Virtual Reality (2012) 16:205–213
DOI 10.1007/s10055-011-0199-5

ORIGINAL ARTICLE

A methodology for optimal voxel size computation in collision detection algorithms for virtual reality

G. Echegaray · D. Borro

Received: 10 January 2011 / Accepted: 12 September 2011 / Published online: 2 October 2011

Abstract Real-time Virtual Reality applications require accuracy but are also time dependent; therefore, in these environments, the time consumption is particularly important. For that reason, when facing the problem of Collision Detection for a Virtual Reality application, we firstly focus our attention on optimizing time performance for collisions among objects. Spatial Partitioning algorithms have been broadly used in Collision Detection. In particular, voxel-based methods are simple and quick, but finding the optimum voxel size is not trivial. We propose a methodology to easily determine the optimal voxel size for Collision Detection algorithms. Using an algorithm which represents volumetric objects with tetrahedra as an example, a performance cost function is defined in order to analytically bound the voxel size that gives the best computation times. This is made by inferring and estimating all the parameters involved. Thus, the cost function is delimited to depend only on geometric data. By doing so, it is possible to determine the optimal voxelization for any algorithm and scenario. Several solutions have been researched and compared. Experimental results with theoretical and real 3D models have validated the methodology. The reliability of our research has also been compared with traditional experimental solutions given by previous works.

Keywords Collision detection · Voxel size · Uniform spatial partitioning · Optimization

G. Echegaray (✉) · D. Borro
Department of Applied Mechanics, CEIT and Tecnun (University of Navarra), Pamplona, Navarra, Spain
e-mail: gechegaray@ceit.es

D. Borro
e-mail: dborro@ceit.es

1 Introduction

Computer graphics offer the possibility of exploring three-dimensional environments interactively. In many applications such as games and simulators, these environments can be represented with physical behavior. Simulating the real world as precisely as possible is particularly interesting for some applications, the ones which are called *virtual reality* (VR) applications.

One of the aspects that influences the behavioral reality of an environment is the way in which two objects interfere with each other. In the real world, two material objects cannot share a point in space at the same time. Therefore constraints need to be defined in virtual environments to restrict such movements. That is why it is important to detect configurations of interpenetrating objects which are called *collisions*.

Collision detection and response is a commonly investigated topic in computer graphics, due to its relevance in applications such as robotics, computational biology, games, surgery simulation, or cloth simulation. Even if the problem of detecting collisions seems purely mathematical, there are some issues which complicate matters considerably. For example, it is necessary to use the limited processing power as efficiently as possible in order to perform the queries within a concrete time frame. Moreover, points in space are represented by floats, which cause rounding errors that can result in a completely different environment behavior. Additionally, the memory consumption of these detection methods must also be considered.

We have focused our attention on optimizing time performance for the computation of collisions. VR applications require accuracy but are also time dependent; therefore, in these environments, the time consumption is particularly important.

One of the common techniques used in collision detection is based on Uniform Spatial Partitioning. These types of methods always have the problem of determining the voxel size which, being a user-defined parameter, can considerably change the performance of the method. If the number of voxels per axis is not optimized, the detection could result in execution times that are too long to use in real time.

We propose a methodology (a sequence of steps) to calculate the optimal voxel size for algorithms based on Uniform Spatial Partitioning. The main contribution of this work is a sequence of steps to infer an analytical method that automatically determines the optimal voxel size based only on the geometry of the scene. This will conclude in an easy and rapid method for improving algorithm time-rates. A previous work (Borro et al. 2004) validated the followed method with objects represented by triangle meshes. Our proposal extends that work to other types of object representations (tetrahedral meshes) and offers a general methodology for any kind of collision detection algorithm based on voxels.

Furthermore, the paper provides a set of experimental results obtained from various 3D models. The input models have been chosen varying geometry and complexity. They have been used to validate the methodology and compare our results with others proposed in the bibliography.

The remainder of the paper is organized as follows. Section 2 gives a short description of the state-of-the-art in collision detection, as well as some previous works on finding the optimal voxel size for the Spatial Subdivision method. Section 3 describes a collision detection algorithm based on tetrahedral meshes, which is later used to infer the formula and carry out experiments. Section 4 introduces the optimal voxel size problem and describes the proposed solution. In Section 5, experimental results are shown, followed by the conclusions in Sect. 6.

2 Related work

This section introduces previous research conclusions on Collision Detection Techniques for VR applications, as well as some methods used in the bibliography to deal with our concrete problem.

One of the most common applications for deformable object collision detection is the so-called *cloth simulation* (Pascal Volino 2000). It is necessary to have an efficient algorithm to handle self-collisions in order to get a dynamic cloth deformation. Another interesting application is the *surgery simulation* (Malone et al. 2010; Kockro et al. 2007; Kockro and Hwang 2009). In these environments, efficient collision detection is extremely necessary, because of their interactive behavior. Collisions for deformable organs and rigid surgical tools need to be solved, as well as tissues self-collisions in the case of topological changes as a result of cutting.

Collisions for dynamically deforming objects can be solved with different techniques, and some of them can also be used for rigid bodies. A lot of research has been done on this topic. Some surveys (Lin and Gottschalk 1998; Teschner et al. 2004) and books (van den Bergen 2004) also address the problem of collisions due to its extension.

The most common methods are based on Bounding Volume Hierarchies (Biesel and Gross 2000; Madera et al. 2010) and Spatial Subdivision (Gissler et al. 2009b). However, many other approaches like image-space techniques (Heidelberger et al. 2003), stochastic methods (Teschner et al. 2004), or distance fields (Gissler et al. 2009a) can be found in the bibliography. The latter is not too appropriate to use in real time for geometrically complex objects.

Bounding volume hierarchies subdivide objects into simpler geometries. Trees of bounding volumes are built using *Spheres* (Hubbard 1993; Bradshaw and O'Sullivan 2002), *Axis-Aligned Bounding Boxes* (van den Bergen 1997; Larsson and Akenine-Mller 2001), *Oriented Bounding Boxes* (Gottschalk et al. 1996; Schmidl et al. 2004), or *Discrete-Oriented Polytopes* (Zachmann 1998; Fnfzig and Fellner 2003). In the case of deformable bodies, the constructed hierarchy must be updated at runtime, which leads to greater time consumption. These techniques have been optimized in recent years, giving better time-rates in creating and updating the tree.

The Spatial Subdivision technique divides space instead of objects. There are many structures to be used, such as *Voxel Grids* (Gissler et al. 2009b), *Octrees* (Smith et al. 1995; Brunet and Navazo 1990), *K-d Trees* (Klosowski et al. 1998), or *Binary Space Partitioning Trees* (BSP Trees) (Melax 2000). Voxel grids offer quick access and update times in the case of objects of more or less equal size which are evenly distributed, but they have considerable storage costs. Octrees, K-d Trees, and BSP Trees are adaptive to local distribution of the objects. However, they present poor performance when objects straddle cell boundaries.

Voxel-based techniques have the problem of choosing the best voxel size to optimize the performance in terms of time consumption. The best idea would be finding a method which automatically calculates the optimal size of the voxel for any scenario. Unfortunately, such a value depends on too many factors and finding an analytical method is a difficult task. That is why most of the works that use voxel-based algorithms have solutions inferred from experimental results. For instance, García-Alonso et al. (Garcia-Alonso et al. 1994) use OBBs subdivided into voxels and have experimentally found that the optimal case is defining 512 voxels per object. Other research cases (McNeely et al. 1999) determine the discretization depending on the resolution they want to obtain. On the other hand, according to Gregory et al. (Gregory et al. 2000), the voxel size is set by multiplying the average edge length by a constant.

Our previous work (Borro et al. 2004) finds an analytical expression of the optimal voxel size for voxel grids; nevertheless, it is focused on the collision detection problem with triangle meshes. Other research (Teschner et al. 2003) has determined that optimal performance is achieved when a grid cell has the same size as the average edge length of all tetrahedra. We will compare our solution with this assertion.

3 Description of the collision detection algorithm

The collision detection algorithm we have used to infer the method is a particular case where objects are represented by tetrahedral meshes. In this approach, the collision problem is solved using a uniform spatial subdivision. This technique subdivides space into *cells*, which are introduced into a vector depending on their position in the world (Teschner et al. 2003). Using voxels, cell access is fast, since the calculation of the cell is made in constant time:

$$cell = floor\left(\frac{p + world}{cell\_size}\right)$$

where p is a point and *world* comprises the size of the scenario in the three axes.

Once we have the indices for each cell, we just need to use a simple function to locate them in a vector:

$$id = k \cdot numCellsX \cdot numCellsY + j \cdot numCellsX + i$$

Most works based on voxels use hashing techniques in order to reduce memory consumption. The chosen hashing function can change the performance of the method. Values given by this function should be uniformly distributed to achieve quality performance. The hash table size also influences, as the use of big tables reduces the possibility of placing several different cells to the same position. Therefore, the algorithm usually works faster. On the other hand, the memory consumption is higher. Some studies have been done on this topic (Teschner et al. 2003), which have found that if the hash table is significantly larger than the number of primitives, the risk of hash collisions is minimal. Besides, hash functions work most efficiently if the hash table size is a prime number.

Using a voxel grid has the benefits of low storage usage (using hashing) and fast cell access. However, the biggest drawback of this structure is the fact that performance depends on the chosen voxel size. We propose an analytical method to compute the voxel size that minimizes the processing time of the algorithm. This topic will be discussed in Sect. 4. The diagram in Fig. 1 describes the collision detection algorithm we have used.

In a pre-process block, the space is subdivided according to the chosen cell size. Afterwards at runtime, this grid is used to determine colliding voxels as well as checking primitives for intersection. These two steps are known as *broad phase* and *narrow phase*, respectively, according to the nomenclature of Hubbard (Hubbard 1993). Once collisions have been detected, a collision response is performed. This provides models with the convenient deformations which are calculated using a specific integration model.

The objects in the scene are represented by vertices (particles) and tetrahedra. From now on, we will use *particle* and *vertex* without distinction. Once the voxel grid has been created, all the particles of a tetramesh are classified with respect to the cell in which they are placed. Afterward, each cell has a list of references to particles contained in that cell. These lists will be modified in every single frame due to the mobility of the objects.

As mentioned before, the runtime block can be divided into two phases: the broad phase and the narrow phase.

In a first pass, all the particles in the scene are classified once again in the voxel grid. At the same time, the bounding boxes of the tetrahedra are recomputed based on their current deformed state.

In a second pass, the algorithm of (Teschner et al. 2003) is executed.

- *Broad Phase*: For each tetrahedra t of one input object, we get its bounding box $AABB_t$. Then, we identify the set voxels in the scene where $AABB_t$ lies. These voxels are called *objective voxels* (v_o). This allows the delimitation of the whole problem to a smaller set of data. For each voxel in v_o, the particles belonging to the second object which lie inside it are analyzed. Each one of them is checked for intersection with $AABB_t$. When a particle p is inside $AABB_t$, the second step (narrow phase) is performed.
- *Narrow Phase*: This step checks whether p intersects t or not. This is done using the barycentric coordinates of the vertex p with respect to the tetrahedron (Teschner et al. 2003).

The lists of the particles occurring in a cell need to be updated for every single object transformation. As has been shown before, using this structure, there are a lot of primitives and particle pairs which can be rejected from intersection testing.

After detecting collisions between primitives, the collision response is performed. For this task, we have used the

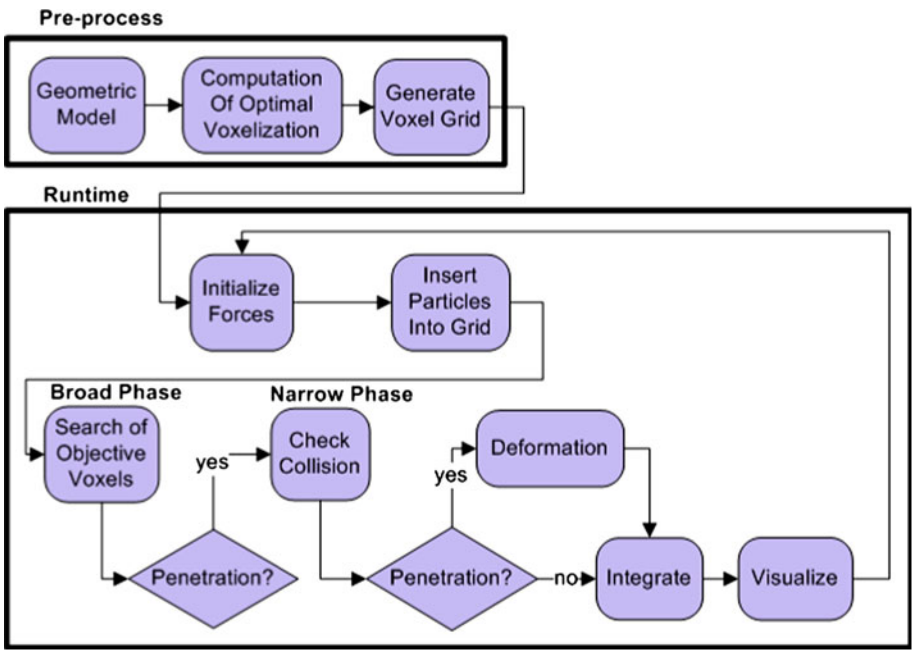

Fig. 1 General design of the algorithm

penalty method based on the penetration depth of intersecting objects. Even if there are many methods to estimate the penetration depth, the one used here computes a consistent penetration depth (Heidelberger et al. 2004), which reduces collision response artifacts inherent to existing approaches. Moreover, a propagation scheme is introduced (Heidelberger et al. 2004) to approximate the penetration depth and direction for vertices with deep penetrations. This procedure will not be described in detail as the goal of this paper is optimizing the detection algorithm (next Section), not the response.

4 Analytical computation of the optimal voxel size

The idea of the Spatial Subdivision is to divide the space into convex regions called *cells*. These are determined by an identifier (calculated according to the coordinates of the cell) and are placed into a vector, in order to classify the interacting objects and determine relations between them.

The use of spatial partitioning techniques involves a basic initial problem: the 3D space is discretized with respect to a user-defined cell size, and the choice of that value can significantly change the efficiency of the algorithm. If the number of cells per axis is not optimal, the detection process may be too long to use in real time. Of course, this value changes depending on the scenario.

The optimal voxel size is the one which enables minimal calculation time cost in the collision detection algorithm. This will be delimited by these two events:

1. If cells are very large compared with the size of the model's tetrahedra, the number of primitives per cell increases, which involves a large amount of point-tetrahedron tests.
2. On the other hand, small cells cause a spreading of primitives to more cells. This will result in a lot of voxel-tetrahedron intersection tests.

What has been done is to extend our previous experience in this topic to generalize a methodology valid not only for scenes of triangle meshes but also for scenes with other types of object representations.

In the next section, the parameters which are supposed to affect the collision detection time in an instant are identified. Then, we will show how to refine this definition of parameters.

4.1 Methodology to infer the cost function

This section describes the methodology to obtain an analytical solution for optimizing the voxelization.

The *cost function* of the algorithm is the mathematical expression that describes its behavior, namely, it predicts the time it will take to solve a collision problem in a determined scenario.

The factors that could be involved a priori in the analytical formula are the following: the number of polygons in the scene, the average of their areas, the average length of the edges, and the average volume of the tetrahedra. Location of the objects and the contact volume could also affect the cost function of the algorithm.

If the cost function is constructed dependent on the voxel size, it is enough to find the value of this variable which makes the function minimum. What we present is an adaptation of the approach we used in our previous work (Borro et al. 2004) for the new algorithm presented in Sect. 3, proving that the methodology might be extended to other collision detection algorithms where objects are not represented by triangles. We will take the collision detection algorithm, infer the cost function of its behavior, and estimate all the parameters of the cost function using only geometrical data from the scenario.

As shown in a previous section, the collision detection algorithm has two levels of precision: voxels and tetrahedra. Each tetrahedron of the object goes through both levels in case of collision. So if t_o is the number of tetrahedra of one object, v_o (objective voxels) is the average number of voxels covered by the bounding box of a tetrahedron, and p_v is the average number of affected particles per voxel, the algorithm behavior could be expressed as in formula (1) (remember that the algorithm has three loops: one works through each tetrahedron of the object, another one through the objective voxels of each tetrahedron, and the last one through the particles of each objective voxel):

$$z = t_o \cdot v_o \cdot p_v \tag{1}$$

This formula only considers the worst case: when one whole object is colliding with another. In a normal case, this does not happen, as the collision response algorithm does not let the objects penetrate each other to such a degree. A typical simulation does not have collisions, or at most a few tetrahedra intersect. That is why z should be separated into two cases:

$$z = t_o \cdot p_{nc} \cdot v_o + t_o \cdot (1 - p_{nc}) \cdot v_o \cdot p_v \tag{2}$$

where p_{nc} is the probability of no collision for each tetrahedron. Experiments made with different scenarios and objects have shown that this probability can always be bounded between 0.98 and 0.99 when the physically based collision response algorithm is implemented (our experiments have used $p_{nc} = 0.99$). This means that the first term of (2) will have much more weight than the second one, as happens in a real simulation where only a few tetrahedra are in collision.

Even if the formula in (2) gives the same weight to the parameters v_o and p_v, this could be refined. The time it takes to determine whether a voxel is empty of particles (t_v) is much lower than checking whether a particle collides with a tetrahedron or not (t_p). For that reason, we have defined a *ratio* r_t between particle and voxel times. It is defined as $r_t = t_v/t_p$. According to our experiments, checking particles for intersection is 5 times more expensive than the voxels time.

Apart from that, t_o is a constant value (it does not depend on the voxel size) and we do not need it to compute the minimum of z. So our z can be written as follows:

$$z = p_{nc} \cdot v_o + (1 - p_{nc}) \cdot v_o \cdot p_v \cdot r_t \tag{3}$$

Once the behavior of the algorithm is established via the cost function, an expression of the parameters v_o and p_v needs to be found in order to associate them with the voxel size (the variable we want to solve) and the geometry of the scene (the only data we have).

4.2 Cost function parameters

This section presents an expression of the cost function which obtains a voxel size (δ) that minimizes the function value.

For a specific voxelization and a scene, v_o and p_v can be directly measured. Nevertheless, if we want to replace them in the expression of the cost function, they need to be approximated as a function of δ and the geometry data. Basically, we modify the derived expressions of Borro et al. (2004) to work with tetrahedra, instead of triangle meshes.

We will deduce the expressions for both parameters in the next lines:

- v_o: As previously stipulated, this is the average number of voxels contained within the bounding box of a tetrahedron t, specifically the ones intersected by its AABB. So in order to express v_o, we first need to estimate the dimensions of the bounding box. In a simpler case, a way of constructing a box that always covers a triangle is by choosing the one in which the side is equal to the largest side of the triangle. So in our case, as the tetrahedra can be in any space orientation, in the worst case the AABB involving a tetrahedron will be a cube in which the side, s_{AABB}, is the largest edge of the tetrahedron.

Once we have the dimensions of the AABB, and based on the voxel size, it is possible to predict the number of voxels intersected by the bounding box in the worst case.

$$v_o = \left\lfloor \left(\frac{s_{AABB}}{\delta} + 2 \right) \right\rfloor^3 \tag{4}$$

- p_v: The number of particles inside a voxel can be approximated finding the average number of triangles contained in the voxel. This could be done with different methods. We use the area of the tetrahedra facets (triangles) to approximate the maximum number of facets per voxel. As we use triangle areas, we can take the same approach of (Borro et al. 2004) to compute this parameter. Taking into account that the biggest triangle a voxel can contain is the one shown in Fig. 2.

The area of that triangle (A_{vox}) is

$$A_{\mathrm{vox}} = \frac{\delta^2\sqrt{3}}{2} \tag{5}$$

Being A_{t} the real average area of the scene triangles, the number of facets per voxel (f_{v}) can be easily predicted:

$$f_{\mathrm{v}} = \frac{A_{\mathrm{vox}}}{A_{\mathrm{t}}} = \frac{\delta^2\sqrt{3}}{2A_{\mathrm{t}}} \tag{6}$$

Once we estimate the number of triangles contained in a voxel and if each triangle joins three particles, the number of particles per voxel is:

$$p_{\mathrm{v}} = 3 \cdot \frac{\delta^2\sqrt{3}}{2A_{\mathrm{t}}} \tag{7}$$

We already have all we need to get the optimal value of the cost function seen in (3): v_{o} and p_{v} have been predicted in this section, and p_{nc} and r_{t} are constant numbers. Then, the cost function z only depends on the voxel size δ. Therefore, we will obtain the optimal voxel size by minimizing the function. This optimal value will be calculated in the pre-process stage and the time consumption for that work is obviously not significant. This can be done by solving the following equation:

$$\frac{dz}{d\delta} = 0 \tag{8}$$

5 Experimental results

Using a software that runs a set of simulations with different voxelizations, it is possible to do several experiments in order to get their execution times and determine the optimal zone experimentally. That will be useful to validate our analytical solution. Approximating the times obtained in the set of simulations with a polynomial function $f(x)$ will allow us to experimentally determine the optimal value and the *optimal zone*. We understand as *optimal zone* the range of voxel sizes that give the smallest computation times for the algorithm: smaller or larger levels of voxelization than the ones in the optimal zone lead to greater frame rates.

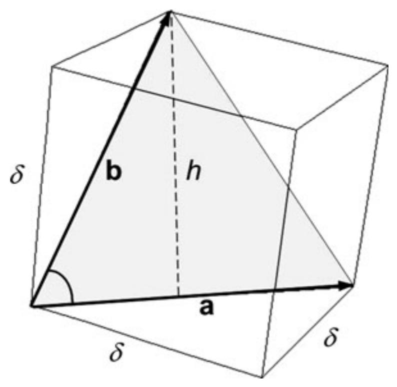

Fig. 2 Largest triangle contained in a voxel

The minimum of the polynomial function used for the approximation is considered the experimental optimal value. The optimal zone involves the values close to the optimal value, which have been delimited as the ones which do not raise the optimal value more than 1 millisecond.

We have performed two types of experiments: tests with simple and theoretical objects and tests using some standard 3D models. The primitive used in the simple tests is the sphere. Although in real simulations complex models are always going to be used, these first experiments can gather information about the method behavior. Table 1 shows the number of tetrahedra of the simple models used to carry out the experiments.

The next lines list some criteria used throughout the entire experiment.

- The developed software automatically runs an analysis for different voxel sizes. For each voxelization level, the same procedure is applied: two identical objects are placed into a position where they collide with each other. Then, necessary forces are applied to the objects (collision response, gravitational forces and so forth) and the simulation runs for 10 frames. Times for those 10 frames are used to calculate the average computation time.
- Analyzed positions have not been chosen randomly. They have been selected in order to have an average number of 1% of the tetrahedra of the scene in collision. As the implemented collision response algorithm does not let the objects penetrate significantly into each other, positions in which more than the 1% of the tetrahedra are in collision will never happen. This value has been chosen after performing some experiments with real models in order to check the average number of colliding tetrahedra.

Table 1 Description of the models

Model	Tetrahedra
Sphere01	41937
Sphere02	52326
Sphere03	60688
Sphere04	72631
Sphere05	80927

Figure 3 shows the results following the steps in the experiments. This one concretely belongs to a scene with two Sphere01 objects. It shows the average times of the experiments carried out with these two objects when using different voxel sizes. Although the cost function of the previous section depends on the voxel size (δ), in graphs and tables the number of voxels per axis (derived from δ) is used, considering that this value is easier to evaluate for the reader. For that reason, the given x values are the number of voxels per axis. The size of the scenario is $4 \times 4 \times 4$. The curve has been approximated with the polynomial function that best fits it. An *optimal zone* [δ_{min}, δ_{max}] is defined by taking an interval around the *optimal value* δ_{opt}.

Each experiment (δ_{min}, δ_{opt}, δ_{max}) is compared with two values: the first one is the voxel size guessed by our analytical formulation (δ_a). The result is also compared with one traditional solution (δ_{trad}) based on the average length of the edges (Teschner et al. 2003). Table 2 gives the results of experiments with different objects.

Note that the optimal value according to our formula lies inside the minimum and maximum experimental values. On the other hand, values given by other solutions differ considerably from the real experimental optimum. On some occasions, they do not even lie within the optimal zone.

In order to make it easier to see the difference between experimental and traditional solutions as well as the analytical ones derived from our research, a relative error ϵ_δ has been used as in (Borro et al. 2004).

$$\epsilon_\delta = \frac{|\delta_{opt} - \delta| \cdot \left(\left\lfloor \frac{\delta_{opt}}{\delta} \right\rfloor + \left\lfloor \frac{\delta}{\delta_{opt}} \right\rfloor \right)}{(\delta_{opt} - \delta_{min}) \cdot \left\lfloor \frac{\delta_{opt}}{\delta} \right\rfloor + (\delta_{max} - \delta_{opt}) \cdot \left\lfloor \frac{\delta}{\delta_{opt}} \right\rfloor} \qquad (9)$$

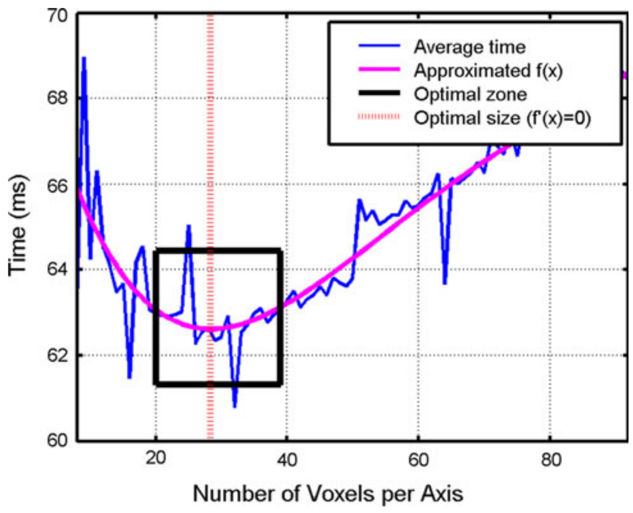

Fig. 3 Experimental average times and approximation of the curve for two Sphere01 models colliding

Table 2 Optimal experimental, our analytical and traditional values for each object

	Sphere01	Sphere02	Sphere03	Sphere04	Sphere05
δ_a	28.73	30.94	32.49	34.48	35.65
δ_{min}	20	22	16	8	20
δ_{opt}	28.33	28.16	34.04	26.63	37.87
δ_{max}	39	38	54	47	57
δ_{trad}	41.90	45.16	47.40	50.29	51.90

Table 3 Error ϵ_δ for both our analytical value (δ_a) and values given by traditional believes (δ_{trad})

Model	ϵ_{δ_a}	$\epsilon_{\delta_{trad}}$
Sphere01	***0.038***	1.272
Sphere02	***0.283***	1.727
Sphere03	***0.086***	***0.669***
Sphere04	***0.385***	1.162
Sphere05	***0.124***	***0.734***

Values in bold italics show numbers between 0 and 1, which are satisfactory results

If the error is zero, it means that δ matches the optimal size found experimentally (δ_{opt}). If $0 < \epsilon_\delta \leq 1$, δ is not exactly the optimum but it lies inside the optimal zone so we consider it as a good approximation.

Table 3 shows the relative errors for two cases: the first column is the error of the approximation made by our formula (based on the average area of the triangles that are part of the tetrahedra), respect to the optimal value obtained experimentally. The second column is the error for the voxel sizes used in traditional solutions (based on the average length of the edges), with respect to the optimal experimental value as well.

As it can be seen, our values are all inside the optimal zone ($0 < \epsilon_{\delta_a} \leq 1$), very close to the experimental optimal voxelization levels. On the other hand, other solutions differ widely from the optimum and in some cases are even outside the optimal zone. This could result in an important increase of the computation time. We have also made approximations based on other geometrical data like the average volume of the tetrahedra. Nevertheless we have not included the results in this paper since the results based on areas are much more accurate.

Once experiments with theoretical models have been shown, results obtained for real models are given in the following figures and tables. Figure 4, Tables 4, and 5 show the complexity of the real models, optimal values, and computed ϵ_δ, respectively.

Once again, the optimal cell size calculated by our method lies inside the optimal zone obtained experimentally. Nonetheless, solutions given by traditional methods are close but do not always belong to the optimal zone.

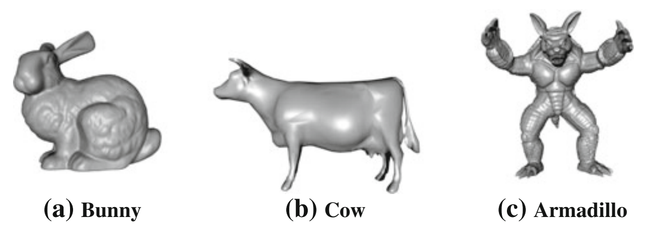

Fig. 4 3D Models used in the experiments: **a** Bunny (20462 tetrahedra), **b** Cow (50380 tetrahedra), **c** Armadillo (16377 tetrahedra)

Table 4 Optimal experimental, analytical, and traditional values for each object

	Bunny	Cow	Armadillo
δ_a	31.35	48.80	31.55
δ_{min}	16	13	13
δ_{opt}	22.97	31.15	20.49
δ_{max}	34	54	37
δ_{trad}	44.94	70.93	45.48

Table 5 Error ϵ_δ for both our analytical value (δ_a) and values given by traditional methods (δ_{trad})

Model	ϵ_{δ_a}	$\epsilon_{\delta_{trad}}$
Bunny	***0.759***	1.992
Cow	***0.772***	1.741
Armadillo	***0.669***	1.516

Values in bold italics show numbers between 0 and 1, which are satisfactory results

Even if these previous methods give an approximation of the optimal voxel size and may be valid for some types of applications, our new approach gets much closer to the exact optimal value.

6 Conclusions

VR refers to computer-generated environments that can simulate physical conditions. One particularly important aspect that influences the realism of VR is the problem of real-time interactive Collision Detection. Time consumption is particularly important in such environments. For that reason, we have presented the concrete problem of optimizing collision detection algorithms based on Uniform Spatial Subdivision. This is made by determining the optimal voxel size. We have studied the difficulties of the approach and proposed a methodology to optimize the selection of the best voxel size. Thus, starting from a Spatial Partitioning collision detection algorithm, we describe the steps to follow, in order to easily estimate the most appropriate cell size to optimize the behavior of the algorithm in any scenario.

The voxel size chosen in the past was usually selected experimentally. Some studies give an approximation of the best voxelization by carrying out experiments, which lead to a method for delimiting a suitable voxel size based on the geometry of the scene. These solutions, however, came quite close to the optimum, but were not always as accurate as some applications demanded. Our proposal gives a general methodology to infer an analytical solution based on a cost function and approximates the optimal size satisfactorily. It has also been validated experimentally and compared with other approaches.

We have presented a particular case to validate the methodology with objects represented by tetrahedra. As said before, our previous work validated the method for triangle meshes, so with these two studies we involve most of the representations used in VR applications. Even if the particular solution presented here is dependent on a concrete algorithm, the cost function can be easily obtained from other algorithms in the same way so the same method can be applied to any algorithm.

Acknowledgments This work has been partially funded by the Government of Navarra within the *Plan de Formacion y de Investigacion y Desarrollo* program that belongs to the *Education* department. The authors are also grateful to Prof. Dr. Matthias Teschner (Computer Graphics Laboratory, University of Freiburg, Germany) for his support to carry out the collision detection algorithm, as well as his suggestions and comments on the subsequent study.

References

Biesel D, Gross M (2000) Interactive simulation of surgical cuts. In: Proceedings of the 8th pacific conference on computer graphics and applications, PG '00. IEEE Computer Society, Washington, DC, USA, pp 116–125

Borro D, García-Alonso A, Matey L (2004) Approximation of optimal voxel size for collision detection in maintainability simulations within massive virtual environments. Comput Graph Forum 23:13–24

Bradshaw G, O'Sullivan C (2002) Sphere-tree construction using dynamic medial axis approximation. In: 2002 ACM SIGGRAPH/Eurographics symposium on computer animation, SCA '02. ACM, New York, NY, USA, pp 33–40

Brunet P, Navazo I (1990) Solid representation and operation using extended octrees. ACM Trans Graph 9(2):170–197

Fünfzig C, Fellner DW (2003) Easy realignment of k-dop bounding volumes. In: Graphics Interface. A K Peters, Halifax, Nova Scotia, pp 257–264

Garcia-Alonso A, Serrano N, Flaquer J (1994) Solving the collision detection problem. IEEE Comput Graph Appl 14:36–43

Gissler M, Dornhege C, Nebel B, Teschner M (2009a) Deformable proximity queries and their application in mobile manipulation planning. In: Proceedings of the 5th international symposium on advances in visual computing: Part I, ISVC '09. Springer, Heidelberg, pp 79–88

Gissler M, Schmedding R, Teschner M (2009b) Time-critical collision handling for deformable modeling. Compu Animat Virtual Worlds 20(2–3):355–364

Gottschalk S, Lin MC, Manocha D (1996) Obbtree: a hierarchical structure for rapid interference detection. In: ACM Siggraph., volume 30. University of North Carolina, NC, pp 171–180

Gregory A, Lin MC, Gottschalk S, Taylor R (2000) Fast and accurate collision detection for haptic interaction using a three degree-of-freedom force-feedback device. Comput Geom 15:69–89

Heidelberger B, Teschner M, Gross M (2003) Real-time volumetric intersections of deforming objects. In: VMV'03. Akademische Verlagsgesellschaft Aka GmbH, München, Germany, 461–468

Heidelberger B, Teschner M, Keiser R, Müller M, Gross M (2004) Consistent penetration depth estimation for deformable collision response. In VMV'04. Akademische Verlagsgesellschaft Aka, Stanford, USA

Hubbard P (1993) Interactive collision detection. In: IEEE 1993 symposium on research frontiers in virtual reality, pp 24–31

Klosowski JT, Held M, Mitchell JSB, Sowizral H, Zikan K (1998) Efficient collision detection using bounding volume hierarchies of k-dops. IEEE Trans Vis Comput Graph 4(1):21–36

Kockro R, Hwang P (2009) Virtual temporal bone: an interactive 3-dimensional learning aid for cranial base surgery. Neurosurg Clin N Am 64(5 Suppl 2):216–229

Kockro RA, Stadie A, Schwandt E, Reisch R, Charalampaki C, Ng I, Yeo TT, Hwang P, Serra L, Perneczky A (2007) A collaborative virtual reality environment for neurosurgical planning and training. Neurosurgery, 61(5 Suppl 2):379–91; discussion 391

Larsson T, Akenine-Möller T (2001) Collision detection for continuously deforming bodies. In: Eurographics 2001, short presentations. Eurographics Association, Blackwell Publishing, pp 325–333

Lin MC, Gottschalk S (1998) Collision detection between geometric models: A survey. In: Proceedings of IMA conference on mathematics of surfaces, pp 37–56

Madera FA, Laycock SD, Day AM (2010) Detecting self-collisions using a hybrid bounding volume algorithm. In: Proceedings of the 2010 third international conference on advances in computer-human interactions, ACHI '10. IEEE Computer Society, Washington, DC, USA, pp 107–112

Malone HR, Syed ON, Downes MS, D'Ambrosio AL, Quest DO, Kaiser MG (2010) Simulation in neurosurgery: a review of computer-based simulation environments and their surgical applications. Neurosurg Clin N Am 67(4):1105–1116

McNeely WA, Puterbaugh KD, Troy JJ (1999) Six degree-of-freedom haptic rendering using voxel sampling. In: 26th annual conference on computer graphics and interactive techniques, SIGGRAPH '99. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, pp 401–408

Melax S (2000) Dynamic plane shifting bsp traversal. In: Graphics Interface. Lawrence Erlbaum Associates, pp 213–220

Pascal Volino NMT (2000) Implementing fast cloth simulation with collision response. In: International conference on computer graphics, CGI '00. IEEE Computer Society, Washington, DC, USA, p 257

Schmidl H, Walker N, Lin MC (2004) Cab: Fast update of obb trees for collision detection between articulated bodies. J Grap GPU, and Game Tools 9(2):1–9

Smith A, Kitamura Y, Takemura H, Kishino F (1995) A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. In: Proceedings of the virtual reality annual international symposium (VRAIS'95), VRAIS '95. IEEE Computer Society, Washington, DC, USA, p 136

Teschner M, Heidelberger B, Mueller M, Pomeranets D, Gross M (2003) Optimized spatial hashing for collision detection of deformable objects. In: VMV (vision, modeling and visualization). Aka GmbH, Munich, Germany, pp 47–54

Teschner M, Kimmerle S, Heidelberger B, Zachmann G, Raghupathi L, Fuhrmann A, Cani M-P, Faure F, Magnenat-Thalmann N, Strasser W, Volino P (2004) Collision detection for deformable objects. In: EUROGRAPHICS, vol 23. pp 119–139

van den Bergen G (1997) Efficient collision detection of complex deformable models using aabb trees. J Graph Tools 2(4):1–13

van den Bergen G (2004) Collision Detection in Interactive 3D Environments. Morgan Kaufmann, Los Altos, CA

Zachmann G (1998) Rapid collision detection by dynamically aligned dop-trees. In: Proceedings of the virtual reality annual international symposium, VRAIS '98. IEEE computer society, Washington, DC, USA, p 90