Efficient Collision Detection and Physics-based Deformation for Haptic Simulation with Local Spherical Hash

Marilena Maule, Anderson Maciel and Luciana Nedel
Instituto de Informática - INF
Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
email: {mmaule, amaciel, nedel}@inf.ufrgs.br

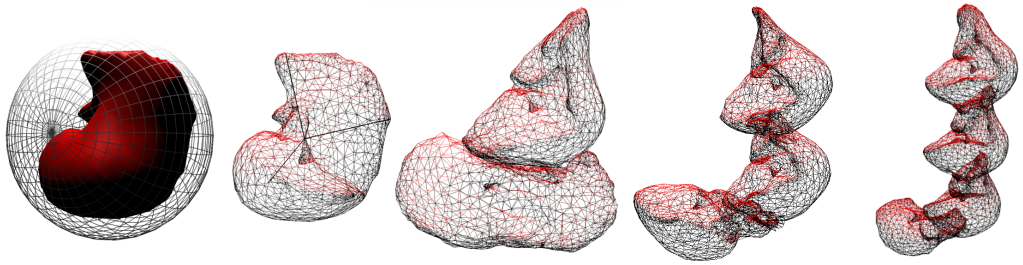

Figure 1. A spherical hash approach is used to handle collisions between complex deformable models in constant time to the number of vertices. The first image shows a tetrahedral mesh encapsulated into its spherical bounding. The other images present frames of real-time animations involving one, two, three and four colliding deformable objects. Since our case study is on surgical simulation, we are using a reconstructed and decimated real liver to exemplify.

***Abstract*—While real time computer graphics rely on a frame rate of 30 iterations per second to fool the eye and render smooth motion transitions, computer haptics deals with the sense of touch, which requires a higher rate of around 1kHz to avoid discontinuities. The use of haptics on interactive applications as surgical simulations or games, for instance, can highly improve the user experience, and the sense of presence. However, its use in complex simulations involving realistic rendering, deformable objects and collision detection requires the development of very performing algorithms. This paper presents an implementation of mass-spring system adapted to CUDA implementation, and a novel method for collision detection in haptic update rate. Important aspects of the port to parallel programming and the GPU architecture are addressed, as for example, strategy and frequency of memory access. A quantitative experiment is also presented to evaluate our methods capability and scalability.**

***Keywords*-GPGPU; physics-based animation; medical simulation; computer animation**

I. Introduction

Haptic interactions are often suitable in computer graphics systems to amplify the user interaction experience. However, the high update rates required demand a great amount of the computational power. While the display update rate needed to fool the eye with smooth transitions rely on a frame rate of 30 Hz, computer haptics deals with the sense of touch, which requires a higher rate of around 1kHz to avoid discontinuities.

Another desirable feature in many graphics applications is physics-based simulation. From games to surgery simulation, physics has become a must have. However, realistic physics-based models are often very complex and also require high computational power. When deformations are involved, even simple physical models, as mass-spring systems, present a too costly numerical integration to run in real time.

The combination of fast deformable physics-based models with haptic interactions is still a challenge. In the present work we highlight two major bottlenecks for such simulations: collision detection and deformation computation. This work explores parallel hardware acceleration and efficient collision detection approaches to solve the problem of simulating several deformable bodies in an interactive environment with haptics. Experiments which may lead to an effective bimanual surgical simulator with haptics have been performed.

Our parallelization approach is based on the general purpose GPU programming, which is no longer new. Many researchers explored the parallelization capabilities of graphics hardware to compute deformation and other procedures using shader languages [1][2]. The greatest challenges then, were how to model the problem in such a restrictive environment where GP (general purpose) data had to be converted into graphics structures like texture memory, vertices, color information, and so on, and then converted back.

More recently, CUDA architecture made the problem easier by providing programming mechanisms of higher-level

languages for GPGPU programming. However, GPGPU is still bounded by the limitations of the general hardware architecture. Such limitations must be in mind in the algorithm project phase to take the best advantage of the parallel power. An important detail that is highlighted in this paper is the data organization, since the data access pattern impacts deeply in the final performance.

In this context, and focusing on the problem of interactive physics-based simulation with haptics, this work proposes an integrated schema for the parallel implementation of a simulator. The major contribution is an efficient method for collision detection with deformable bodies inspired on [3], and adapted to run on the GPU. We focus on data organization to run efficiently in the GPU paradigm.

In the next section we review the literature of physics-based deformation in graphics, collision detection, and haptic interactions. In Section III we present how the deformation problem is modeled to run in a parallel CPU implementation using OpenMP, and in GPU, using CUDA. Section IV describes the implementation of the local spherical hash algorithm for collision detection, and Section V details the approach and the devices used to render haptics and how the feedback forces are calculated. Finally, Section VI presents the experiments leaded and our results, and Section VII the conclusions and future work.

II. Related Work

A. Efficient mass-spring systems

A mass-spring system is a physically-based and less expensive technique used to represent deformable objects in interactive applications where physical accuracy is not mandatory. Over the last decades, they have been applied in real-time. Generically, the dynamic of a mass-spring system is based on time-dependent formulae (*force-laws*). These forces can be physically-based, such as linear springs or dampers, or designed to hold some desired configuration.

Recently, some works [4][5]took advantage on both GPU performance and parallelism, which usually fits in the simulation needs – simple procedures performed over a lot of data (streaming processing model) – to implement physically-based simulations. This model fits in mass-spring system requirements, and has been addressed by researches.

The first proposal of a GPU-based mass-spring system, to our knowledge, was done in [6]. The authors perform deformations on 3D rectilinear meshes extracted from medical datasets. All particles are represented in a 2D texture, and the topology is encoded using texture coordinates. Thus, each vertex is always connected to its 18 neighbors, which can be a significant limitation in simulation of more general systems (with small valences). Explicit Verlet integration is used to integrate the model dynamics.

Georgii et al. [7] discusses a mass-spring system that allows deformations to tetrahedral meshes. Particles are also stored in a 2D texture (here called *vertex texture*); however, the mesh topology is represented by a stack of vertex textures, where each texture encodes one of the tetrahedral neighbors adjacent to the respective vertex in the vertex texture. The authors also discuss a technique (valence textures) which avoids the memory overhead required to represent vertices with high valences. With this representation, they reach interactive rates through a GPU-based computation of Verlet integration method. However, the memory overhead introduced by storing each neighboring vertex of the tetrahedra separately and the multiple spring forces calculation are severe drawbacks in their proposal.

Tejada et al. [8] present a GPU-based system which both performs physically-based deformation and volume visualization of tetrahedral meshes on recent GPUs. The authors propose a GPU-based implementation of implicit solvers, in order to reach *stable* simulations with larger time-steps. The model topology is implemented through two textures, which encode the vertex neighborhood, and additional textures to hold the vertex positions, velocities and external forces.

B. Collision Detection

The problem of collision detection came into the computer simulation problematic because, as their real world counterparts, simulated objects are also expected not to penetrate each other. The problem involves checking the geometry of the target objects for interpenetration, which is usually made using static interference tests. We refer to [9], [10] and [11] for detailed surveys on general collision detection.

Following the point approach, a hybrid approach based on uniform grids using a hash table and OBB-trees is presented in [12] and provides a method for fast collision detection between the haptic probe and a complex virtual environment. OBB-trees are also used in [13] to reduce the problem to local searches. Though they provide only point-based haptic rendering, intersections are detected against a line segment defined by the moving path of the haptic interaction point.

Image-based methods for collision detection use the graphics hardware to detect collisions. As they are based on the image space they are not bound by the size of the meshes being tested but by the size of the image rendered. For that reason, they can be used to detect collisions between deformable objects. However, as these methods are based on a discrete approximated representation of the objects – an image – their precision depends on the errors of this discretization, which represents a limitation as they provide limited information for collision response. In [14] every object is rendered in 2 depth buffers containing the least and the greatest depth value for every pixel of the object. The range of values represents the object in each pixel and can be used to detect contacts between convex objects. In [15], the authors use an image-based method to detect collisions in line-like interactions with deformable objects. In a related paper [16] they also discuss issues such as those pertaining

to the use of a discrete projection of the line on the mesh in the context of a collision detection and response algorithm.

Some hash table-based methods have also been proposed. In [3] a spherical hash is used to detect collisions. The method evaluates in constant time the distance between a point in one mesh and the closest point of another mesh. However, it limits the mobility and the deformation of the object around which the hash table is built due to the coordinate system being fixed. More details about this method are given in section IV where an extension is proposed.

Interactive applications, especially in computer graphics, require efficient collision detection methods to render real time graphics. This problem is solved in most cases using traditional collision detection methods with the state of the art hardware. However, while in real time computer graphics *interactivity* is limited to a display rate of 30 frames per second, in multimodal virtual environments involving haptic interactions, a much higher update rate of about 1 kHz is necessary to ensure continuous interactions and smooth transitions [17]. Due to the presence of deformations, most of the common assumptions upon which the methods cited here are ineffective. Additionally, thorough methods are inefficient due to the associated computational complexity.

III. Fast deformable models

GPGPU has been used to accelerate the execution of algorithms in many domains, including the simulation of deformable bodies. Combining the efficiency of mass-spring systems with the power offered by the parallelization with GPU programming, it is possible to quickly calculate the numerous small steps required to explicitly integrate forces into displacements in a physical simulation. For this, the data structures used should be adapted to take advantage of the GPU memory organization and access politics.

A. Problem Statement

We want to simulate soft body deformation using a mass-spring system. The deformable object is modeled as a tetrahedral mesh, where each vertex corresponds to a particle and each edge is a spring connecting two particles. Each system step is achieved by evaluating the Euler integration of the parameters of each mass m_i, as shown in Equations 1, 2 and 3. The new position of the mass (x_{i+1}) is given by the older position (x_i) added to its velocity (v_{i+1}), where (v_{i+1}) comes from the previous velocity (v_i), which is damped $(1-d)$ to simulate energy loss (caused by friction) plus the mass acceleration (a_i) in the time interval Δt. The acceleration (a_i) is the addiction of the gravity acceleration g with the pondered spring forces (F_{ij}) acting in the mass m_i.

$$x_{i+1} = x_i + v_{i+1} \tag{1}$$

$$v_{i+1} = v_i(1-d) + a_i\Delta t \tag{2}$$

$$a_i = \sum F_{ij}/m_i + g \tag{3}$$

These parameters compose the dataset to be stored in order to integrate the system. Assuming that all particles have $m_i = 1$, we will need to save positions (x_i), velocities (v_i), as well as the springs rest and instantaneous lenghts for each mass m_i.

B. CPU Implementation

Each CPU core takes one particle at time, evaluates it, and takes the next. Even with multiple cores, many architectures reserve some cache to each core. So, to take advantage of the cache locality, it is convenient to organize the data in an array of structures, each structure corresponding to a particle as shown in Figure 2, so all the data needed to evaluate that particle will be at hand.

Figure 2. Data structure for mass-spring system implementation in sequential processors. The organization in an array of structures allows cache locality, providing the data needed to serial computing.

C. GPU Implementation

The mapping of the system parameters to structured data in GPU memory must be done carefully. The memory access pattern is different in the GPU and the memory hierarchy is designed to hide the delay from the massive data fetch, keeping the processing units busy all the time. All the threads will run the same code at the same time. They will first access their respective positions, then their velocities, and so on. So, if we use the same organization, with arrays of structures, all parameters for one thread will be brought to cache at the same time when the thread only needs the position, which may generate lack of cache to other threads. However, if instead of using an array of structures, we use a structure of arrays, as shown in Figure 3, when all the threads need the positions, only the positions will be in the cache, the same occurring to the other parameters.

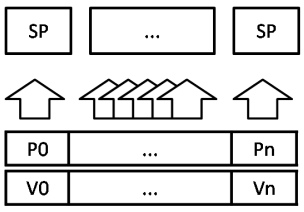

Figure 3. Data structure for mass-spring system implementation in GPU. The organization in a struct of arrays allows cache locality to the many threads running concurrently, providing the data needed to parallel computing.

The springs lengths are indirectly stored using an adjacency list which indexes each particle j connected with the particle i. The neighborhood is stored in two vectors. The *NeighborsIndexes* vector contains all the adjacency lists sequentially. The *Bases* vector contains the indices to the first elements of each list, as shown in Figure 4. To compute the forces acting on the particle i we sum the contributions from all its neighbors by traversing the *NeighborsIndexes*, from de index given from the i^{th} entry of the *Base* vector to the index given by the $i+1^{th}$ entry of the same vector. Thus, the *Base* vector needs $n+1$ entries, where n is the number of particles and the $n+1^{th}$ entry stores the size of the *NeighborsIndexes* vector, which is the total of springs.

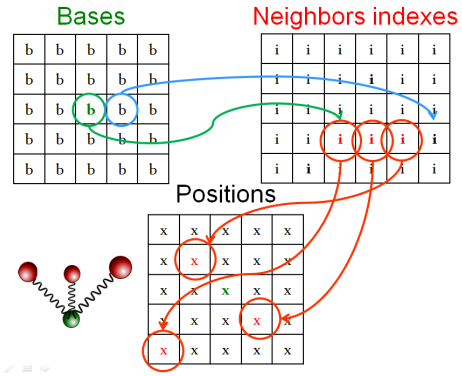

Figure 4. Data structures for mass-spring connectivity implementation in GPU. Each index in the *Base* vector points to the lists in the *NeighborsIndexes* vector which indexes the connected particles.

IV. Fast collision detection with Local Spherical Hash

In this section we introduce the *Local Spherical Hash* (LSH) method, inspired on the *Spherical Sliding* method – presented in [3] – for collision detection involving deformable bodies. The *Spherical Sliding* method builds a hash table – called Spherical Sampling Table (SST) – where each cell is indexed by the angles θ and ϕ in a spherical coordinates schema.

One of the meshes is considered fixed because its center is at the origin of the spherical coordinates system, and its triangles will be stored in the SST. To build the SST, a ray is traced from the origin to the cell, oriented by polar angles, and the cell is filled with the triangles intercepted by the ray. When computing collision detection, for each vertex from the mobile mesh, the nearest triangle from the fixed mesh is obtained in constant time.

Our method builds a similar hash table in preprocessing time for n meshes. The initialization is made with the tetrahedral mesh in its rest state, before the deformation starts. The first step is to build a bounding box around the mesh. The vertex closest to the center of the bounding box is taken to be the origin of the local reference system. The entire mesh is then translated in order to put this central vertex in the origin of the global reference system. To build the axes of the local frame, we choose the three vertices on the mesh surface closest to the x, y and z axes of the global reference system. Algorithm 1 describes this procedure in high level, and Figure 5 shows an object with its spherical bounding and deformable local coordinate system.

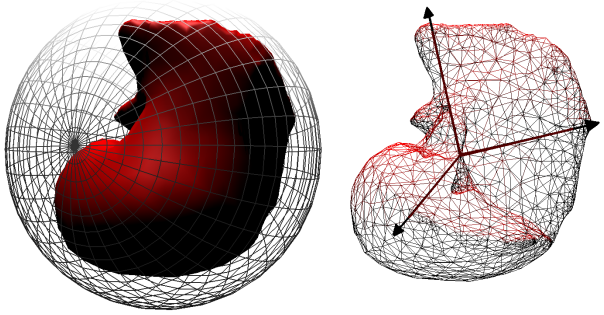

Figure 5. Spherical Bounding (left) encapsulates the model and provides a fast first collision test. Deformable Local Coordinate System (right) allows the correct mapping of the deformed vertices.

Algorithm 1 Building the local frame

build the bounding box (BB) // visit all the vertices and collect the *MIN* and *MAX* coordinates;
find the BB central point extracting the mean point between *MIN* and *MAX*;
set the origin of the local frame *ic* by taking the mesh vertex closest to the BB central point;
translate the mesh in order to put the vertex *ic* on the global reference system (GRS) origin;
define the axes of the local frame as:
i_u = index of the mesh vertex closest to the GRS axis y;
i_v = index of the mesh vertex closest to the GRS axis x;
i_w = index of the mesh vertex closest to the GRS axis z.

In a second preprocessing stage, all vertices on the object surface are transformed into spherical coordinates, with respect to the local reference system. ϕ and θ values are then normalized to index a hash matrix as shown in Equations 4 and 5, where $ratio_t$ and $ratio_p$ define the size of the cell in degrees. The index of the vertex is added to a list associated with the selected cell. Algorithm 2 describes this procedure.

$$i_x = 180\theta / ratio_t \pi \tag{4}$$

$$i_y = 180\phi / ratio_p \pi \tag{5}$$

The last step is then the collision detection itself, done at each integration step of the mass-spring system. The first test is done with the bounding sphere of the mesh; if there is intersection, the subsequent tests transform the vertices

Algorithm 2 Building the spherical hash table

map the vertex to local frame coordinates;
draw a vector from the local frame origin to the vertex v_i;
calculate the spherical coordinate for the vector, obtaining ϕ and θ;
add the index of the vertex v_i to the list of a hash matrix indexed by a scale transformation on ϕ and θ.

into the deformed coordinates system. The collision test is made once for each pair of objects. Consequently, we can say that we have made $N(N-1)/2$ tests between objects, where N is the number of objects. For the M_i mesh, tests should be made only with the mesh with indexes greater than i. Algorithm 3 presents in detail this procedure. Notice that no update step is required for the hash matrix. The table is constructed at the initialization step and do not require updates because the local axes move and deform with the mesh.

Algorithm 3 Collision detection between meshes

```
for all vertex v_i on a mesh M_i do
  map v_i to the local frame of the mesh M_j, generating v'_i;
  map v'_i to spherical local coordinates and find the corresponding cell on the hash matrix of mesh M_j;
  for all k index in the vertex list of the cell do
    take the vertex v_k of the corresponding mesh M_j;
    if dist(v_i, O_Mj) < dist(v_k,O_Mj) then
      report a collision
    end if
  end for
end for
```

As creation of the hash table is made only once in preprocessing time, it is performed on the CPU and the results are sent and stored in the GPU memory. The hash is represented by a matrix containing the bases that index an array that contains the lists of particles in each cell of the hash. Each particle is represented by an index that points to its parameters in the other data structures, including the array of positions.

Each particle of a mesh M_i is mapped to be processed independently by a thread of the GPU. The thread, in turn, maps the position of the particle to the spherical coordinates of the mesh M_j, indexing the hash for M_j and obtaining the list of particles of mesh M_j that can possibly collide with the particle of mesh M_i. If the distance from the v_i particle from mesh M_i to the origin of M_j is less than the distance from the colliding vertex v_j to the origin of M_j, then there is a collision. The GPU implementation of the hash matrix is similar to the adjacency list described in Section III.

The mapping works even if the basis is non-orthogonal. Only possible problems are:

1) Two of the basis vectors become linearly dependent. But it only happens if the model is completely crushed;
2) Two of the particles defining the basis invert positions. It also only happens if the model undergo severe and non-biologically acceptable deformation.

Of course, mapping resolution will vary with frame deformation, but this can either improve or worsen a bit the detection quality.

Combining the efficiency of the Local Spherical Hash algorithm with the power offered by the parallelization achieved with GPU programming, it is possible to quickly detect collision between a number of complex deformable objects during a physical simulation.

V. Haptic interaction

A. Haptic interactions

Haptics is the human sense of touch. It is related to the perception of tactile and proprioceptive information as vision is related to visual information. Computer haptics is analogous to computer graphics and deals with various aspects of computation and rendering of touch information when the user interacts with virtual objects.

Computer haptics require a haptic rendering device. The latest smartphones and other mobile devices like media players and game consoles all come with some sort of haptic interface. Moreover, haptic mice and joystick are also available. Such devices render vibration and bumps as tactile information to the user. Tactile information improve user productivity while using the graphical interface as now touch brings new cues for interaction, especially when combined with visual and auditory information.

Other less popular but well developed devices render force information. The most successful and widely used haptic devices today are the Phantom family developed by Sensable. The Phantom Omni, for example, is a generic 6 degree of freedom (DOF) input and 3 DOF output force-feedback device featuring an articulated arm with three uniaxial rotational joints and one 3-axis rotational gimbal.

More than just a device, an interaction model must be implemented to calculate the feedback forces. They are often based on some sort of collision detection and response (see section II-B). As soon as the collisions and contacts between virtual probe and the meshes are resolved, the collision information has to be used to provide essentially two types of response. The first response is a reaction in the model, which will deform if it is soft or move aside if it is rigid. The second is the calculation of a reaction force which will be delivered to the haptics device to determine how it will push the hand of the user. This second response is what we call *haptic rendering*.

Haptic rendering algorithms make use of basic interference tests to provide force feedback. The first haptic

rendering algorithms were based on vector field methods [18]. Then, to overcome the many drawbacks of vector fields, the concept of god-object was introduced by Zilles and Salisbury [19]. A god-object is a virtual model of the haptic interface which conforms to the virtual environment. Actually, one cannot stop the haptic interface point from penetrating the virtual objects. The god-object represents the virtual location of haptic interface on the surface of the objects, the place it would be if the object was infinitely stiff. Recently, Ortega et al. [20] introduced an extension of the god-object paradigm to haptic interaction between rigid bodies of complex geometry. The authors used a 6-dof stringed haptic workbench device and constraint-based quasi-statics with asynchronous update to achieve stable and accurate haptic sensation. However, due to the use of an inefficient collision detection method, they cannot guarantee a 1kHz frame rate, which may result in inaccurate representation of high-frequency interactions such as when objects slide rapidly on each other.

B. Haptic rendering strategy

The haptic rendering procedures were implemented targeting the Sensable Phantom Omni force-feedback device (as the one shown on Figure 6. Concerning software, we used Open Haptics library to communicate with the Phantom, but only the input/output application program interface to communicate with the Phantom through the asynchronous HD interface. As we need total control of the system behavior for optimization reasons, we are not using any of the included methods for collision detection or scene modeling. Additionally, it is important to state that these methods cannot handle deformable models.

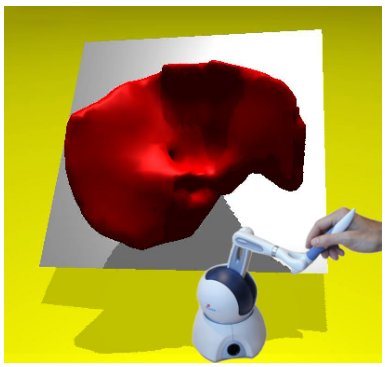

Figure 6. Application interface with the shovel tool (in gray) used to interact with a reconstructed human liver model. Phong shading and shadow mapping were used to increase scene realism. In the lower right we can see the Phantom Omni device in use.

The interactive experience is based on the manipulation of the Phantom arm. Moving the Phantom arm, the user controls a flat rectangular tool like a shovel in the virtual environment, as shown in Figure 6. Positions and orientations of the tool are read from the device and used as the input for our simulator, that physics module detects the collision at every simulation step, while the graphics engine use it for rendering purposes.

The collision detection method applied for tool-mesh contact determination is trivially performed calculating scalar products with the tool plane normal and edges, as illustrated in Figure 7. If a particle was above the plane on the instant time $t-1$, and on time t it is under the plane, then, for each tool vertex a vector from the vertex to the projection of the particle over the plane is traced. If all the angles between the vector to the particle and the adjacent edge of the tool are less than $90°$, then a collision is reported.

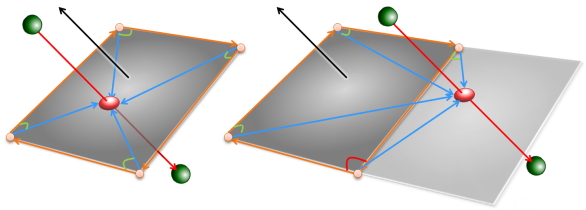

Figure 7. Collision detection between a mass point of the deformable object and the shovel (gray rectangle): vectors are traced from the corners of the tool to the projection of the particle over the plane. If all of them produce a sharp angle with the tool border, then there is a collision. If at least one angle is greater than $90°$, then the particle is not colliding with the tool.

Once a collision between an object and the shovel is detected, the force feedback exerted by the Phantom on the user hand should be calculated. This force is calculated by measuring the distance from each vertex of the object (only vertices that crossed the shovel) that crossed the shovel plane till the plane itself.

The greatest distance is then used to set the force feedback. This value is efficiently found using GPU processing. A reduction operation is performed and consists in the subdivision of an array of distances among all threads recursively, until the result (greatest distance) is found and put in the first position of the array.

VI. Experiments and Results

In order to evaluate the performance and practicability of the techniques proposed in this paper as well as the CUDA implementation, a system prototype was developed. The system – consisting of an interactive virtual environment with realistic graphics and haptic rendering – conjugates a number of functionalities, including physics-based deformation, collision detection between complex deformable models, contact evaluation between a haptic probe and complex meshes, force-feedback computation and great quality Phong shading with shadow maps (see Figure 8). Different modules run at different update rates, being the haptics computation and collision detection made at higher rates than physics and graphics.

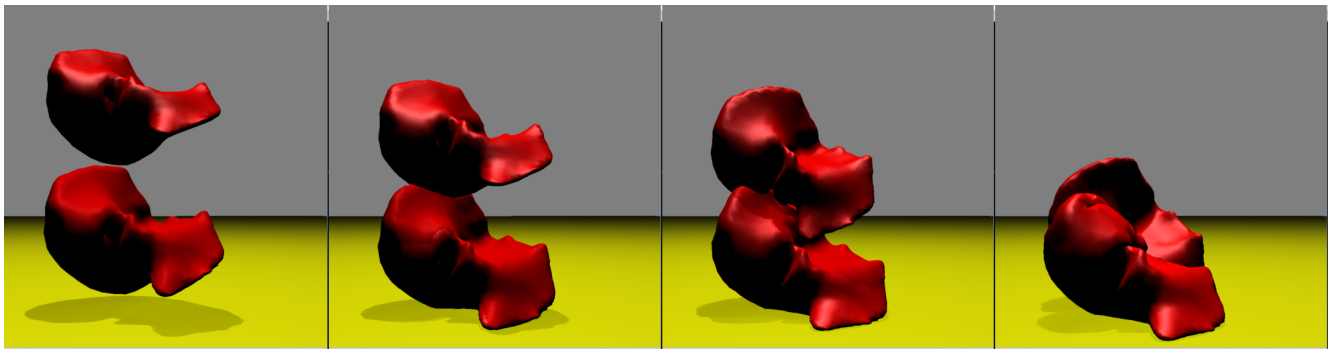

Figure 8. Four frames of a simulation of two objects dropping down, colliding and going to its rest position on the floor.

The performances are measured for a complete working system. As the GPU is used for both graphics and general purpose, we purposely limited the number of graphics fps to save GPU time for deformation. It is widely known that around 30 graphic updates per second is enough to fool the human eye.

As our target application is liver surgery simulation (also named hepatectomy, a surgery for liver resection and transplantation), a human liver model was reconstructed from the Visible Human dataset and used as a case study. The model composed by 1,312 mass points, 2,484 triangles, and 12,736 springs was used to perform experiments with the system. The hardware used in the experiments is a PC with an Intel Core 2 Quad Q6600 (2.4 GHz) processor, 2.0 GB RAM, and a GeForce 9600 GT (512 MB, 64 cores at 650 MHz).

Our tests were made comparatively. For this, we also implemented a parallel and very efficient CPU version of the technique. The CPU implementation uses the OpenMP API to parallelize the evaluation of the particles of the mass-spring system, and the collision detection between models and the interaction tool. A loop traverses all particles from a mesh M_i evaluating the collision with each mesh M_j. This loop is broken in so many threads as cores in the CPU.

Table I presents the graphics results achieved for the tests performed in CPU and GPU. In the same way, Table II presents the results for the same tests, but performed considering physical simulation rates. In all tests the haptic rate is kept around 1kHz. Observe that the update rates for graphics and physics processing decreases logarithmically in CPU when the number of objects increases. In GPU, the average update rates also decrease in the same rate, but are one degree of magnitude higher. The chart on Figure 9 summarizes these differences.

One limitation of the algorithm for collision detection may rise when handling meshes with large concavities or protuberances. This happens because in such cases the spherical coordinates system maps many vertexes to few cells, generating subsamples in others cells, which reduces the algorithm accuracy. One solution to maintain accuracy with those meshes, is to subdivide de mesh in a hierarchy of LSHs and handle them separately. Nevertheless, with smooth organic shaped objects, the algorithm is efficient and accurate.

	GPU		CPU	
n. objects	**mean fps**	**std. deviation**	**mean fps**	**std. deviation**
2	38	2.70	6	0.16
4	20	1.60	3	0.06
6	13	0.32	2	0.06
8	9	0.87	1	0.1

Table I
GRAPHICS FRAME RATE

	GPU		CPU	
n. objects	**mean fps**	**std. deviation**	**mean fps**	**std. deviation**
2	1,882	115	297	26.8
4	982	82.5	143	11.7
6	654	32.7	95	5.95
8	469	41.7	69	8.53

Table II
PHYSICS FRAME RATE

VII. CONCLUSION

The work presented in this article is part of a realistic surgery simulator that will involve high-level graphics and haptic rendering of soft tissues. We presented an efficient implementation of a mass-spring system in GPU using the CUDA architecture, as well as a novel collision detection method for deformable models. The system achieves real-time for graphics and haptic rendering. This means that update rates around 1kHz can be achieved for force-feedback while keeping graphics above 30Hz and explicitly integrating mass-spring simulation stable.

Tests have shown that an equivalent parallel implementation in CPU is one degree of magnitude slower than in GPU. The massive parallel power of GPUs enables a number of applications that were not possible in CPU implementations. Such level of efficiency and accuracy allows for the development of physically complex interactive systems, which require haptic feedback, as for example, realistic surgical simulators, immersive virtual environments, and games.

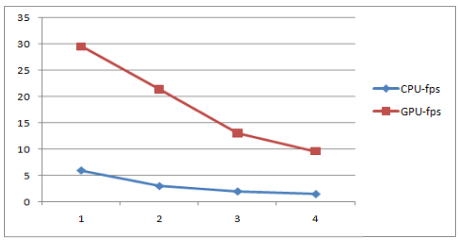

Figure 9. Graphics frame rate (FPS) decreases as the scene complexity increases for both CPU and GPU. Nevertheless GPU shows better results.

Actually, LSH does not handle self-collision. However, future works should still explore the subdivision of the models for a hierarchy of LSH to handle this issue and improve the physical model including volume preservation, for example. Other types of deformation involving topological changes – like cutting – should also be explored as previous approaches suffer from this kind of interaction. We believe our approach will keep the high rates achieved despite of topological changes.

ACKNOWLEDGMENT

This work was supported by Microsoft Brazil, and CNPq-Brazil through grants 481268/2008-1, 309092/2008-6, and 302679/2009-0.

REFERENCES

[1] C. A. Dietrich, J. L. D. Comba, and L. P. Nedel, "Storing and accessing topology on the gpu: A case study on mass-spring systems," *ShaderX 5 - Advanced Rendering Techniques*, pp. 565–578, 2006.

[2] O. Comas, Z. Taylor, J. Allard, S. Ourselin, S. Cotin, and J. Passenger, "Effcient nonlinear fem for soft tissue modelling and its gpu implementation within the open source framework sofa," *ISBMS*, 2008.

[3] A. Maciel, R. Boulic, and D. Thalmann, "Efficient collision detection within deforming spherical sliding contact," *IEEE Transactions on Visualization and Computer Graphics*, pp. 518–529, 2007.

[4] A. Lefohn, I. Buck, J. D. Owens, and R. Strzodka, "Gpgpu: General-purpose computation on graphics processors," in *Tutorial 3 at IEEE Visualization*, October 2004.

[5] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *HWWS 2002: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2002, pp. 109–118.

[6] J. Mosegaard, P. Herbor, and T. S. Sorensen, "A gpu accelerated spring mass system for surgical simulation," in *13th Medicine Meets Virtual Reality*, 2005.

[7] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on gpus," in *Simulation and Visualisation 2005*, 2005.

[8] E. Tejada and T. Ertl, "Large steps in gpu-based deformable bodies simulation," *Simulation Theory and Practice - special issue on Programmable Graphics Hardware*, to appear.

[9] M. C. Lin and S. Gottschalk, "Collision detection between geometric models: a survey," in *Proceedings of the 8th IMA Conference on Mathematics of Surfaces*, 1998, pp. 37–56.

[10] P. Jimnez, F. Thomas, and C. Torras, "3D Collision Detection: A Survey," *Computers and Graphics*, vol. 25, no. 2, pp. 269–285, Apr 2001.

[11] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics forum*, vol. 24, no. 1, pp. 61–81, mar 2005. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2005.00829.x

[12] A. Gregory, M. C. Lin, S. Gottschalk, and R. Taylor, "Fast and accurate collision detection for haptic interaction using a three degree-of-freedom force-feedback device," *Comput. Geom. Theory Appl.*, vol. 15, no. 1-3, pp. 69–89, 2000.

[13] C.-H. Ho, C. Basdogan, and M. A. Srinivasan, "Efficient point-based rendering techniques for haptic display of virtual objects." *Presence*, vol. 8, no. 5, pp. 477–491, 1999.

[14] M. SHINYA and M. FORGUE, "Interference detection through rasterization," *Journal of Visualization and Computer Animation 2*, p. 132134, 1991.

[15] J.-C. Lombardo, M.-P. Cani, and F. Neyret, "Real-time collision detection for virtual surgery," in *CA '99: Proceedings of the Computer Animation*. Washington, DC, USA: IEEE Computer Society, 1999, p. 82.

[16] G. Picinbono, J.-C. Lombardo, H. Delingette, and N. Ayache, "Improving realism of a surgery simulator: linear anisotropic elasticity, complex interactions and force extrapolation." *Journal of Visualization and Computer Animation*, vol. 13, no. 3, pp. 147–167, 2002.

[17] C. Basdogan, S. De, J. Kim, M. Muniyandi, H. Kim, and M. A. Srinivasan, "Haptics in minimally invasive surgical simulation and training," *IEEE Computer Graphics and Applications, IEEE Computer Society,*, pp. 56–64, 2004.

[18] T. H. Massie and J. K. Salisbury, "The phantom haptic interface: A device for probing virtual objects." in *Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, C. J. Radcliffe, Ed. Chicago: ASME, 1994.

[19] C. B. Zilles and J. K. Salisbury, "A constraint-based god-object method for haptic display," in *IROS '95: Proceedings of the International Conference on Intelligent Robots and Systems-Volume 3*. Washington, DC, USA: IEEE Computer Society, 1995, p. 3146.

[20] M. Ortega, S. Redon, and S. Coquillart, "A six degree-of-freedom god-object method for haptic display of rigid bodies," in *VR '06: Proceedings of the IEEE Virtual Reality Conference (VR 2006)*. Washington, DC, USA: IEEE Computer Society, 2006, p. 27.