Comparing efficient data structures to represent geometric models for three-dimensional interactive environments

Abstract

Data structures have been explored for several domains of computer applications in order to ensure efficiency in the data store and retrieval. However, data structures can present different behaviours depending on applications that they are used . Three-dimensional interactive environments offered by techniques of Virtual Reality require operations of loading and manipulate objects in real time. In this kind of applications , realism and response time are two important requirements. Efficient representation of geometrical models plays an important part so that the simulation may become real. In this paper, we present the implementation and the comparison of two topologically efficient data structures – Compact Half-Edge and Mate-Face – for the representation of three-dimensional objects for three-dimensional interactive environments for medical training. The structures have been tested at different conditions of processors and RAM memories. The results show that both these structures can be used in an efficient manner. Even both structures has shown qualities that would justify their use, Mate-Face structure has shown itself to be more efficient for the manipulation of neighborhood relationships and the Compact Half-Edge was more efficient for loading of the geometric models.

Keywords:
Data Structures, Compact Half Edge (CHE), Mate Face (MF), three-dimensional objects, neighbourhood relationships

1. Introduction

Data structures are widely explored, developed and improved to solve several problems in computer applications. Systems that demand processing large volumes of data in a short time are interesting problems to apply efficient data structures.

As technology has improved, computer graphical applications have become more common. These applications include three-dimensional (3D) interactive environments using Virtual Reality (VR) technology, which provides interactive and immersible systems involving users in a real-time computational simulation [1].

In general, VR attempts to immerse the users senses so that the virtual representation of life is as close to reality as possible. Therefore, applying VR techniques in computational tools that seek to simulate real-world situations are interesting approaches.

The use of three-dimensionality and interaction in real time makes VR an even more attractive proposition for training and simulations, once it gives the user the possibility of exploring and repeating several different procedures without any wear and material maintenance costs.

In most simulation and training applications the effect desired in terms of skills acquisition is only reached when the application provides sufficient realism so that the user may experience sensations close to the real ones. This realism requires a number of factors regarding visualization and also interaction with the three-dimensional environment.

Realistic objects require the use of models with appropriate colors, textures and lighting, and also requires high resolution in several applications. Such representations have essentially been explored based on two main lines: three-dimensional object reconstruction using real images or synthesized objects for building artistic models, nearly always using applications which generate meshes. This latter category, which is highlighted in this work, can produce objects with a large number of vertices and cells that are manipulated in real time, for example, to detect collisions and simulate deformations arising from user interaction.

Methods for these and other functionalities work with meshes by manipulating sets of vertices, changing their positions within the Virtual Environment (VE), so that they can simulate changes that occurred in the objects considering their properties [2]. The computational costs of these iterative methods are normally high, especially in terms of processing time.

Considering the paradox between precision, computational cost and the need for feedback in real time, what is essentially needed in the case of interactive 3D environments applications is the study of Data Structures (DS) to enable the efficient storage and recovery of data for representing flexible objects.

Within this context, as presented, the purpose of this work is to compare the Data Structures known as *Mate-Face* [3] and *Compact Half-Edge* [4], considering as basic parameters the processing time and the memory use. These DSs were integrated to a framework that allows the generation of three-dimensional interactive medical training applications [5] The applications generated by this framework are basic tools to simulate biopsy exams, where a human organ is represented by a surface composed by interconnected vertices. This type of application requires precision to realistically simulate the procedure. Therefore, a vast number of points are stored to represent human organs.

From the analysis conducted it is possible to supply an efficient way to manipulate vertices and cells that represent synthetic objects. Therefore, functions as collision detection and deformation of objects can be improved and contribute to generate applications with greater realism.

This work is established as follows:section 2 shows the concepts about the representation of topological meshes. Section 3 presents a historical overview and related works that address the issue of efficient data structures. The development of the DSs appraised is shown in section 4. The results obtained with completed experiments and the discussions about them are made available in section 5. Finally, section 6 presents the conclusions of the work.

2. Meshes representation

Considering the definitions as proposed by Cunha [3] and Ferreira [6], the present work takes into account the concepts presented hereunder, for the implementation of Data Structures.

With given $p_0, p_1, ..., p_n \in \mathbb{R}^m$, the space defined in Equation 1 is known as an **affine space** (aff) generated by $p_0, p_1, ..., p_n$.

$$S = \{p \in \mathbb{R}^m : p = \sum_{i=0}^{n} \lambda_i p_i, \quad \sum_{i=0}^{n} \lambda_i = 1\} \tag{1}$$

A **simplex** σ of dimension k, also known as a k-simplex, is defined by the convex hull involving $k+1$ points $p_0, p_1, ..., p_k \in \mathbb{R}^m$, in general position, e.g., the points are placed in such a way that the vectors $p_1 - p_0, p_2 - p_0, ..., p_k - p_0$ are linearly independent.

This means to say that a k-simplex σ can also be defined by the set shown in Equation 2.

$$\sigma_k = \{p \in R^m : \quad p = \sum_{i=0}^{k} \lambda_i p_i, \quad \sum_{i=0}^{k} \lambda_i = 1, \quad \text{with,} \quad \lambda_i \geq 0 \quad \text{for} \quad i = 0, \ldots, k\} \tag{2}$$

When the context is understood, we shall say that the simplices of dimensions 0, 1, 2 and 3 are respectively a *vertex*, *edge*, *triangle* and *tetrahedron*. Similarly, the points $p_0, p_1, ..., p_k$ belonging to a k-simplex σ are called *vertices* of σ. A subcell, also known as a **face of a simplex** σ is the convex hull of a subset of vertices of σ and is, by definition, also a simplex. A simplex is said to be adjacent to (or in the neighbor of) another simplex if they both have a common sub-cell.

A **simplicial complex**, also known as a mesh, is a collection K of simplices, where the following conditions hold:

1. If γ is a face of a simplex of K, then $\gamma \in K$;
2. The intersection of two simplices of K is either empty or a sub-cell of one of the simplices;
3. Any compact set C intersects K at a finite number of simplices.

Simplicial complexes are a discrete representation of mathematical objects that we call manifold. In the particular case where the dimension of the manifold is 2, it is called a surface. Figure 1 shows examples of meshes that represent manifolds of dimension 2 (on the left) and 3 (on the right).

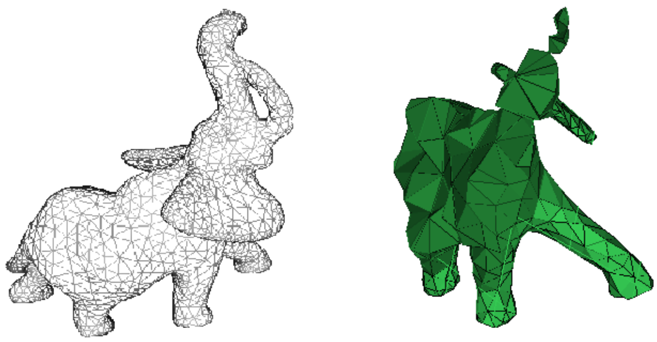


Figure 1: On the left, a triangular mesh of one surface, and on the right a tetrahedral mesh of one volume.

For a simplicial complex K of dimension n, a $(n-1)$-simplex is an **inside simplex** when it is shared by two n-simplices; should this not be the case, it is a **boundary simplex**. The simplices with dimensions of less than $(n-1)$ contained in boundary simplices are also considered boundary simplices. The **boundary** of K is a subcomplex $S \subset K$ comprising all the boundary p-simplices , for $p = 0, 1, \ldots, n-1$ of K.

Topological Data Structure seeks to index the elements of the mesh in a way that it represents the relations of incidence and adjacency among the different elements and also to make it easier to recover the information. The method in which the structures are stored may be implemented explicitly (in vector form, for example) or implicitly (by recovering the information through arithmetic operations). The advantage of using implicit forms is the low memory consumption, as the explicit forms enable greater access and less recovery time.

3. Related Work

Over the years, mainly due to the intensified activity of computer science in different areas of knowledge, it has become more and more important to find ways to conserve memory consumption while increasing the efficiency of data structures to make applications as close as possible to reality.

Studies of topological DSs started in the 1970s. Besides the comprehension of works which had previously defined DSs, there are also comparative projects about topological DSs.

In 1975 there the first significant structure was introduced, developed by Baumgart [7], called Winged-Edge. Since then, other experts have based their works on this theory to create other structures over time. Figure 2 shows the sequence of DSs that emerged.

Winged-Edge was one of the first work projects proposed to represent surfaces in $\mathbb{R}^3$. It uses edges to access the data of a mesh. Each edge keeps the vertices of its extremities, the left

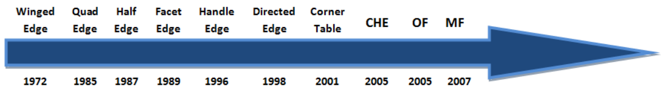


Figure 2: Time line showing the DSs thus created.

and right faces and also the preceding and succeeding edges in relation to the left face [7]. In 1983 Guibas and Stolfi [8] presented a generalization of *Winged-Edge*, called *Quad-Edge*, which is able to represent non-orientable surfaces. Then there is *Half-Edge*, as proposed by Mantyla in 1988 [9], which also uses edges to access the data of a mesh; each edge is divided into two *half-edges*, one for the left-hand face and the other for the right-hand face, with opposite directions (Figure 3). Each *half-edge* stores one vertex and one incident face, while for each vertex and face their respective *half-edges* are then stored.

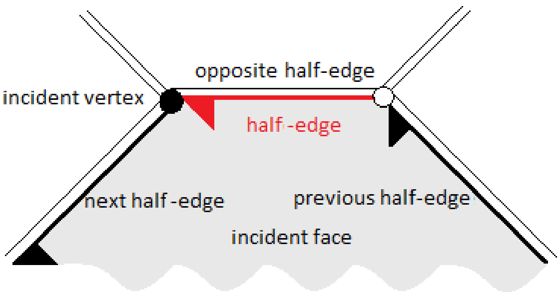
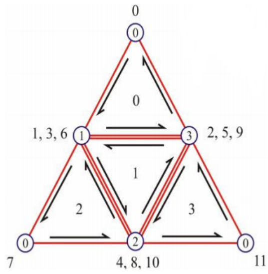


Figure 3: *Half-edge* and its associations.

In 1996, Lopes [10] presented the *Handle-Edge* which is just an extension of the *Half-Edge*, with the difference that here there is explicit representation of boundary curves and also the use of nodes for representing the mesh elements. The nodes are: faces (triangles), vertices (insiders and boundary) and *half-edge* (insiders and boundary).

The first structure to use the concept of scalability was *Directed-Edges*, presented by Campagna *et al.* in 1998 [11]. In 2001, Rossignac and their collaborators [12] proposed the *Corner-Table*, a structure which represents triangular meshes using the concept of corners to represent the association of a triangle with its vertices. The triangles are implicitly stored using the equation $t = \lfloor c/3 \rfloor$ (where t is the number of the triangle and c is the *corner*), while the neighborhood among triangles is defined by a corner c and its opposite corner $O[c]$ which has the same opposite edge. Figure 4 shows a tetrahedron and its elements using the *Corner-Table* structure.

Opposite-Face (OF), presented by Lizier in 2004 [13] is based on the *Corner-Table* DS. It explicitly represents the vertices and the cells of a mesh, while the edges and the faces (only in three-dimensional cases) are stored implicitly. The vertices and cells are stored in single and non-negative indices, which allows direct access to any one of them. The representation of the vertices takes place through its geometric coordinates. The cells are represented by an index vector which shows the vertices of a given cell and also a vector of opposite cells, where the opposite cells of each of the vertices are duly stored.



Corner	Vertex	Triangle	Opposite
0	0	0	2
1	1	0	2
2	3	0	2
3	1	1	0
4	2	1	0
5	3	1	0
6	1	2	3
7	0	2	3
8	2	2	3
9	3	3	1
10	2	3	1
11	0	3	1

Figure 4: Open tetrahedron and its elements (adapted from Cunha [3])

Data Structure	Objects	Method of Indexation
Winged-Edge	[a] OS without boundary	By Edges
Quad-Edge	[b] NOS without boundary	By Edges
Half-Edge	OS without boundary	By *Half-Edges*
Handle-Edge	OS with boundary	By *Half-Edges*
Directed-Edge	OS without boundary	By *Half-Edges*
Corner-Table	OSs without boundary	By *corners*
Opposite-Face	NOS without boundary	By *corners*

[a]OS = Orientable surfaces
[b]NOS= Non-orientable surfaces

Table 1: Data Structures and Objects Represented.

All the Topological Data Structures presented are used to represent different objects, whether in two or three dimensions, which means that each and every DS uses its own technique. For greater clarity, Table 1 shows the relationships among the DSs and the objects thus represented.

There has been continuous research conducted aimed at finding efficient DSs for the representation of different kinds of objects. Many of these investigations explain how each one operates in terms of storage, neighborhood relationships and also comparisons between different algorithms of the same DS, as shown next.

Chen *et al.* [14] presented a versatile data structure known as *Edge-Based*, based on central edges. These researchers also proposed an efficient algorithm which calculates the least distance between two vertices. The algorithm was analyzed using speed tests and memory consumption. Ce Fan [15] described another versatile data structure based on edge-symmetry, which can be applied to represent three-dimensional objects. Performance was also analyzed based on time and space complexity.

Weiler [16] compared four different data structures (*winged-edge, modified winged-edge, vertex-edge and face-edge*), all of which are based on edges to obtain a relationship of adjacency, with representations aimed at solid objects with curvatures. These structures are classified into groups according to the different forms of usage from the information obtained through the adjacency relationship they have in common. The four DSs were analyzed and also compared regarding the time and space required for storage and also the complexity of the algorithms.

Also Floriani and Hui [17] compared topological DSs of different representations, which are classified based on the dimensions of the objects worked on (two or three dimensions) and their own methods of representation. The comparisons are made for several structures of each category, based on the storage cost and the complexity efficiency.

In our work we consider the need for realism to represent objects in three-dimensional interactive environments. With this purpose in mind, this work implements and compares the performances of two DSs: *Compact Half-Edge* (CHE) and *Mate-Face*, in order to efficiently represent two-dimensional geometric models of human organs.

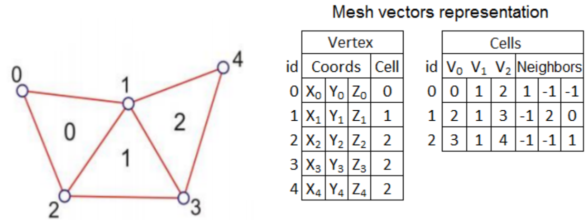


Figure 5: Representation of the mesh and also of vertices and cell vectors, by the MF structure (adapted from Cunha [3]).

4. Data structures development

In this work we have implemented two data structures: *Mate-Face* (MF) [3] and *Compact Half-Edge* (CHE) [4]. CHE has the advantage of being a scalable structure able to balance performance and memory. If there is any availability, the structure allows the use of additional memory to improve their performance and, for example, the border edges can be stored in a separate vector, so that there is no need to conduct a search to obtain the border of a given surface. On the other hand, MF has an interface which is both simple and efficient, which can also represent mixed meshes, making it easy to integrate with any application using meshes.

Both DSs were implemented in Java programming language. The codes developed consider structures that represent triangular and two-dimensional meshes. Next, we present details of the operation of the structures selected and the implementation made in order to make the recovery of information more efficient.

4.1. Mate-Face Data Structure

The *Mate-Face*, developed by Cunha [3], is a flexible structure created with the capacity to represent simplicial complexes in two and three dimensions, as well as meshes with other types of polygons, such as quadrilaterals. The structure was based on the *Opposite-Face* structure [13], and the main differences are the possibility of representing edges and faces in explicit form and also the method of indexing neighboring cells.

The *MF* consists of a vector of vertices which stores the respective coordinates, as well as a reference to the last incident cell; a vector of cells which stores their respective vertices and a reference to the adjacent cells, and a mesh that stores the cells and vertices. This structure may also contain a vector of edges, but in this work this possibility was not implemented in its first version, with the edges being implicitly represented. Figure 5 shows the composition of the *MF*.

In Figure 5 the position 0 of the vertex vector represents the vertex 0, which is defined by the coordinates x_0, y_0 and z_0, and which is incident to cell 0. We also see that the position 1 of the cell vector represents cell 1 (the same index), which is formed by vertices 2, 1 and 3, and its neighboring cells numbered 2 and 0. Following these rules for storing data, we can represent the whole mesh through these two vectors.

The neighborhood relationship in *MF* is not always obtained through corners as is the case with the *OF* structure it can also be obtained through *half-edges*, depending on the type of mesh considered. In other words, the neighborhood relationship depends on the type of mesh used, and not just on the opposite vertices. In the cases of triangles and tetrahedra, the adjacent cells continue to be indexed based on opposite vertices.

In the present paper the neighboring cells are established after the vectors of vertices and cells have been loaded. The cell vector is examined to identify cells that have two common adjacent vertices, e.g., which have one common edge. For this, is necessary to reach the vertices of one cell based on a specific incident vertex v, using Equations 3 and 4, where $next(i)$ and $prev(i)$ produces the local index of the next and the previous vertex of v respectively, and the symbol "%" is the division remainder.

$$next(i) = (i + 1)\%3 \tag{3}$$

$$prev(i) = (i + 2)\%3 \tag{4}$$

Figure 6 shows the neighborhood relationships for each cell constructed in relation to the index of opposite vertices.

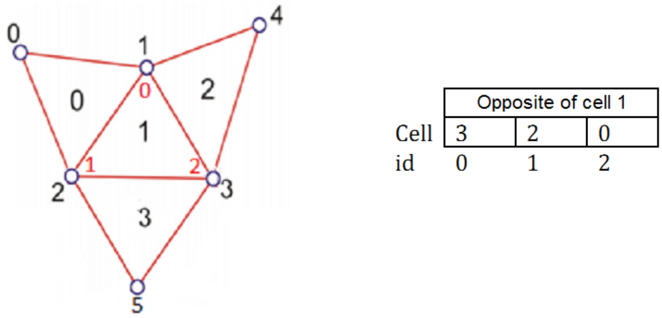


Figure 6: Neighborhood relationship by opposite vertex (adapted from Cunha [3]).

Analyzing Figure 6, we can see the neighborhood relationship being constructed through the local l indices of the cell in question. This means that cell 1 is the neighbor of cell 3 through the vertex opposite 1 with local index equal 0. Likewise, we see that this same cell 1 is also the neighbor of cell 2 through the vertex opposite vertex 2 with local index $l = 1$, and also for cell 0 through the vertex opposite vertex 3 with local index $l = 2$.

Though the storage of these indexes is reliable, it is a lengthy process as its complexity is quadratic relative to the number of faces. The processing time was reduced by creating a second method to construct the neighborhoods, making use of a *hash* table as an auxiliary structure to optimize the process.

4.2. Compact Half-Edge Data Structure

The *Compact Half-Edge* structure was presented by Lages et al [4]. This is a structure for the representation of triangles divided into 4 levels, in order to enable changes to the amount of data stored, improving efficiency. In the present work levels 0 and 1, and part of level 2 were implemented, sufficient to represent the structures in the context of this work.

At **Level 0** (known as the **Triangle Soup**) only the basic information of the mesh is stored: the vertices with their coordinates and also the cells that contain them. The coordinates of the vertices are stored in vector G. This level is only for viewing the mesh, and does not represent the relationships of adjacency. Each triangle is implicitly represented by 3 *half-edges*. The index i of the *half-edge* is represented by Equation 5, where t is the index of the cell and k is the index of the *half-edge* he in the cell. In other words, the index of the *half-edge* is 3 times the index of the cell plus the local index of the he in the cell. Cell 30, for example, has the *half-edges* 90, 91 and 92.

$$i = k + 3 * t \tag{5}$$

The *he's* are represented by a vector V of integers, in which the position he shows the index of the vertex of origin, known as the "foot ". Equations 6 and 7 re used to obtain the previous he and the next he within a cell.

$$next(he) = 3 * \lfloor he/3 \rfloor + (he + 1)\%3 \tag{6}$$

$$prev(he) = 3 * \lfloor he/3 \rfloor + (he + 2)\%3 \tag{7}$$

Level 1 deals with the connection among the triangles, known as **Adjacency among Triangle**. For this purpose, the concept of opposite half-edge is used. As each edge is only attached to two vertices, these vertices are therefore adjacent. We therefore look for half-edges that form the same edge but considering the opposite directions, as shown in Figure 7.

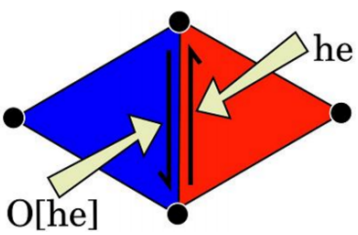


Figure 7: Representation of *half-edge* and opposite *half-edge* on *CHE* Data Structure (extracted from Lopes [10]).

The adjacencies are stored in a vector O. The index of the position he for this vector contains the index of its opposite half-edge. If the he is a border, which means it does not have an opposite edge, then the index stored is -1. Like in the *MF* structure, a second method was created in order to add adjacencies using a hash table as an auxiliary tool to optimize the algorithm.

In **Level 2**, which is known as **Representation of Cells**, two new vectors are created, *Edge Map* (HE) and *Vertex Half-Edge* (VH). The vector *HE* contains the edges of the mesh. The edge is represented by one of its *he's*, as the other can be recovered using the vector O. For the vector *HE*, the vertices that make up the edge are also stored, recovered using its constituent *he's*.

The vector *VH* stores, for each vertex index, an incident *half-edge*. This vector is important for star operations on the vertex (explained below), in which there is a need to pass through all the incident cells of a certain vertex. As is the case for the *CHE* structure, the process of searching for the cells starts with the opposite cells. This means that it is necessary to know one edge that is incident on that vertex.

In this work, we have created a *Boolean* variable to state when to add, or not add, the incident *half-edge* to the vector. If this is not added, then there is a search through the vector V to find an incident *half-edge*.

At the last level (**Level** 3, called **Representation of Boundary Curves**), a structure is created to explicitly represent the border curves of the surface. Each border curve is represented by one of its incident *half-edges* and all the others can be found by other element vectors. To store the curves, a vector designated *CH* is created.

Figure 8 shows how a *CHE* is composed with all its levels. Besides the aforementioned structures and methods, a method has been created for scanning the star based on a given *half-edge*, in a manner similar to that of the *MF* structure.

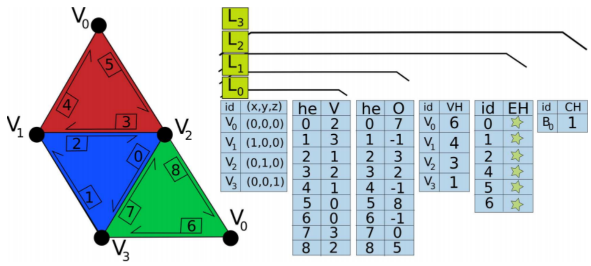


Figure 8: Representation of *half-edge* and opposite *half-edge* in the CHE Data Structure (extracted from Lopes [10]).

4.3. Recovery of the Neighbors of a Vertex

A relevant operation in a DS is the Operation of *Star of a Vertex*, which passes through all the incident cells attached to a given vertex. This operation is extremely important in 3D interactive environments when operations such as deformation must be executed. This type of functionality must find the neighboring vertices of a selected vertex to displace them in order to simulate changes in the object format.

The operation is performed based on a cell containing the starting vertex. In the example of Figure 9-a, the starting cell is cell 0. In this example, this is the last cell to be stored which contains the vertex highlighted in red. After this, there is the passage through neighboring cells with this same vertex, in a cycle, until it returns to the starting cell. In the case of a border vertex, as there are no more neighboring cells in that direction, the direction is then inverted in order to pass through all the cells until the other extremity, as shown in Figure 9-a. In Figure 9-b we see a complete Star cycle in the first level of vertices, which means only those which have cells in common with the central vertex (these vertices are highlighted in blue) and the second level which contains the vertices that have cells in common with the vertices of the first level (these vertices are highlighted in green in Figure 9-c).

The algorithms that traverse the star of a vertex are slightly different in the two structures, as seen in Algorithm 1 (for the MF structure) and Algorithm 2 (for the CHE structure). Both are based on the assumption that the mesh is coherently orientated. Notice that in lines 3 to 8 of Algorithm 1 there is a local search for the position of a given vertex within the cell that contains it, which does not exist in Algorithm 2.

Input: A vertex v_i
1 Let c_i be a cell that contains v_i;
2 Let L be a list of vertices neighboring v_i;
3 Find i the position of v_i in c_i;
4 $v_{n_i} \leftarrow$ next(i) in c_i;
5 Add v_{n_i} in L;
6 $c_o \leftarrow$ the opposite cell to c_i through the vertex v_{n_i};
7 **while** $\underline{c_o \neq c_i}$ **do**
8   Find i the position of v_i in c_o;
9     $v_{n_i} \leftarrow$ next(i) in c_o;
10   Add v_{n_i} in L;
11     $c_o \leftarrow$ the opposite cell to c_o through the vertex v_{n_i};
12 **return** $\underline{L}$

Algorithm 1: Algorithm to scan the first level of the star of a vertex in the MF structure

Input: A vertex v_i
1 Let he_i a half-edge that contains v_i;
2 Let L be a list of vertices neighboring v_i;
3 $he_{n_i} \leftarrow$ next(he_i) ;
4 $v_{n_i} \leftarrow$ vertex "foot "of he_{n_i} ;
5 Add v_{n_i} in L;
6 $he_o \leftarrow$ the opposite half-edge to he_{n_i} ;
7 **while** $\underline{he_o \neq he_i}$ **do**
8     $he_{n_i} \leftarrow$ next(he_o) ;
9     $v_{n_i} \leftarrow$ vertex "foot "of he_{n_i} ;
10   Add v_{n_i} in L;
11     $he_o \leftarrow$ the opposite half-edge to he_{n_i};
12 **return** $\underline{L}$

Algorithm 2: Algorithm to scan the first level of the star of a vertex in the CHE structure

5. Results and Discussions

In order to compare the structures implemented, we conducted tests using meshes with different number of vertices and different levels of complexity (Figure 10).

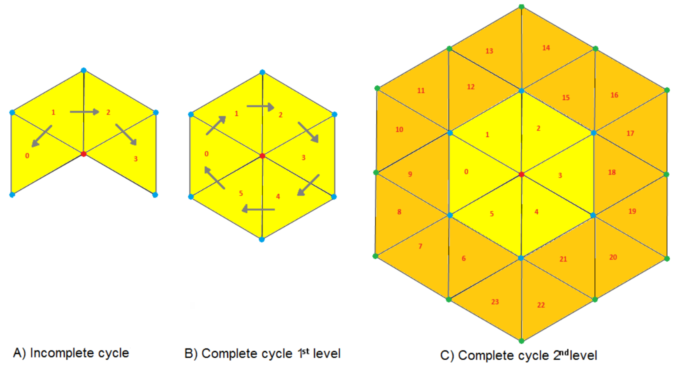


Figure 9: Star formats of the vertex, in MF: incomplete, complete, and multiple levels

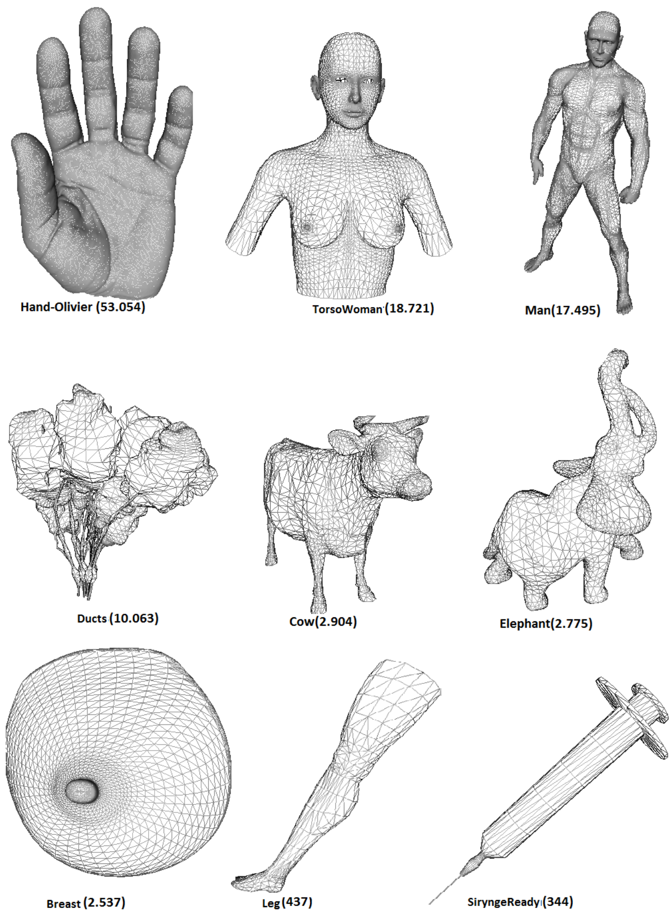


Figure 10: Meshes used in the tests, with the respective amount of vertices.

Three computers with different configurations were used to carry out the test (Table 2). Evaluating the performance of meshes using different computer configurations allows composing a discussion, even if inceptive, about how the processor and memory influenced the results.

In order to minimize the influence of the computer conditions during the tests, we made sure the computers did not keep other programs running, with only the programs related to the Operational System remaining active. Additionally, the tests

Table 2: Configuration of the computers used in the experiments.

Resource	PC1	PC2	PC3
Processor	Ci3 U380 @1.33GHz	Ci5 M430 @2.27GHz	Ci5 661 @3.33GHz
Memory (RAM)	4Gb	3Gb	4GB
OS	Wins7-64 bits	Win7-64 bits	Win7-32 bits

were executed 30 times in each equipment, recording all the processing times. At the end, the mean times were computed.

We first evaluated the loading mesh times in the implemented DSs in order to verify the influence of the mesh size on the performance of the structures. Each mesh was read from an ASCII file and its data were loaded onto the structures shown in Section 4. The test was executed 30 times using the implementation with *hash* table and 30 times without the *hash* table. Considering each previously cited equipment, we processed each mesh 180 times (total of 1620 executions) and recorded the time of each execution.

5.1. Processing time for loading meshes

Figures 11 and 12 show the comparison of the processing time for the three different computers, before the loading meshes using *hash* table. Figures 13 and 14 show the results obtained without using the *hash* table.

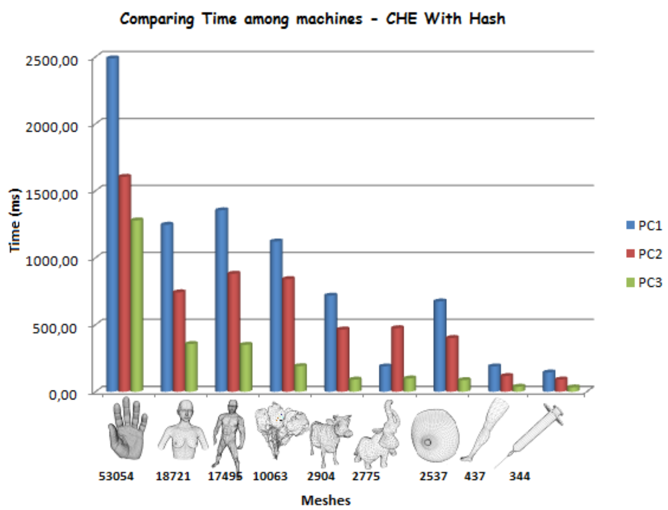


Figure 11: Comparison of processing time – *CHE* with *hash* Table

The first discussion is about the general performance of the DSs considering the number of vertices of the meshes. As expected, the four graphs show that the processing time is proportional to the number of vertices. The confirmation of this fact was somewhat expected, once the formation of the explicit memory structures (vectors) is directly dependent on the number of data units (vertices). This supports the hypothesis that efficient structures are needed to represent these objects in 3D interactive environments, as it requires precise modeling of the

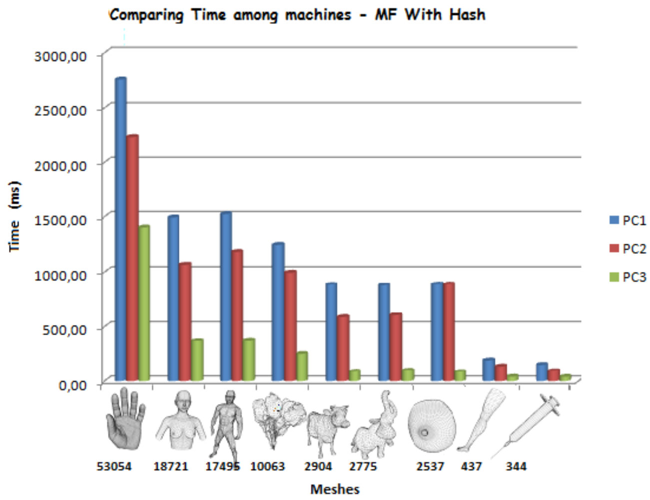


Figure 12: Comparison of processing time – *MF* with *hash* Table

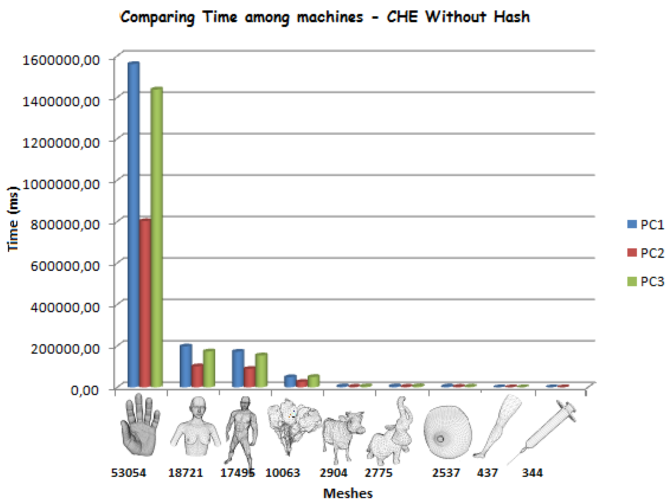


Figure 13: Comparative Graph between machines – *CHE* without *hash* Table

objects and consequently requires structures with many vertices to provide a realistic sensation.

The second observation is based on the graphs shown in Figures 11, 12, 13 and 14, which implies that some conclusions can be reached about the influence of the machines configuration on the results obtained.

We see, for example, that PC1, which has a slower processor than the others, had the worst performance in almost all time comparisons, which proves that, within this context, the processor required significant influence for the performance of the system. We also see that compared to PC2 this machine has the same memory capacity and the same memory capacity as PC3. Even so, in general the processor capacity showed to be the most decisive factor in the tests conducted.

In the comparison between PC2 and PC3, the results varied according to the search method used. Using a *hash table*, PC3 is always much faster than PC2, but without the use of *hash*

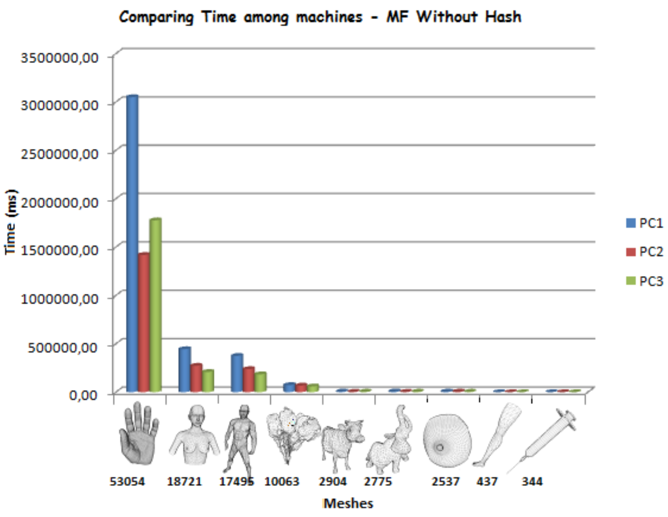


Figure 14: Comparative Graph between machines – *MF* without *hash*

its performance becomes worse than that of PC2 for meshes with a high number of vertices. This is probably because PC3 uses an operating system of 32 bits, meaning that it works with smaller memory words during the internal communication of the operating system. Communication using larger words, as in PC2 (operational system of 64 bits) may contribute to the fact that processing time is reduced, especially when there is a large amount of data to be transmitted. Therefore, although it has a faster processor, PC3 proved to be slower for systems that require many searches (accesses to RAM), which is the case of structures that do not use a *hash table* .

After analyzing the dependence of the linear structures implemented relative to the number of vertices of the models and the influence of the machine settings on the performance of the programs, it is interesting to discuss the effect of using *hash* table on the implementation of *hash* algorithms. Figures 15 and 16 show the graphs referent to the processing time for loading each of the meshes evaluated with and without the use of the *hash* table .

Figure 16 shows a comparison of the times with *hash* and without *hash* using the *CHE* structure. We see that the time difference is hundreds of times for the structures with a greater number of vertices.

For example, for the first model (*hand-olivier*), which has 53,054 vertices, the processing time with the *hash* table was 1,790 ms, which is less than 2 seconds. The same model, loaded by the method not using the *hash* table, required a total of 1,266,871 ms, which is more than 707 times than the procedure using the *hash* table.

An asymptotic growth was observed in the difference between the times with and without the use of the *hash* table. By way of example, for a model with a lower number of vertices, it can be seen that the *Breast* model (2,537 vertices) required 2,970 ms without the *hash* table and 387 ms with the *hash* table, which means that the time difference is only sevenfold.

It can also be observed that the same statements apply to the *MF* structure, as seen in Figure 15. For the *Hand-Olivier* model, the processing time with the *hash* table was 2,134 ms, while the same model, loaded without the *hash* table, required 2,081,108 ms considering the same processing machine. This means that the time without the *hash* is 975 times longer than the processing time with hash for this model. Comparing the second model mentioned here, the *Breast* model, the loading of the *MF* structure required 5,242 ms without the *hash* table and 621 ms with the *hash* table, which shows a difference of approximately eightfold, confirming the exponential behavior.

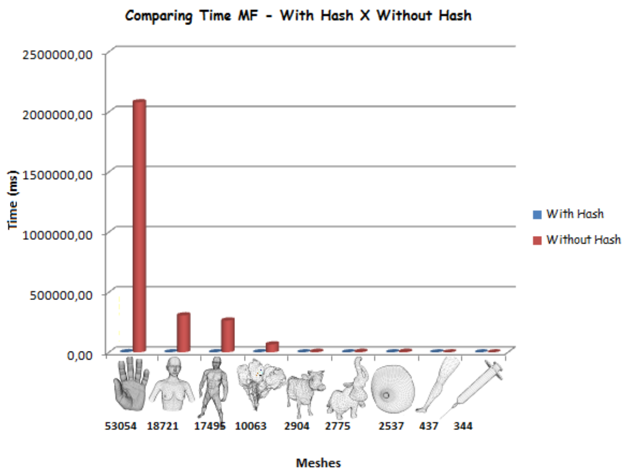


Figure 15: Comparative Graph *MF* with *hash* versus *MF* without *hash*

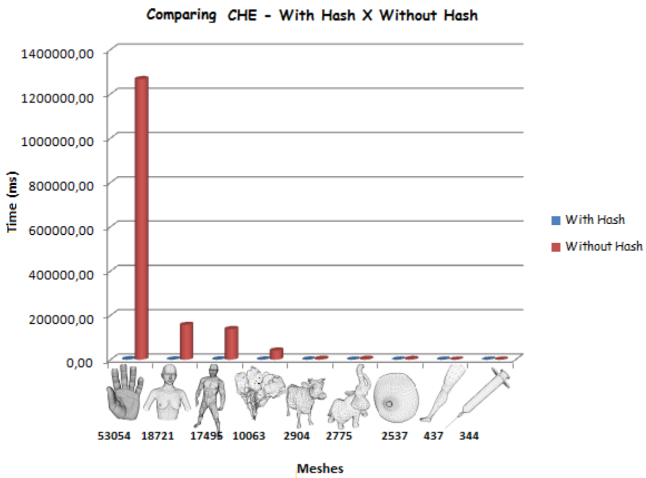


Figure 16: Comparative Graph *CHE* with *hash* versus *CHE* Without *hash*

Comparing the processing times obtained with each structure (Figures 17 and 18), we see a better performance for the CHE structure, shown by the models with more vertices. The loading of the Hand-Olivier model, for example, took 1,790 ms in the version with hash table for the CHE structure and 2,134 ms in the same version for the MF structure, a processing time difference of about 19% (Figure 17). Considering the same conditions for the second model (TorsoWoman, with 18,721 vertices), the time was 781 ms with hash table using CHE and 977

Comparing time without *hash* – *CHE* versus *MF*			
Mesh	*CHE*	*MF*	Percent
Hand-Olivier (53,054 vertices)	1,266.87	2,081.11	64%
TorsoWoman (18,721 vertices)	156.99	308.10	96%
Man (17,495 vertices)	138.12	265.31	92%
Ducts (10,063 vertices)	41.42	66.54	60%
Cow (2,094 vertices)	3.79	5.00	32%
Elephant (2,775 vertices)	4.01	4.99	24%
Breast (2,537 vertices)	2.91	5.24	80%
Leg (437 vertices)	0.19	0.22	16%
SiryngeReady (344 vertices)	0.13	0.16	23%

Table 3: Processing time (in seconds) for the meshes without *hash table* in both structures

Comparing Time with *hash* – *CHE* versus MF			
Mesh	*CHE*	*MF*	Percent
Hand-Olivier (53,054 vertices)	1.79	2.13	19%
TorsoWoman (18,721 vertices)	0.78	0.98	25%
Man (17,495 vertices)	0.86	1.03	20%
Ducts (10,063 vertices)	0.72	0.83	15%
Cow (2,094 vertices)	0.42	0.52	24%
Elephant (2,775 vertices)	0.25	0.53	112%
Breast (2,537 vertices)	0.39	0.62	59%
Leg (437 vertices)	0.11	0.12	9%
SiryngReadt 344 vertices)	0.09	0.09	0%

Table 4: Processing time (in seconds) for the meshes with *hash table* in both structures.

ms with hash table using MF, which means that there is a 25% time addition. This rate undergoes a variation, for other models, that ranges between 0% and 112%, and the tests conducted show that the processing time addition rate is not proportional to the number of vertices, according to Tables 3 and 4.

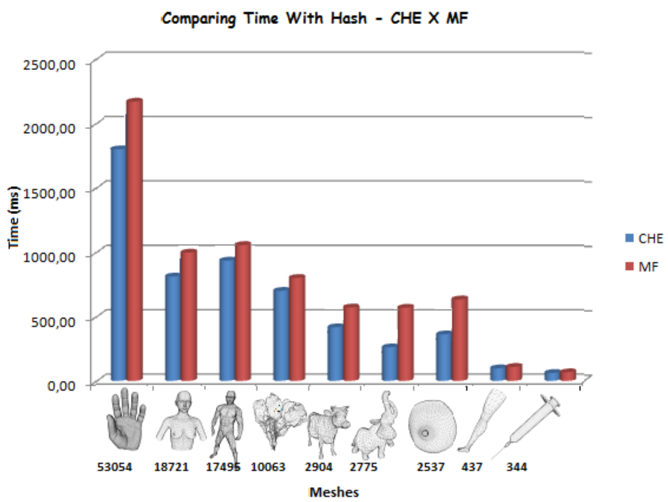


Figure 17: Comparative Graph with *Hash* – *MF* versus *CHE*

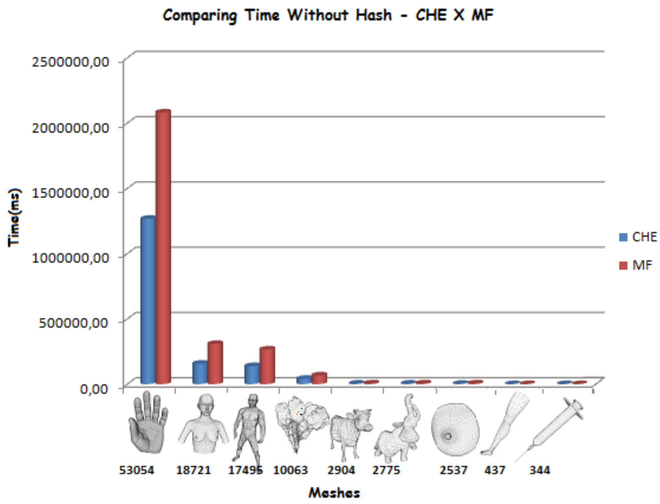


Figure 18: Comparative Graph without *Hash* – *MF* versus *CHE*

The use of the concepts of *half-edge* and opposite *half-edges* is more efficient than the use of corners and opposite cells. This result is extremely important for the context of the present work, considering the high number of vertices demonstrated by the models that represent objects in 3D interactive environments.

5.2. *Access time*

To test the access time for the mesh data, tests with the Vertex Star were conducted. In order to ensure the results achieved, we conducted tests considering 30 random vertices from each mesh. The accesses to the star of these vertices were performed from the first to the fourth level of each model tested. For each mesh, as well as for the previous tests, the process was repeated 30 times on each vertex and the average of the values obtained were computed Figure 19 shows the results of these tests for the 4 meshes that produced the most significant results. Table 5 shows the number of vertices at each level for these meshes.

What was observed is that *MF* is faster than *CHE* in almost all cases. There was only one case in which the *CHE* structure was faster than the *MF* (the second level of the *cow* mesh, where difference was 0.04 ms). The access time difference was of 15 ms. As an example, we mention the *Breast* mesh which showed a time of 1.26 ms for the *MF* structure at the fourth level, and 4.23 ms for the *CHE* structure at the same level. At the second level, the same mesh also showed a small time difference in the comparison of the structures, with 0.08 ms execution time *MF* and 0.04 ms with *CHE*.

For three specific meshes - Elephant, Man and Ducts - the *CHE* structure had a peak in execution time, showing a greater difference when compared with *MF*. This is the case with the Man mesh: with the *MF* structure the access to its fourth level took 0.94 ms while for the *CHE* structure the time was 15.88 ms.

5.3. *Use of Memory*

The analysis of memory use was carried out conceptually. We computed the values used for each of the data structures as analyzed, considering the implementation characteristics using the Java programming language.

For the *Mate Face* structure, two arrays are created:

- a vertex array, with the coordinates of the vertices (3 variables of the *float* type) and a cell which contains these coordinates (1 variable of the *int* type);
- a cell array, containing the vertices that each cell contains (3 variables of the *int* type) and the opposite cells (3 variables of the *int* type).

Variables of the *float* and *int* types take up 4 bytes of memory. Therefore, Equation 8 is used to calculate the memory use, where V and C are respectively the number of vertices and the number of cells and Mm is the total of bytes used for *Mate Face* structure.

$$\begin{aligned} Mm &= (3*4+4)*V + (6*4)*C = \\ &= (16*V + 24*C) \quad \text{bytes} \end{aligned} \tag{8}$$

For the *CHE* structure, three vectors are created:

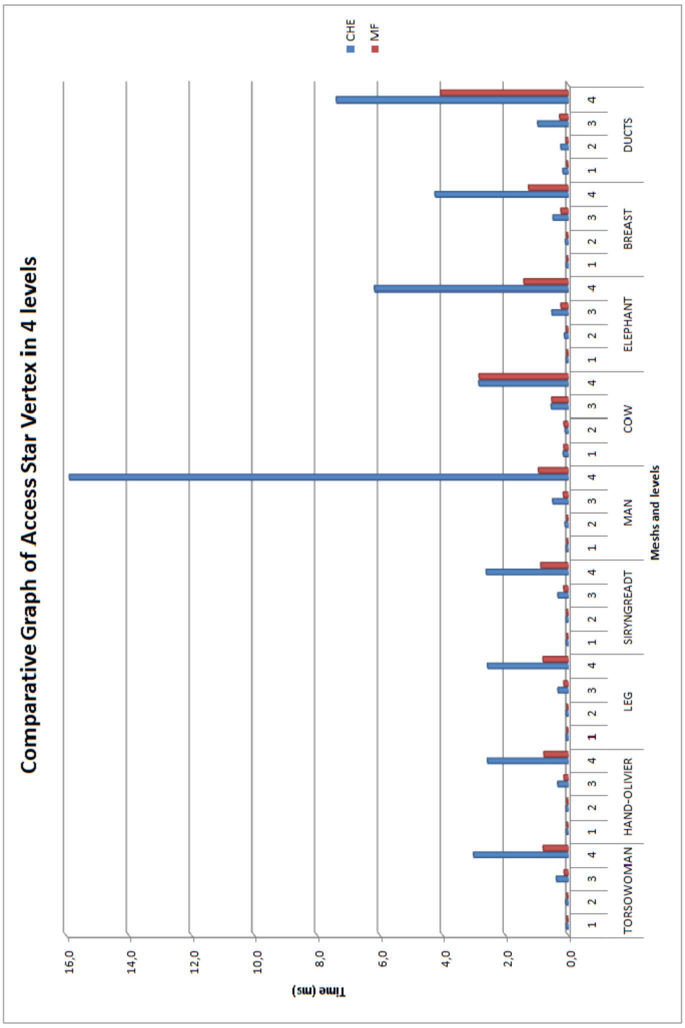


Figure 19: Comparison of processing time to the star vertex operation – Level 1 to 4.

Amount of vertices in each scanned vertex				
Mesh	Level 1	Level 2	Level 3	Level 4
TorsoWoman	6	17	34	57
Hand-olivier	6	17	35	58
Leg	6	18	35	58
SiryngReadt	7	19	37	61
Man	7	19	40	69
Cow	7	20	40	68
Elephant	7	21	43	72
Breast	7	22	49	90
Ducts	9	23	47	76

Table 5: Number of vertices per level, for each mesh as shown on the graph of Figure 19

- a array with the coordinates of the vertices (3 variables of the *float* type) and one half-edge containing these coordinates (1 variable of the *int* type);
- a array with half-edges, containing the index of their foot vertices (1 variable of the *int* type);
- a array of opposite half-edges, with the index of the opposite half-edges (1 variable of the *int* type).

As the number of half-edges is equal to 3 times the number of cells, Equation 9 calculates how much memory is used for the *CHE* structure, where V and C are the number of vertices and the number of cells respectively.

$$\begin{aligned} Mc &= (3*4+4)*V + 4*(3*C) + 4*(3*C) = \\ &= (16*V + 24*C) \quad \text{bytes} \end{aligned} \tag{9}$$

Evidently, the *CHE* structure has a greater use of memory, if it used with more levels. However, as in this work the structure has been used to half of Level 2 and has showed itself to sufficiently efficient at this Level, in this analysis both structures show exactly the same consumption of memory.

5.4. Integration of the Data Structures into the ViMeT Framework

A last and practical result of this work is the integration of the DSs in the *ViMeT Framework*. This framework is composed of a set of classes and a instantiation tool that allow generating applications for simulate biopsy exams. They were implemented in the Java language in order to provide a free and easy way to generate applications in this domain (available at http://www.each.usp.br/lapis/). In these applications the simulations of human organs must take into account the physical properties of these objects in order to provide realistic sensations during the virtual training. Furthermore, users must receive feedback of their actions in real time. Therefore, functionalities like collision detection and deformation require the displacement of a many vertices in a short time interval.

This integration required remodeling the instantiation tool to allow choicing the data structure (Figure 20) and the adjustments made into the code.

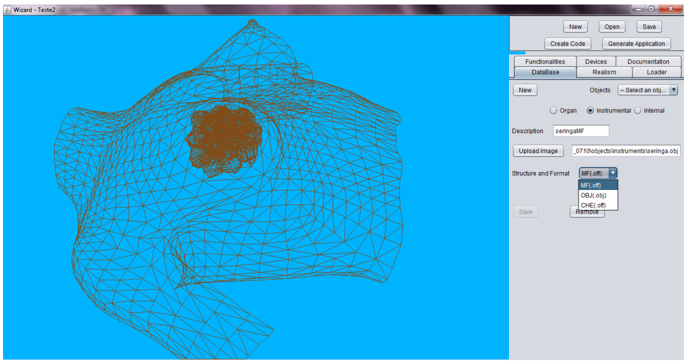


Figure 20: New Wizard Interface of the *ViMeT Framework*, with the possibility of choosing the structure for loading.

6. Conclusions

The results obtained indicate that both structures are interesting for applications in 3D interactive environments which require precision in the objects representation and response to user actions in real time. With this adequacy, we can verify that each one is indicated for a type of use in these interactive environments.

Based on the comparative analysis of the loading time for the models, in all tests conducted until now, the conclusion reached for the two DSs used in this work, was that, even with changes in memory, processor and operational systems for the machines used, the concepts of *half-edge* and opposite *half-edges* of the *CHE* structure ensure greater speed, that is 55% faster in the tests carried out.

In relation to access time, tested based on the Vertex Star using 30 random vertices in 4 levels, *MF* was usually faster. It should also be noted that the *CHE* structure, though usually slower, with regards to access time it is faster for loading meshes. Moreover, the CHE structure is easily scalable to represent other geometric structures such edge-maps and boundary curves.

Therefore, when the application requires constant loading of objects and these objects do not frequently have their topological structure changed the *CHE* structure is recommended. Games that generally that use 3D environments are good examples of this class of applications. However, when the loading is not often required, but the objects should accurately reproduce physical properties by displacement of the vertices, Then the *MF* structure is more suitable. Virtual medical training represents typical examples of these applications.

Another conclusion reached by this work regards to the use of the *Hash Table*, which has shown to be extremely efficient, and proved through tests conducted on machines with different configurations. This is because the hash table significantly reduces the number of memory accesses for searching neighborhoods, capacitating its use for loading the model more than 900 times faster.

In terms of memory use, the two structures proved to be exactly the same, except that *CHE* may be expanded to represent a greater number of geometrical structures such as border curves and edge maps. It is obvious that this expansion will involve an additional consumption of memory.

Acknowledgments

The authors thank to National Council for Scientific and Technological Development (CNPq) - process 559931/2010-7, So Paulo Research Foundation (Fapesp) - process 2010/15691-0, and National Institute of Science and Technology - Medicine Assisted by Scientific Computing (INCT-MACC).

References

[1] Burdea GC, Coiffet P. Virtual Reality Technology. Comput Animat Virtual Worlds 2003;16(5):559–60. doi:http://dx.doi.org/10.1002/cav.v16:5.

[2] Oliveira ACMTG, Pavarini L, Nunes FLS, Botega LC, Rossato DJ, Bezerra A. Virtual Reality Framework for Medical Training: implementation of a deformation class using Java. In: Proceedings of ACM International Conference on Virtual Reality Continuum and Its Applications; vol. 1. Hong Kong; 2006,.

[3] Cunha ILL. Estrutura de dados mate face e aplicações em geração e movimentos de malhas. Master Thesis; Instituto de Ciências Matemáticas e de Computação - ICMC-USP; São Paulo (SP) - Brazil - In Portuguese; 2009.

[4] Lages M, Lewiner T, Lopes H, Velho L. Che: A scalable topological data structure for triangular meshes. Tech. Rep.; 2005.

[5] Oliveira ACMTG, Nunes FLS. Building a open source framework for virtual medical training. J Digital Imaging 2010;23(6):706–20.

[6] Ferreira MIC. Estruturas de dados topológicas para variedade de dimensão 2 e 3. Master Thesis; Pontífica Universidade Católica do Rio de Janeiro - PUC-Rio; Rio de Janeiro (RJ) - Brazil - In Portuguese; 2006.

[7] Baumgart BG. A polyhedron representation for computer vision. In: Proceedings of the May 19-22, 1975, national computer conference and exposition. New York, NY, USA: ACM; 1975, p. 589–96.

[8] Guibas LJ, Stolfi J. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing. New York, NY, USA: ACM. ISBN 0-89791-099-0; 1983, p. 221–34. doi:http://doi.acm.org/10.1145/800061.808751. URL `http://doi.acm.org/10.1145/800061.808751`.

[9] Mantyla M. An Introduction to Solid Modeling. Monography - Computer Science Press; Computer Science Press; 1988.

[10] Lopes H. Algorithms to Build and Unbuild 2 and 3 dimensional manifolds. PHD Thesis; Pontífica Universidade Católica do Rio de Janeiro - PUC-Rio; Rio de Janeiro- Brazil; 1996.

[11] Campagna S, Kobbelt L, peter Seidel H. Directed edges - a scalable representation for triangle meshes. Journal of Graphics Tools 1998;3:1–11.

[12] Rossignac J, Safonova A, Szymczak A. 3d compression made simple: Edgebreaker on a corner-table. In: Shape Modeling International Conference. 2001, p. 278–83.

[13] Nonato L, Castelo A, de Oliveira M, Lizier M. Topological approach for detecting objects from images. In: Electronic Imaging 2004. International Society for Optics and Photonics; 2004, p. 62–73.

[14] Chen G, Pang M, Wang J. Calculating shortest path on edge-based data structure of graph. In: Proceedings of the Second Workshop on Digital Media and its Application in Museum & Heritage. Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-3065-6; 2007, p. 416–21. URL `http://dl.acm.org/citation.cfm?id=1333786.1333801`.

[15] Fan C. A versatile data structure schema and algorithms based on edge-symmetry. In: Information Management, Innovation Management and Industrial Engineering (ICIII), 2010 International Conference on; vol. 2. 2010, p. 519 –21.

[16] Weiler K. Edge-based data structures for solid modeling in curved-surface environments. Computer Graphics and Applications, IEEE 1985;5(1):21 –40.

[17] De Floriani L, Hui A. Data structures for simplicial complexes: an analysis and a comparison. In: Proceedings of the third Eurographics symposium on Geometry processing. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. ISBN 3-905673-24-X; 2005,URL `http://dl.acm.org/citation.cfm?id=1281920.1281940`.