the model with a Finite Element Method discretization. However they used computing meshes with few tetrahedral elements (up to 1500) and the rebuild of the linear system (for every geometry change, the topology is constant) did not cause any performance problem. These kinds of simulations with bigger meshes (around 10.000 tetrahedral elements) will not achieve real time performance.

We believe that several applications in computer graphics could benefit from our methodology, in order to improve their convergence and their performance, reducing the number of iterations and the computation time. The methodology proposed in this paper, will specially contribute to the rapid simulation of dynamic meshes. To the best of our knowledge, there are no investigations in the same direction as the proposed in this paper. There have been made several efforts regarding the improvement of solvers, either with new methods or with parallelization techniques on the CPU or the GPU, but there is not enough information about the intelligent handling and processing of linear systems.

3. Methodology

The algorithm, which we are proposing, aims at effectively building, updating and solving linear systems, by means of reducing the processing time and of minimizing the memory consumption. A linear system represented in the matrix form is:

$$Ax = b \tag{1}$$

where A is the matrix of coefficients, x is the vector of unknowns and b is the vector of solutions. Based on the study of the related work, we have understood the objective of this system in getting the solution of the problem. The matrix of coefficients aims at collecting the information for the equations regarding the connectivity of the vertices (edge topology) and of the unknowns (vertices without boundary conditions) themselves. The process for building the matrix of coefficients is normally based on traversing the given mesh (element by element), in order to identify the neighboring elements, which need to contribute to an edge (non diagonal positions) or to a vertex (diagonal positions). This process is expensive and when the geometry and the topology of mesh is changed, the matrix of coefficients needs to be also changed.

Hence, we realized that if we could have the information concerning the elements around an edge, we could easily collect and build the information of the non diagonal positions of the linear system. Moreover, if we store the computed information for every edge (elements around the edge) within a small *edge matrix*, we will have enough flexibility to modify or recompute only the incident edges to a vertex, when the vertex changes, without affecting the rest of the linear system. Analogically, we can perform the same strategy for storing the computed information of the diagonal positions of the linear system, computing a small *diagonal matrix* for every unknown. These two sets of matrices are an equivalent representation of the matrix of coefficients, which we will refer to as the *equivalent matrix*.

Structure wise, we no longer use the traditional sparse matrix ([SGV05]) to store the matrix of coefficients. Instead, we have divided it into two vectors (see Figure 1 for a visual representation). In one vector we store the *edge matrices* and on the other we store the *diagonal matrices*. Every element of both vectors is a *nxn* matrix where *n* is the dimension (1D, 2D or 3D) of the problem.

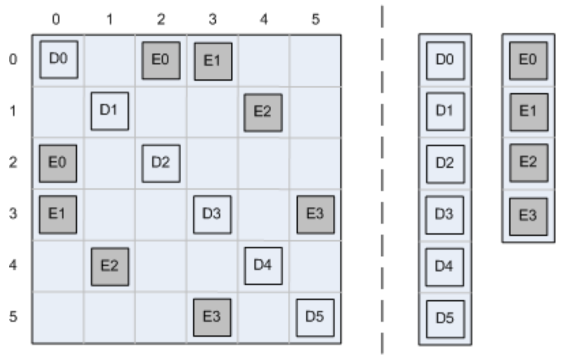

Figure 1: *Graphic representation of a matrix of coefficients (left). The same matrix of coefficient represented as the equivalent matrix (left).*

The two main characteristics of a matrix of coefficients, sparsity and symmetry, are used by the *equivalent matrix* to minimize memory consumption: only the nonzero values are stored and only one instance of every edge is stored for every pair of connected vertices. In order to explain how our algorithm uses this characteristics for building the *equivalent matrix* and for the sake of clarity, we have subdivided the process into three steps:

1. Constructing the needed neighboring information
2. Computing the set of *edge matrices*
3. Computing the set of *diagonal matrices*

These three simple steps enable the minimization of the space in memory and the reduction of the processing time. In addition, the new representation of the matrix of coefficients allows the flexible handling and modification of individual vertices or edges, without affecting the rest of the matrix. The example mesh shown in Figure 2 will be used in association with Figure 3 and Figure 4 to accompain the explanations given in 'Computing the edge matrices' and 'Computing the diagonal matrices' respectively.

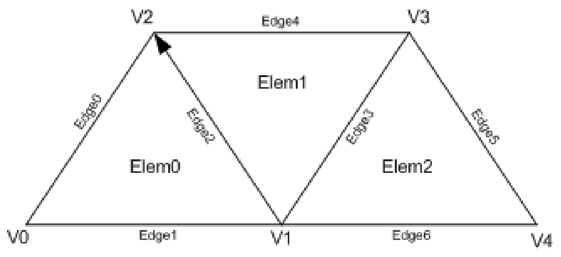

Figure 2: *A 2D mesh composed of three elements.*

3.1. Build of the equivalent matrix

The *equivalent matrix* replaces the matrix of coefficients by a set of small matrices, which can be computed faster and which requires less space in memory. In order to avoid traversing the whole mesh, when computing the matrix of coefficients, we precompute the neighboring information. We also precompute the *element matrices*, which are the basis for computing the edge and diagonal matrices. Normally, the *element matrices* are not computed, since the contribution of the elements is directly considered during the solution of the system. However, we aim at effectively handling geometrical or topological changes, therefore it is more efficient to use the *element matrices* for updating the *edge matrices* or the *diagonal matrices* according to the changes, than recomputing the needed *element matrices* every time.

Constructing the neighboring information

We need to construct three kinds of neighboring information: i) elements around an edge, ii) elements around an unknown and iii) vertices of an edge. This information is computed during the initialization process and it is updated, if some changes to the topology of the mesh are made. The neighboring information allows us computing the non diagonal and the diagonal positions of the matrix of coefficients independently and it also provides us with the information regarding the relationship between vertices. We are using a mesh data structure, which automatically constructs the neighboring information, however we will explain this process for the sake of completeness.

During the loading process of the mesh, we initialize three double arrays (db), where we store the needed information for the edges ($dbEdg$), for the unknowns ($dbUkn$) and for the vertices of the edges ($dbVtsEdg$). For every read element, we append its index to its six corresponding edges in $dbEdg$ and to its four corresponding vertices (unknowns) in $dbUkn$, and we also add the corresponding pair of vertices to every one of the six edges within $dbVtsEdg$. By the end of the loading process, the neighboring information is also ready.

Computing the edge matrices

Given the neighboring information of the elements around an edge ($dbEdg$) and the element matrices, we can easily compute the non diagonal positions, by means of traversing the elements around the edge and adding the contribution of the corresponding element. On Figure 3 it is shown the computations involved in the build of Edge2. Since both elements Elem0 and Elem1 share Edge2, the components of both elements regarding this edge (colored in grey) are added to the Edge2's matrix. Note that each element has two contributions to every used edge and since one is the transposed of the other, only one is added to the edge matrix. The used contribution is chosen based on the direction of the edge.

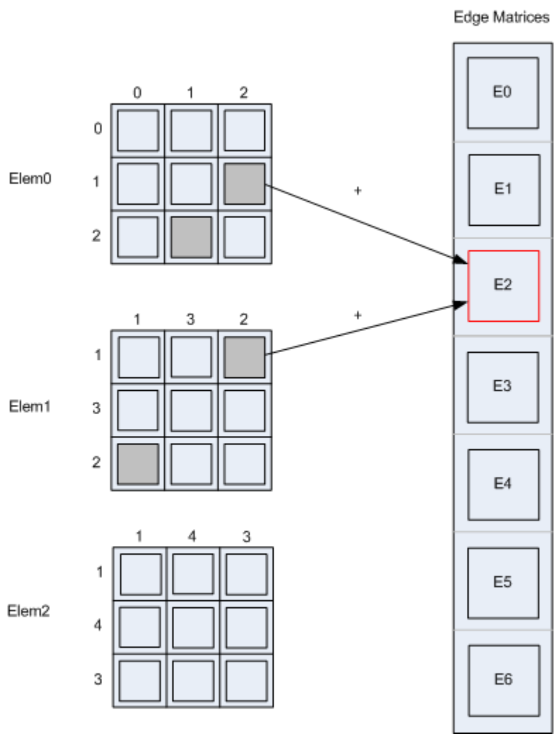

Figure 3: *Element contributions to the build of the third edge matrix (E2).*

Computing the diagonal matrices

In a similar way, we compute the diagonal positions of the linear system. In this case, we use the neighboring information regarding the elements around an unknown ($dbUkn$) and we consider the contribution of every involved element to the diagonal. Consider Figure 4, where the build of the diagonal for vertex V1 is being performed. From Figure 2 it is clear that vertex V1 is shared by the three elements. Therefore, D1 is calculated by adding up the three contributions to V1 of the three elements.

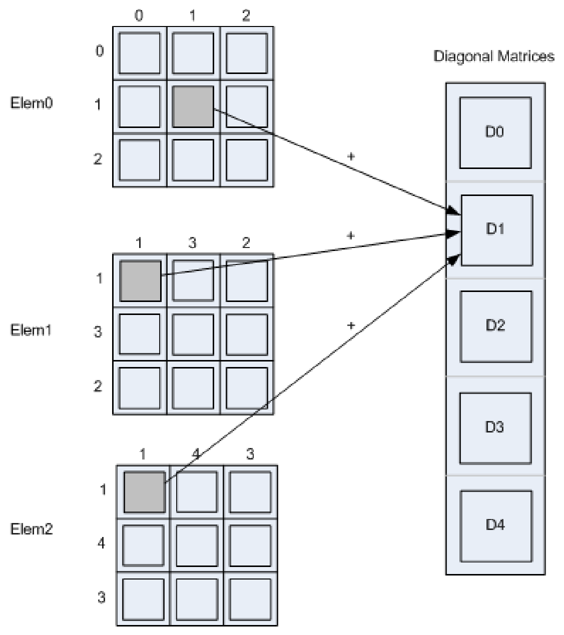

Figure 4: *Element contributions to the build of the second diagonal matrix (D1).*

The union of the *edge matrices* and the *diagonal matrices* is equivalent to a matrix of coefficients. Although, we need to consider these two sets of matrices, in order to solve the system, we can arbitrary decide the order in which we want to solve it. This flexibility could be advantageous for a rapid convergence, since it is equivalent to having a linear system with an optimized ordering of the unknowns, leading to an improvement of the performance of the solver ([OLHB02, Bar96, BW98]). Moreover, since the matrices within the two sets are independent, we can change or update them without a major effort, because we would only need to recompute a small set of matrices, avoiding the computation of the whole set of equations.

3.2. Update of the equivalent system

Every change that is done to the topology of the mesh implies an update to the matrix of coefficients so that the latter continues to represent the changed mesh. For the sake of functionality, the implemented update method allows the processing of more than one change to the mesh per call. Our method's procedure is based on adding or removing the contribution of the elements that were affected by the change. The arrays of *edge matrices* and *diagonal matrices* effectively support the addition and removal of vertices or elements. The arrays are implemented with a buffer according to the the size of the mesh, enabling addition of new vertices or elements. In case of removal, the corresponding matrices are flagged as "removed", but there are no memory reallocations, in order to avoid this expensive task. The double arrays for the neighboring information have the same capabilities. Algorithm 1 shows the steps taken to perform the update:

```
1  foreach removed element do
2      remove element contribution
3  end
4  reserve space for new elements
5  foreach added element do
6      compute element contribution
7      add element contribution
8  end
9  foreach moved node do
10     foreach element shared by this node do
11         add element to recomputeVector
12     end
13 end
14 foreach element in recomputeVector do
15     remove element contribution
16     recompute element contribution
17     add element contribution
18 end
```

Algorithm 1: Method to update the equivalent matrix

The algorithm can be subdivided into three phases, representing the three possible kinds of changes, which can be applied to the topology of the mesh:

1. Remove an element (line 1 to 3) - The contribution of each removed element is subtracted from the corresponding entries in the equivalent matrix, according to the neighboring information (no memory reallocation is performed);
2. Add an element (line 4 to 8) - Additional space is reserved (the additional space is obtained from the available buffer) to incorporate the new elements. The contribution of each added element is computed and added to the equivalent matrix (to the corresponding entries according to the neighboring information);
3. Move a vertex (line 9 to 18) - To avoid recomputing the contribution of an affected element more than once, an initial loop is done to find out which elements are affected by the movement of the vertices. For each affected element, its contribution is removed from the equivalent matrix. Then, its contribution is recomputed using the new vertices positions and finally it is added back to the equivalent matrix.

Note that adding or removing an element contribution to the equivalent matrix is done by iterating over the edges and diagonals that are shared by that element. If a vertex is removed, no special operation needs to be considered, since it implies the elimination (flagged as "removed") of the corresponding diagonal matrix and the edges matrices incident to that vertex. The neighboring information is also updated, therefore the following operations will not involve the "removed" vertices.

3.3. Solution of the equivalent system

The implemented algorithm to solve the linear system of equations is the Conjugate Gradient as it was proposed in [She94]. The only noticeable change done so far is the way we multiply the equivalent matrix with a vector. Although, we have chosen the Conjugate Gradient method for solving the linear system, our methodology is completely independent of the kind of solver. Our algorithm is suitable for other iterative methods or even direct methods.

Multiplication of the equivalent matrix with a vector

Having the matrix of coefficients stored in the *equivalent matrix* form requires a special method for its multiplication with a vector.

```
1  foreach rst in resultVector do
2      resultVector[rst] = 0
3  end
4  foreach edge in edgeVector do
5      edgeStartVertex = start vertex of this edge
6      edgeEndVertex = end vertex of this edge
7      resultVector += (edgeVector[edge] * multiVector)
8      resultVector += (edgeVector[edge]^T * multiVector)
9  end
10 foreach diag in diagVector do
11     resultVector += (diagVector[diag] * multiVector)
12 end
```

Algorithm 2: Multiplication of the equivalent matrix with a vector (multiVector) and its result (resultVector).

Algorithm 2 shows the main steps we perform to multiply the equivalent matrix with a vector. We start by setting the *resultVector* to zero so that the results of the multiplications performed over the matrix can be added to it. For every edge, we find the vertices that form that edge (edgeStartVertex and edgeEndVertex) by consulting the neighborhood information (*dbVtsEdg*). This is done to know which is this edge position in the matrix. Doing so it is known with which position of *multiVector* this edge should be multiplied and in what position of *resultVector* it should be stored.

Also notice that for every edge two multiplications are done. This is due to the symmetric characteristic of the matrix. To provide a better understanding of this procedure consider the example shown in Figure 5. Assuming that we are iterating on edge E1 and that this edge has vertex 0 as a starting vertex and vertex 3 as an ending vertex. In line 7 of the algorithm we would multiply E1 by V3 and store its result in R0 (red boxes). And since E1 also connect vertex 3 to vertex 0 with the same value, in line 8 we store in R3 the multiplication of the transposed E1 with V0 (green boxes).

After processing all the edges, we iterate on the diagonals. The process is similar but simpler, for each diagonal relates a vertex to itself.

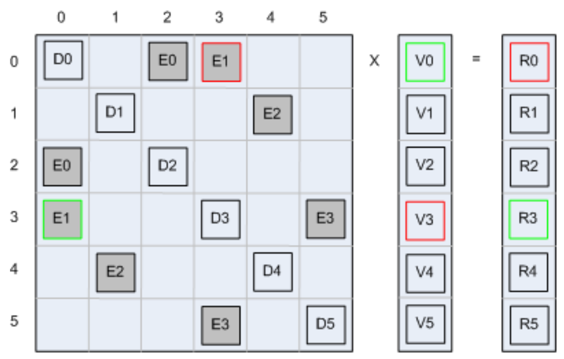

Figure 5: *Multiplication of the equivalent matrix (represented as a normal matrix for simplification) with a vector. The green and red line colored boxes indicate the used values when the multiplication iterates on edge E1.*

4. Results

We have implemented the build of the *equivalent matrix* (*DEM*) and its multiplication with a vector , as proposed in the previous section. This implementation is not complex and it can easily be reproduced following the given indications. Our implementation is single threading and a desktop PC Intel Core 2 Quad Q6600 with 3.25 GB RAM was used to make the measurements. In order to compare the performance of our algorithm, we have also implemented the classical form to represent the matrix of coefficients, the sparse matrix. We programmed four implementations of the sparse matrix: i) linked lists without the symmetric characteristic (*CSM*), ii) linked lists with the symmetric characteristic (*SSM*), iii) compressed row storage (*CRS*) and iv) block compressed row storage (*BCRS*).

The first two implementations (*CSM* and *SSM*) were only made for the sake of completeness, hence a simple array of linked lists was employed. The last two implementations (*CRS* and *BCRS*) were done without considering the symmetric characteristic, since we wanted faster access during the multiplication of the matrix with a vector (SPMV). The implementation of the multiplication with a vector was made for the four representations as well. For the build process of *CSM*, *SSM*, *CRS*, and *BCRS*, we have used the neighboring information, in other words, we store the needed information in the representation without traversing the whole mesh. For the multiplication process, the neighboring information was not used for *CSM*, *SSM*, *CRS* and *BCRS*, since each one has fields for identifying the corresponding position in the mesh.

In order to measure the performance of the five implementations, we have used two different tetrahedral meshes: i) a gargoyle with almost 50.000 elements and ii) a hand with almost 100.000 elements. Table 1 presents the relevant topological information of the meshes (shown in Figure 6). In order to present the capabilities of the update process, we

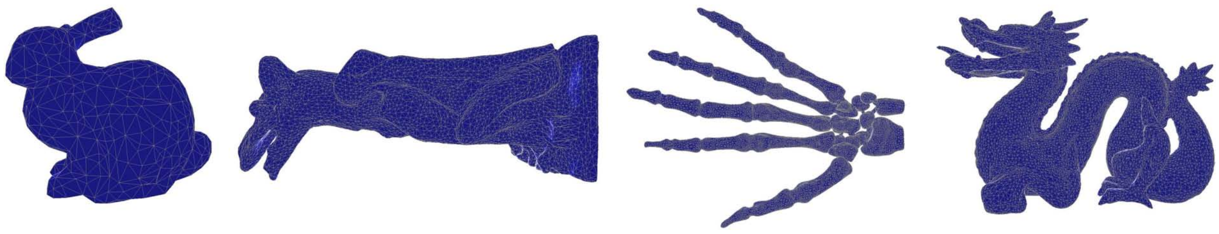

Figure 6: *Mesh models for the measurement.*

have chosen three operations representing the basic changes in the mesh (remove an element, add an element and move a vertex): i) decimate, ii) mirror and iii) scale. The decimate operation takes a mesh an remove 50% of the elements of the model. The mirror operation doubles the number of elements of the mesh. The scale operation moves every vertex of the mesh. For the test of these operations, we have used the dragon, the gargoyle and the bunny (Figure 7 shows the three forementioned operations, one on each model).

Table 1: *Topological information of the meshes for the measurement.*

Mesh	Vertices	Edges	Elements
Bunny	2.087	12.796	9.997
Gargoyle	13.044	71.873	49.996
Hand	26.649	144.669	99.995
Dragon	26.436	144.285	100.000

We have made two kinds of measurements, one for the build process and one for the multiplication process. We have considered 20 multiplications, in order to be able to measure the time, since the measurement for a single multiplication is not very accurate. Table 2 shows the results in milliseconds for the mesh models and the two processes. These measurements were made for the five representations (*CSM*, *SSM*, *CRS*, *BCRS* and *DEM*) of the linear system.

Table 2: *Measurements for the build and multiplication processes (in milliseconds).*

Process	Build		Multiplication	
Mesh	Gargoyle	Hand	Gargoyle	Hand
CSM	203	456	213	598
SSM	115	265	140	347
CRS	459	901	313	691
BCRS	203	420	83	225
DEM	94	215	78	179

These results show, that our algorithm is faster than the other four implementations. Our algorithm is for the build process between 49% and 80% faster than the *CSM CRS* and *BCRS* implementations, since these implementations require almost two times the space in memory than the *SSM* and ours. However, for the multiplication process, the *CRS* and *BCRS* implementations will have an advantage, because of the direct access, but this is not the case for the *CSM*, where the access is more expensive (because of the linked lists). Although the build process for the *SSM* implementation is similar to ours, we are still 18% faster.

In the multiplication process our algorithm performs between 44% and 75% faster than the *CSM*, *SSM* and *CRS* implementations. The reason for these results is the expensive access for the linked lists (*CSM* and *SSM*). The *CRS* implementation is slower than the *CBRS* implementation, because it also considers the nonzero entries of the *nxn* element matrices (*n* is the dimension of the problem). On the other hand, the *BCRS* implementation considers the *nxn* element matrices as a block, improving the performance during the multiplication process. *DEM* is in this case only 6% faster for the Gargoyle and 20% faster for the Hand. These results also show, that the *BCRS* algorithm does not present a proportional behavior to the number of elements, but our does.

Although the *BCRS* algorithm could be an interesting alternative for the multiplication process, it will be useless for topological changes, since it will require the addition and removal of block entries and therefore memory reallocation in the arrays (the memory is continuous), a special characteristic that *DEM* can handle very well. In terms of memory consumption, the *CSM*, *CRS* and *BCRS* implementations require more memory than the *SSM* and the *DEM* implementations, since these do not profit from the symmetric characteristic of the matrix of coefficients. Only the upper or the lower part of the matrix of coefficients are stored in the *SSM* and the *DEM* implementations, hence they have the same memory consumption.

The primary aim of our implementation is to effectively support geometrical and topological modification of mesh-based techniques, requiring the utilization of linear systems. Nevertheless, we have also explored the limits of our implementation in terms of real time performance. Table 3 presents the measurements for 20 iterations (not only the multiplication) of the conjugate gradient method with a Jacobi preconditioner for three meshes with different sizes.

Table 3: *Meshes with real time performance (time in milliseconds).*

Mesh	Vertices	Elements	Time	FPS
Bunny_10	2.087	9997	16	63
Gargoyle_30	7.944	29.998	47	21
Gargoyle_50	13.044	49.996	99	10

These results reveal that our algorithm can perform in real time with meshes up to 30.000 elements and at interactive rates with meshes up to 50.000 elements. As mentioned above, our algorithms aims at effectively handling geometrical and topological changes. Because of that, we store the pre-computation of the element matrices, in order to easily update the equivalent system, whenever a change in the topology (remove or add vertices and elements) or in the geometry (move vertices) happen. The three implemented operations (decimate, mirror and scale) are extreme examples, since they employ at least the 50% of the elements or the vertices and this is not a common activity in computer graphics, where fast performance is expected.

The decimate operation removes half of the elements of the model, hence only the elements, which are on the boundary of the removal are recomputed (see table 4). Since the elements on the boundary of the removal are much less than half of the model, the update process is much faster than the reconstruction.

Table 4: *Measurements for the decimate operation (in milliseconds).*

Mesh	Initialization	Update	Reconstruction
Bunny	140	< 1	67
Gargoyle	702	71	359
Dragon	1.427	47	719

The update process for the mirror operation takes approximately 50% of the time of the reconstruction (see table 5), because it only processes the mirrored elements in comparison with the reconstruction, which processes two times the number of elements (the original and the mirrored ones).

The update process for the scale operation takes slightly more time than the reconstruction (see table 6), however moving the complete set of vertices in a single step is the worst case scenario, thus any other operation involving moving vertices will perform much faster than the reconstruction.

The presented three operations show the performance of our algorithm for geometrical and topological changes. We

Table 5: *Measurements for the mirror operation (in milliseconds).*

Mesh	Initialization	Update	Reconstruction
Bunny	140	125	276
Gargoyle	702	783	1.416
Dragon	1.427	1.380	2.857

Table 6: *Measurements for the scale operation (in milliseconds).*

Mesh	Initialization	Update	Reconstruction
Bunny	140	140	141
Gargoyle	702	736	703
Dragon	1.427	1.483	1.427

demonstrated that our algorithm is in normal conditions (no extreme examples) much faster than a complete reconstruction of the linear system. This is an improvement towards mesh-based applications such as simulation, shape deformation, virtual surgery, and fluid/smoke animation, among others, where geometrical (and sometimes topological) changes affect the linear system, which needs to be solved. These kinds of applications will not only benefit from faster updates, but also from faster solutions, as demonstrated with the multiplication process.

Hence, our methodology enables interactive rates for dynamic meshes with up to 50.000 elements in a single thread. From the previous results, we have also realized that the proposed algorithms behave proportional to the size of the meshes. In other words, the complexity of the algorithms is $O(n)$.

5. Conclusions and future work

We have presented algorithms, which use the neighboring information of the mesh to effectively build, update and solve linear systems. Our algorithms avoid the assembly of the matrix of coefficients and they also reduce the processing time and the memory consumption. The proposed methodology also enables the handling of the set of equations with more flexibility, allowing their iteration and solution in any arbitrary order. We have started the exploration of this aspect and we are evaluating different alternatives, for example: reverse Cuthill-McKee (RCM), self-avoiding walk (SAW), out-in ordering (ordering from the boundary to the interior) or multi layer solving. The last three options are based on the neighboring information of the mesh, a characteristic, which we have also exploited in this paper. This flexibility is a step forward towards the simulation of meshes with dynamic topology or dynamic meshes. We will further implement and refine our algorithms, we want to explore other

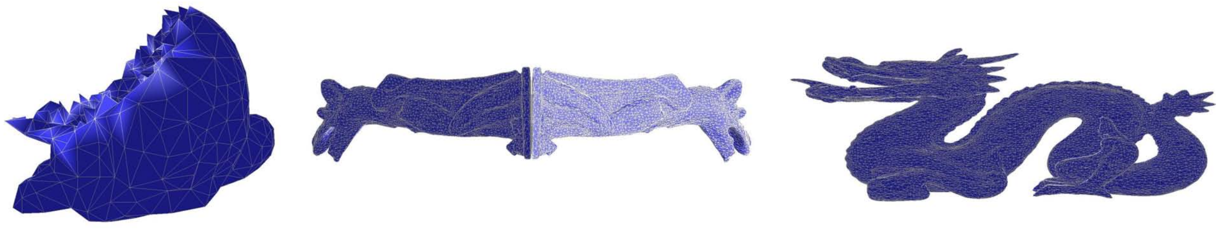

Figure 7: *From left to right: decimated bunny, mirrored gargoyle and scaled dragon.*

iterative solvers and we have already strated with a direct solver (Cholesky factorization). We want also to investigate parallelization schemes, either for the CPU or the GPU. We also plan to develop a framework, where the modification of meshes and its real time simulation will be feasible, aiming at integrating design and analysis of mechanical objects within the same environment.

6. Aknowledgement

This work is partially supported by the European projects 3D-COFORM (FP7-ICT-2007.4.3-231809) and FOKUS K3D (FP7-ICT-2007-214993).

References

[ACF*06] Alexa M., Cani M.-P., Frisken S., Singh K., Schkolne S., Zorin D.: Interactive Shape Editing: SIGGRAPH 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (2006), ACM New York, NY, USA. 2

[ARS06] Augarde C., Ramage A., Staudacher J.: An element-based displacement preconditioner for linear elasticity problems. *Computers and structures 84*, 31-32 (2006), 2306–2315. 2

[Bar96] Baraff D.: Linear-time dynamics using lagrange multipliers. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM, pp. 137–146. 5

[BFMF06] Bridson R., Fedkiw R., Muller-Fischer M.: Fluid simulation: Siggraph 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, pp. 1–87. 2

[BN96] Bro-Nielsen M.: Surgery simulation using fast finite elements. In *VBC '96: Proceedings of the 4th International Conference on Visualization in Biomedical Computing* (London, UK, 1996), Springer-Verlag, pp. 529–534. 2

[BNC96] Bro-Nielsen M., Cotin S.: Real-time volumetric deformable models for surgery simulation using finite elements and condensation. In *Computer Graphics Forum* (1996), pp. 57–66. 2

[Bow09] Bower A.: *Applied Mechanics of Solids*, 1 ed. Taylor and Francis, August 2009. 2

[BW98] Baraff D., Witkin A.: Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM New York, NY, USA, pp. 43–54. 5

[GBT07] Gissler M., Becker M., Teschner M.: Constraint sets for topology-changing finite element models. In *Virtual Reality Interactions and Physical Simulations VRIPHYS* (November 9 2007), pp. 21–26. 2

[Hug87] Hughes T.: *The finite element method: linear static and dynamic finite element analysis*. Prentice-Hall Englewood Cliffs, NJ, 1987. 2

[KFCO06] Klingner B. M., Feldman B. E., Chentanez N., O'Brien J. F.: Fluid Animation with Dynamic Meshes. In *International Conference on Computer Graphics and Interactive Techniques* (2006), ACM New York, NY, USA. 2

[Lan03] Langtangen H.: *Computational partial differential equations: numerical methods and diffpack programming*. Springer Berlin, 2003. 2

[MSJT08] Müller M., Stam J., James D., Thürey N.: Real time physics: class notes. In *International Conference on Computer Graphics and Interactive Techniques* (2008), ACM New York, NY, USA. 2

[MTPS08] Mezger J., Thomaszewski B., Pabst S., Strasser W.: Interactive Physically-Based Shape Editing. In *ACM Solid and Physical Modeling Symposium (SPM)* (2008). 3

[NMK*06] Nealen A., Müller M., Keiser R., Boxerman E., Carlon M.: Physically Based Deformable Models in Computer Graphics. *Computer Graphics Forum 25*, 4 (2006), 809–836. 2

[OLHB02] Oliker L., Li X., Husbands P., Biswas R.: Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review 44*, 3 (2002), 373–393. 5

[SG04] Smith I., Griffiths D.: *Programming the finite element method*. Wiley, 2004. 2

[SGV05] Smailbegovic F., Gaydadjiev G. N., Vassiliadis S.: Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2005* (November 2005), pp. 445–448. 3

[She94] Shewchuk J.: An introduction to the conjugate gradient method without the agonizing pain. *Computer Science Tech. Report* (1994), 94–125. 2, 6

[SVDV00] Saad Y., Van Der Vorst H.: Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics 123*, 1-2 (2000), 1–33. 2

[vFTS06] von Funck W., Theisel H., Seidel H. P.: Vector Field Based Shape Deformations. In *International Conference on Computer Graphics and Interactive Techniques* (2006), ACM New York, NY, USA. 2