Applied Soft Computing 10 (2010) 225–235

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

A comparative study of partitioning methods for crowd simulations

G. Vigueras *, M. Lozano, J.M. Orduña, F. Grimaldo

Departamento de Informática, Universidad de Valencia, Burjassot, Valencia, Spain

ARTICLE INFO

Article history:
Received 20 December 2008
Received in revised form 1 July 2009
Accepted 5 July 2009
Available online 22 July 2009

Keywords:
Crowd simulation
Partitioning method
Load balancing

ABSTRACT

The simulation of large crowds of autonomous agents with realistic behavior is still a challenge for several computer research communities. In order to handle large crowds, some scalable architectures have been proposed. Nevertheless, the effective use of distributed systems requires the use of partitioning methods that can properly distribute the workload generated by agents among the existing distributed resources.

In this paper, we analyze the use of irregular shape regions (convex hulls) for solving the partitioning problem. We have compared a partitioning method based on convex hulls with two techniques that use rectangular regions. The performance evaluation results show that the convex hull method outperforms the rest of the considered methods in terms of both fitness function values and execution times, regardless of the movement pattern followed by the agents. These results show that the shape of the regions in the partition can improve the performance of the partitioning method, rather than the heuristic method used.

1. Introduction

The simulation of large crowds of autonomous agents has become an essential tool for many virtual environment applications in education, training, and entertainment [2,23,9,18,10]. Crowd simulations model the motion of crowds and other flock-like groups as interacting particles that display different behaviors in 2D/3D scenes [20,4]. Agent-based crowd simulations aim to capture the nature of a crowd as a collection of individuals, each of which can have their own goals, knowledge and behaviors [19]. These applications require both rendering visually plausible images of the virtual world and managing the behavior of autonomous agents at interactive rates. In this sense, simulating the realistic behavior of large crowds of autonomous agents is still a challenge for several computer research communities.

The sum of graphical quality and realistic behavior requirements results in a computational cost that highly increases with the numbers of agents in the system, requiring a scalable design that can handle simulations of large crowds in a feasible way. In this sense, some proposals tackle crowd simulations as a particle system with different levels of details [3,24] and other proposals focus on providing efficient and autonomous behaviors to crowd simulations [15,7].

In order to achieve the required scalability, the crowd should be distributed among different processors. In previous works, we proposed an architecture that can simulate large crowds of autonomous agents at interactive rates [11,26] where the simulation world was partitioned into subregions and each one assigned to one parallel server. A scheme of the architecture presented in [26] is shown in Fig. 1. This figure shows how the space occupied by crowd agents is partitioned into three subregions, and each one is assigned to one Action Server (AS, labeled in the figure as AS_x). In turn, each AS is hosted by one computer. The agents are execution threads assigned to one Client Computer (labeled in the figure as $Client_x$) associated to the corresponding server, AS_x. Each AS process hosts a copy of the Semantic Database. However, each AS exclusively manages the portion of the database representing the agents in its region. In order to guarantee the action consistency near the border of the different regions (see $agent_k$ in Fig. 1), the ASs can collect information about the surrounding regions by querying the servers managing the adjacent regions. Additionally, the associated Clients are notified about the changes produced by the agents located near the adjacent regions by the ASs managing those regions. This architecture allows to scale up the system, although it also requires an efficient partitioning method. This partitioning method should efficiently assign the agents to the existing servers in such a way that the number of messages exchanged among the servers is reduced and the system is well balanced.

In order to reduce the overhead imposed by the partitioning method for crowd simulations, in this paper we analyze the use of irregular shape regions (convex hulls) for solving the partitioning problem. We have compared a partitioning method based on

* Corresponding author. Tel.: +34 963544489.
E-mail addresses: Guillermo.Vigueras@uv.es (G. Vigueras), Miguel.Lozano@uv.es (M. Lozano), Juan.Orduna@uv.es (J.M. Orduña), Francisco.Grimaldo@uv.es (F. Grimaldo).

1568-4946/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2009.07.004

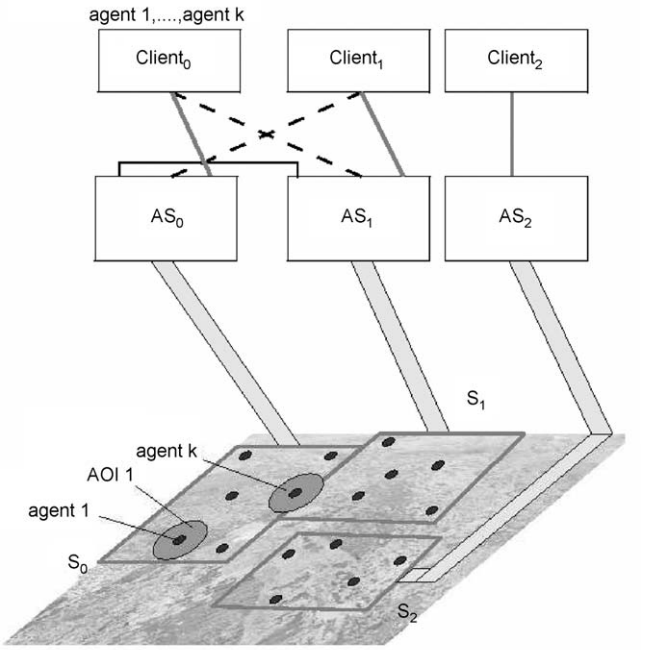

Fig. 1. General scheme of the distributed crowd architecture.

convex hulls with two techniques that use rectangular regions. One of them uses a heuristic search method (GA) and the other one uses an algorithmic method (R-Tree). The performance evaluation results show that the method based on convex hulls outperforms the rest of the considered methods in terms of both fitness function values and execution times, regardless of the movement pattern followed by the agents. As a result, this method provides better partitions than the other methods while requiring shorter execution times. These results indicate that the shape of the regions in the partition has a major influence on the performance of the partitioning method, rather than the search method used.

The rest of the paper is organized as follows: Section 2 shows some related work about crowd simulations and partitioning methods. Section 3 explains the metrics used for comparing the performance of the considered partitioning methods. Next, Section 4 describes different region-based partitioning methods (some of them using rectangular regions and one of them using convex hulls) and how they have been implemented for partitioning distributed crowd simulations. Section 5 presents the performance evaluation of the considered methods. Finally, Section 6 presents some concluding remarks and future work to be done.

2. Related work

The sum of graphical quality and realistic behavior requirements in crowd simulations results in a computational cost that highly increases with the numbers of agents in the system. Therefore, crowd simulations require a scalable design that can handle simulations of large crowds in a feasible way. In this sense, some proposals tackle crowd simulations as a particle system with different levels of details (e.g.: *impostors*) in order to reduce the computational cost [3,24]. Although these proposals can handle crowd dynamics and display populated interactive scenes (10,000 virtual humans), they are not able to produce complex autonomous behaviors for their actors. Other proposals focus on providing efficient and autonomous behaviors to crowd simulations [15,7]. However, they are based on a centralized system architecture, and they can only control a few hundreds of autonomous agents with different skills (pedestrians with navigation and/or social behaviors for urban/evacuation contexts). Taking into account that pedestrians represent the slowest human actors (in contrast to other kind of actors like drivers in cars, for example) these results show that scalability has still to be solved in multi-agent crowd simulations. In order to address this problem, we proposed a distributed architecture for crowd simulation [11,26]. In that architecture, the crowd system is composed of many Client Computers, that host agents, and one Action Server (AS), that is responsible for checking the actions (e.g. collision detection) sent by agents [11]. Later, that architecture was improved by parallelizing the AS in a distributed-server fashion [26], in such a way that the simulation world was partitioned into subregions and each one assigned to one parallel AS, as shown in Fig. 1. However, any distributed architecture requires an efficient partitioning method that efficiently assign the agents to the existing servers in such a way that the number of messages exchanged among the servers is reduced and the system is well balanced.

Typically, there are two different approaches for partitioning a crowd simulation. One of them is based on the criterion of workload [22,14], so that different groups of agents are executed in different computers. The other approach is region-based, in such a way that the virtual world is split into regions (usually a 2D cell from a grid) and all the agents located at a given region are assigned to a given computer [17]. Both approaches should guarantee the consistency of the simulation (for example, two different agents cannot be located at the same point in the virtual world). The most appropriate approach for the architecture shown Fig. 1 is the region-based approach, because each Action Server manages a region of the virtual world.

The region-based partitioning problem for crowd simulation has been previously addressed. A representative proposal has achieved that a PLAYSTATION-3 (IBM Cell Engine processor) can display a crowd composed of 15,000-fishes at 60 frames per second [18]. This work incorporates spatial hashing techniques and it also distributes the load among the Cell Engine Synergistic Processor Elements (SPEs) [8]. The same social forces model has been also integrated in a PC-Cluster with MPI communications among the processors, although the number of simulated agents is still low (512 agents) and the execution times are far from interactive [27]. Another work describes the use of a multicomputer with 11 processors to simulate a crowd of 10,000 agents at interactive rates [17]. However, they use static agent-processor assignment, and no workload balancing is provided. A different proposal uses Genetic Algorithms (GA) for partitioning crowd simulations, in order to improve the partitioning efficiency [12]. However, all these region-based partitioning methods add a significant overhead to the system, and this overhead should be limited as much as possible in order to provide scalable partitioning methods. Also, there are purely graphic approaches [16,25] that are not concerned with scalability problems because they are not focused on managing the behavior of a high number of autonomous agents.

3. Methodology

The partitioning problem consists of finding a near-optimal partition of regions (containing all the agents in the system) that simultaneously fulfills two conditions: it minimizes the number of agents near the borders of the regions, and it properly balances the number of agents in each region too. The first element required for comparing different partitioning techniques is to define a homogeneous criterion for measuring the quality of the partitions provided by all the methods.

In order to achieve this goal, we have defined the following fitness function to be minimized [12]:

$$H(P) = \omega_1 \cdot \alpha(P) + \omega_2 \cdot \beta(P), \quad \omega_1 + \omega_2 = 1 \tag{1}$$

The first term in this equation measures the number of border agents in the resulting partition P (those agents whose surroundings (Area Of Interest or AOI [21]) crosses the region boundaries). Since the management of the border agents should be performed by two or more servers, the workload generated by these agents is higher that the one generated by those agents located far from the region borders (in order to check the action of a border agent, each server should send a locking request to the other servers managing the AOI of that agent. These locking requests allow to maintain the consistency of the virtual world in the border areas, but they involve several servers, requiring a much higher computational cost). Therefore, the number of border agents must be minimized. It should be noticed that those agents located in overlapped regions can be considered also border agents, since they should be managed by more than one server. That is, the resulting partition P should contain regions with the minimum area of overlapping. Concretely, $\alpha(P)$ is computed as the sum of all the agents whose AOIs intersect two or more regions of the virtual world (see ag_k in Fig. 1). $\beta(P)$ is computed as the standard deviation of the average number of agents that each region contains. Therefore, $\beta(P)$ measures how balanced the partition P is. Finally, ω_1 and ω_2 are weighting factors between 0 and 1 that can be tuned to change the behavior of the search as needed. Although only the heuristic method uses this function for guiding the search, for comparison purposes we have used $H(P)$ as the global fitness function for measuring the quality of the partitions provided by all the considered methods. The reason is that in order to make a fair comparison in the performance evaluation section (Section 5), the same fitness function must be used for all the methods.

The results in this paper correspond to weights ω_1 and ω_2 set to 0.6 and 0.4, respectively. We have set these values because we have empirically observed that parallel ASs are significantly affected by the number of locking requests among servers (these requests are used for guaranteeing the exclusive access of a single server to those agents in overlapping regions, that is, the $\alpha(P)$ values). However, avoiding the ASs saturation (achieving balanced partitions, that is, good $\beta(P)$ values) is also crucial for DVE-like systems [14]. Therefore, when evaluating the fitness function for a partition, the importance given to the number of border agents is slightly greater than the one given to the load balancing, a criterion that makes sense in the distributed system showed in Fig. 1.

All the methods considered in this paper initially use the k-means algorithm to obtain the initial partition. Once the simulation starts, the partition should be adapted to the current state of the crowd every server cycle. During the simulation each server knows the location of the agents in its region and also the number of agents and the mass center of the region assigned to its neighbor servers. While the heuristic method uses a Genetic Algorithm (GA) guided by $H(P)$ to search a near-optimal partition of rectangular regions, the other two methods use spatial clustering techniques to provide a near-optimal partition. In the latter cases, the servers periodically assign each of their agents ag_k to the server controlling the region r_i that minimizes the following function:

$$f_{\text{alloc}}(ag_k, r_i) = dstMC(ag_k, r_i) + nAgs(r_i) * dstMC(ag_k, r_i) \quad (2)$$

where f_{alloc} is the allocation function, $nAgs(r_i)$ provides the number of agents in region r_i, and $dstMC(ag_k, r_i)$ corresponds to the Euclidean distance from ag_k to the center of mass of the region r_i. Since f_{alloc} should be minimized, the first term in f_{alloc} considers a spatial criterion and the second term balances the server workload. Every time a partition is updated, the corresponding state (center of mass and number of agents) is sent to the neighbor servers.

4. Region-based partitioning methods

4.1. R-Tree

The R-Tree is one of the most popular dynamic index structure for spatial searching [5]. We have implemented a partitioning method based on the R-Tree structure that is aimed to optimize the area of the rectangles enclosing the crowd. The most interesting feature of this approach is that it is an efficient structure for managing the partitioning problem, since it let us to handle the crowd motion as insertions and deletions in the tree, where the f_{alloc} criteria can be easily introduced.

An R-Tree is a height-balanced tree structure that splits the space with hierarchically nested, and possibly overlapping, Minimum Bounding Rectangles (MBRs). A MBR is the minimum rectangle that encloses a single or a group of agents. Each node of an R-Tree has a variable number of entries (up to some pre-defined maximum). Each entry within a non-leaf node stores two kinds of data: a way of identifying a child node, and the MBR of all entries within this child node. Each entry within a leaf node stores two kinds of information: the actual data element, and the MBR of the data element. There are two parameters that define the shape of a R-Tree: the maximum and the minimum number of entries in a node (we will denote these parameters as M and m, respectively). On other hand, choosing an adequate splitting method is important, since insertions and deletions will generate node splits to keep the tree balanced. Therefore, the splitting method used for dividing a node when it has more than M entries can determine the performance of the R-Tree method. In our implementation, we have chosen a value of 5 for the parameter M and a value of 2 for the parameter m. The reason is that the use of these values generate the least CPU utilization during R-Tree updating. Regarding the splitting method, we have used the *Quadratic split* method [5], since it shows a lower CPU utilization for tree searches than the *Linear split* method and there are no significant differences with respect to the *Linear split* method for low values of the parameter M.

In order to illustrate the implemented R-Tree algorithm, Fig. 2 shows and example of how the partitioning criterion f_{alloc} is used. A set of fifteen agents, represented as labeled circles is shown in Fig. 2. The square around each agent represents the MBR of the agent, that will be used for R-Tree insertion or deletion. The set of agents is partitioned in two regions, delimited each one by the MBR enclosing its agents. For each region, different subregions are depicted based on the agents location, and the resulting R-Tree for the two regions is shown on the right side. The value of parameter M is set to 3 in both R-Trees, and there is only one non-leaf node, the root node. Each entry in the root node contains the MBR of it child node (named each MBR with *Rx* where *x* varies from 1 to 6). These MBRs will be used to guide the search during insertions and deletions in the tree.

Fig. 2(b) shows how the R-Trees evolve when agents 8 and 12 move. When the agent 8 moves, firstly is applied the f_{alloc} criterion to determine whether it must change the region or not. In this case, $f_{\text{alloc}}(ag_8, \text{region}_1) < f_{\text{alloc}}(ag_8, \text{region}_2)$, that is, the criterion determines a region exchange. Thus, agent 8 is deleted from the R-Tree 2 and it is inserted in R-Tree 1. The movement of agent 12 does not imply a region change, but the reinsertion in the R-Tree of Region 1 implies a brach change, since it is reinserted as a child of MBR *R*5.

In order to illustrate the partitions provided by the R-Tree method, Fig. 3 shows three different instants of a crowd simulation. In this Figure, agents are represented as dots, and each MBR in the partition shows a different level of grey. It can be seen that at the beginning (Fig. 3(a)) some overlapping exists among the MBRs of the regions. As the simulation evolves (Fig. 3(b)

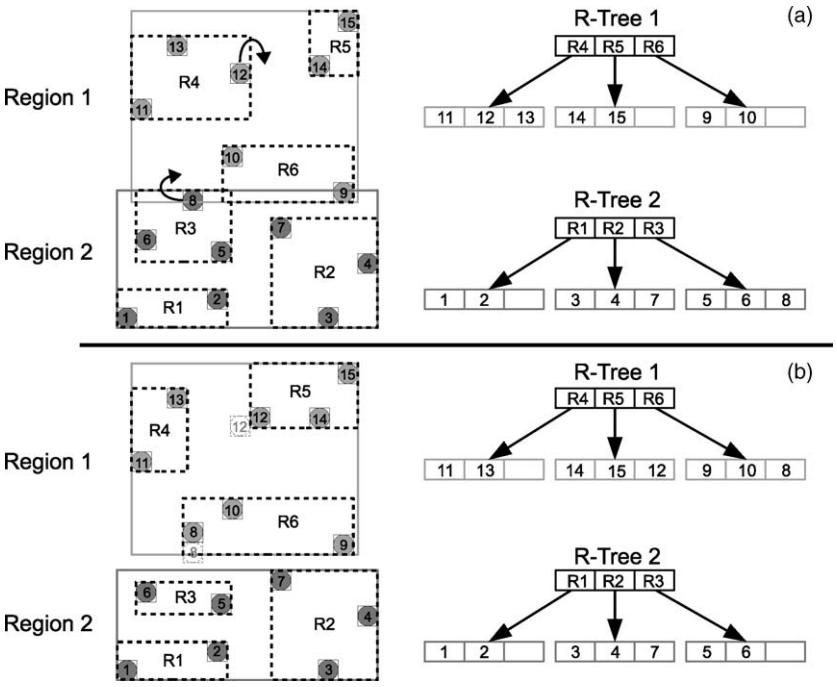

Fig. 2. Update of two R-Trees (a) initial state and (b) after agents movement.

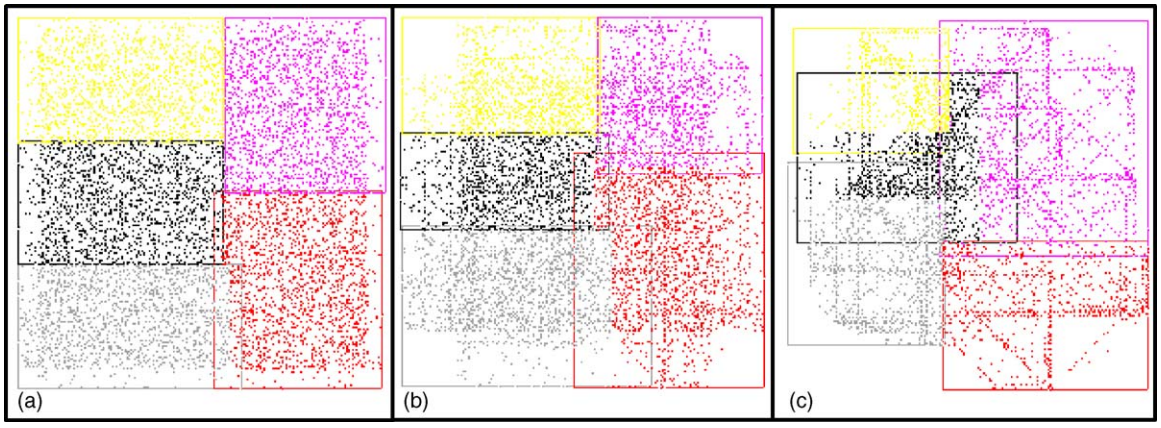

Fig. 3. Snapshots of the partitions provided by R-Tree at different simulation stages (a) beginning; (b) middle; (c) end.

and (c)), the R-Tree provides partitions with an increasing overlapping area.

4.2. A Genetic Algorithm

Genetic Algorithms (GA) consist of a search method based on the concept of evolution by natural selection [13,6]. GA start from an initial population, made of *R chromosomes* (solutions of the considered problem), that evolves following certain rules, until reaching a convergence condition that maximizes a fitness function. In this case, we have used $H(P)$ as the fitness function. Each iteration of the algorithm consists of generating a new population from the existing one. In the GA proposed for solving this problem [12], a chromosome consists of an integer array that contains *k Minimum Bound Rectangles (MBR)*. Each MBR is a quadruple $[x\_\min, x\_\max, y\_\min, y\_\max]$ that defines a rectangular region of the virtual world enclosing a subset of the crowd. Thus, a chromosome defines a partition of the crowd in k regions. As an example, Fig. 4 shows a chromosome for a given population with $k = 4$.

In order to generate a new population from the existing one, different operators can be applied to the chromosomes. The selection operator allows to select those population individuals that will be used for reproduction in each iteration of the algorithm. The purpose of the selection operator is to give more chances to the most suitable individuals (chromosomes) in the current population. The crossover operator allows the generation of an offspring from the previously selected ancestor chromosomes. The mutation operator consists of the random alteration of each of the elements (genes) in the chromosome with a mutation probability. The purpose of mutation is to produce population diversity. Finally, the replacement operator consists of replacing

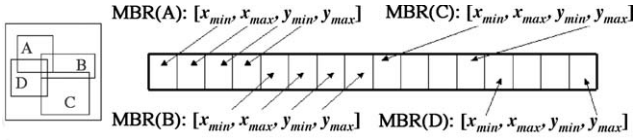

Fig. 4. Chromosome used for the Genetic Algorithm.

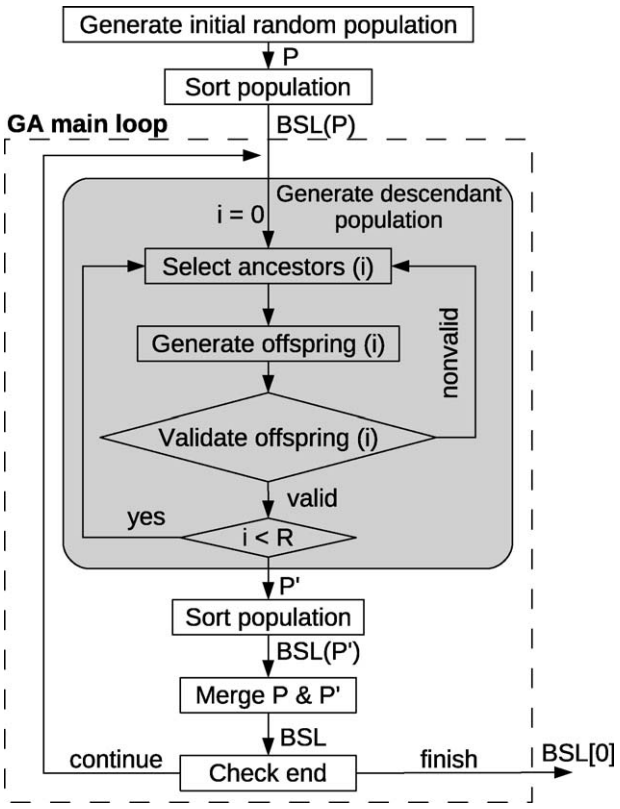

Fig. 5. The Genetic Algorithm main loop.

the current population by their offsprings or a mix of both current chromosomes and offsprings.

Fig. 5 shows the diagram representing the main loop of the implemented GA. As most of heuristic methods, it starts from an initial population of R chromosomes randomly generated. These chromosomes are sorted by the fitness function $H(P)$ associated with each chromosome in ascending order (the first chromosome is the one with the best $H(P)$ value). We have denoted this sorted list as *Best Solutions List (BSL)*. This list represents the initial population for the Genetic Algorithm, and it will contain the best R solutions found until that iteration by the GA, that is, the current population pool. The value of R is a parameter that must be tuned. We have used a value of $R = 10$, since the execution time of the partitioning method is limited by the server cycle in the crowd simulation. For greater values of R, the GA partitioning method exceeded the allowed server cycle.

Each GA iteration consists of generating a *descendant* generation of R chromosomes, starting from an *ancestor* generation. The way that the algorithm provides the next generation determines the behavior of the GA. We have chosen a sexual reproduction technique [13], in such a way that each descendant is generated starting from two ancestors. In each iteration we have used a pseudo-random selection operator. Concretely, the first ancestor for the ith chromosome of the population is the ith chromosome of the population in the previous iteration. The second ancestor is randomly selected among the 50% of the previous population with the best fitness function.

From each two ancestors, an offspring is obtained by applying a crossover operator. Concretely, we have computed a randomly skewed average of the corresponding coordinates in each of the ancestors. This skewed average is computed for all the coordinates in an MBR and for all the MBRs in a chromosome. As an example, Fig. 6 shows the MBRs corresponding to two ancestors and an example of the resulting offspring. In this figure,

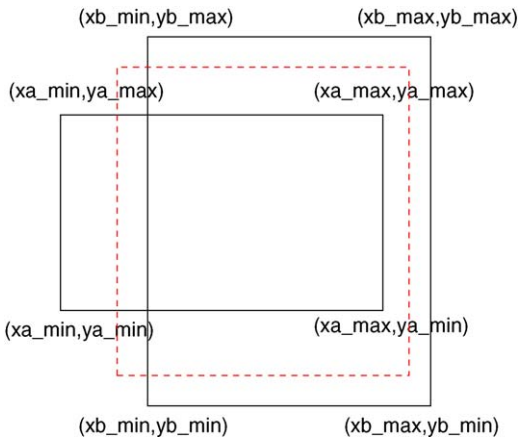

Fig. 6. Offspring generation.

we can see an ancestor MBR a whose coordinates are defined by the quadruple $[xa\_\min, xa\_\max, ya\_\min, ya\_\max]$ and a second ancestor MBR whose coordinates are defined by the quadruple $[xb\_\min, xb\_\max, yb\_\min, yb\_\max]$. From these two MBRs, the MBR with dashed lines is computed.

It must be noticed that this reproduction method can produce non-valid offsprings, because they define MBRs with different shapes. For example, the resulting MBR in Fig. 6 is narrower than ancestor a. If the rest of MBRs in the resulting chromosome do not include the area of MBR a not covered by the resulting offspring, then the agents located at that area will not be assigned to the semantic database. Moreover, the initial partition, provided by the previous execution of the GA, can be corrupted due to the movement of agents during the AS cycle. In this case, the initial population starts from a modified partition where some regions are expanded as necessary to cover all the agents at the current locations. When an invalid offspring is generated it is simply discarded, and another offspring is generated from different ancestors in the population. When all the ancestor population has been used for producing offsprings and the number of valid offsprings reaches R, then the replacement operator should be applied. Concretely, the new offsprings and the previous chromosomes in the BSL are sorted and merged to obtain the new BSL (population pool) for that iteration. The mutation operator is not used, since exchanging two or more coordinates between different MBRs could lead to invalid chromosomes.

Finally, the finishing condition should be checked in order to detect when the GA should finish. On the one hand, we have established the execution time as one of the finishing conditions of the algorithm, since one of the main constraints in crowd simulations is the execution time of the search. This time must be shorter than a fraction of the AS cycle, denoted as T, in order to provide an effective partition. Concretely, we have set T to half of the AS cycle period, that is, 125 ms. On the other hand, in order to ensure that the proposed method provides the best possible solution, we have added the decrease of $H(P)$ as a convergence condition. That is, if the $H(P)$ value of the first chromosome in the BSL is not decreased in two successive iterations, then the algorithm finishes. Therefore, the first chromosome in the BSL is chosen as the result of the search either when the convergence condition is reached or when the algorithm has been executed during T milliseconds.

Fig. 7 shows a snapshot of the different partitions provided by the GA method during a simulation. It can be seen that at the beginning (Fig. 7(a)) some overlapping exists among the regions in the partition. As the simulation evolves (Fig. 7(b) and (c)), the GA method provides partitions in which the region overlapping area is

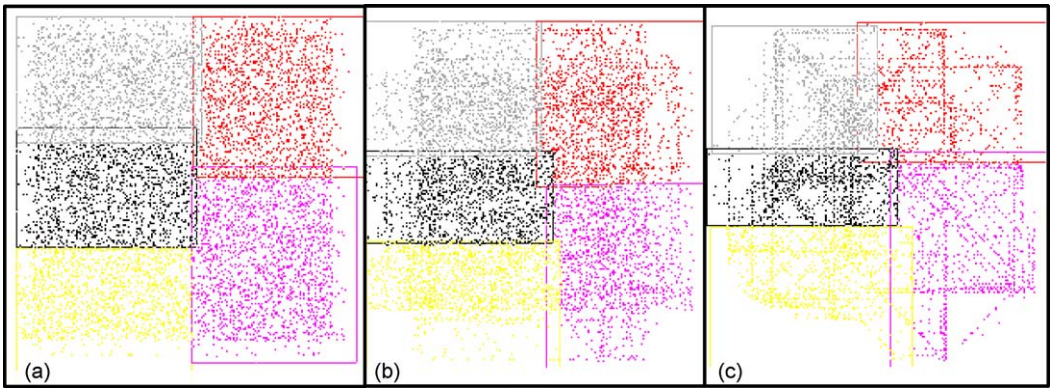

Fig. 7. Snapshots of the partitions provided by GA at different simulation stages (a) beginning; (b) middle; (c) end.

lower than in the case of the R-Tree method. The reason for this behavior is due to the heuristic search procedure carried out by this method.

4.3. Convex hull

This approach is similar to the R-Tree technique described above, since it is aimed to optimize the area of spatial structures enclosing the crowd. However, unlike the R-Tree technique, it is based on computing the convex hull of the points representing agents in a given region. The partitioning technique in distributed crowd simulations can benefit from the use of convex hulls, since these spatial structures inherently reduce the area of the regions assigned to servers when compared to the rectangles used in the R-Tree technique, and one of the purposes of the partitioning technique is to minimize the overlapping area of different regions.

Concretely, we have implemented the Quickhull algorithm (QHull) [1]. As stated above, each server periodically updates the agents assigned to its region according to the f_{alloc} function. Once the agents have been inserted, the convex hull can be recomputed, so the center of mass and the number of agents can be also updated.

The complete set of steps followed by each server to update its region is shown in Fig. 8.

Since agents can migrate among the different regions considered (and therefore they should be re-assigned to different servers, notice the *Assign* function in Fig. 8) a new convex hull should be computed each time the partitioning method is executed, and no updating of previous convex hulls are used.

The computing of the convex hull for each region (*Compute_con_hull*() function) is illustrated in Fig. 9. The steps followed by the implemented Quickhull algorithm [1] are the following ones: initially, the first hull is created with the most distant points (agents) in the vertical and horizontal directions. This process is shown in Fig. 9(a) and it requires to access to every point of the initial set ($|n|$), so it has a linear cost ($O(n)$). The initial hull may not include all the points in the region, as Fig. 9(b) shows. Then, the algorithm recursively finds and connects the most distant points in the orthogonal direction to each side of the previous hull. Fig. 9(c) represents these steps. The recursion ends when all the agents are inside of the current hull, as shown in Fig. 9(d).

Since the execution time of the partitioning method is limited by the server cycle in the crowd simulation, the partitioning method must add the lowest overhead as possible. Although the temporal cost of the QHull method greatly increases with the number of spatial dimensions, agents are represented as 2-d points, and therefore the Hull method has a cost of in $O(n \cdot log(v))$, being n the number agents and v the number of the hull vertices. Thus, the QHull method seems to be theoretically adequate for solving this problem.

In order to illustrate the partitions provided by the QHull method, Fig. 10 shows three different instants of a crowd simulation. It can be seen that at the beginning (Fig. 10(a)) there is no significant overlapping among the regions of this partition. In

```
1 {int N; /*Number of agents*/
2 int k; /*Number of regions*/
3 int i,j;
4 int Minimum = Max_value;
5 /*Quick Hull Method*/
6 for (i=1;i<=N;i++){
7       for (j=1;j<=k;j++){
8           f_alloc(ag_i, r_j) = dstMC(ag_i, r_j) + nAgs(r_j) * dstMC(ag_i, r_j)
9           if (f_alloc(ag_i, r_j)<Minimum){
10              Minimum = f_alloc(ag_i, r_j)
11              MinRegion = r_j
12          }
13      }
14      Assign(ag_i, MinRegion)
15 }
16 Compute_convex_hull();
17 }
```

Fig. 8. Algorithm for the Quick Hull method.

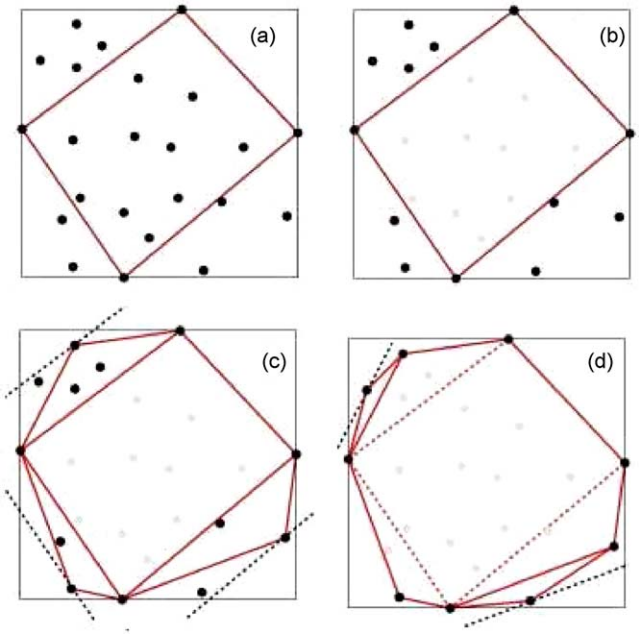

Fig. 9. Steps followed by the Quickhull algorithm.

addition, the QHull method is able to keep a very low overlapping during all the simulation (Fig. 10(b) and (c)).

5. Performance evaluation

In this section, we compare the performance obtained by the three methods described in the previous section: R-Tree, GA and QHull. We propose the evaluation of the partitioning methods by simulation. That is, we have extracted the motion patterns of each agent off-line, and we have used this information as an input for the considered methods. Concretely, we have evaluated each method in a crowd simulation composed by 8000 autonomous agents to be distributed in five regions. The simulations have been performed on a sequential system consisting of an Intel Core Duo processor running at 1600 MHz and 2 GBytes of RAM.

We have evaluated two different crowd scenarios: an evacuation and an urban environment. The evacuation scenario consists of a structured 2D world where there are several emergency exits. The autonomous agents must try to escape from the world as soon as possible. For this scenario we have used the same well-known movement patterns considered in our previous work [12]. In order to achieve these movement patterns, we have considered the following 2D world configurations: *full*, where there are a lot of emergency exits uniformly distributed within the 2D world (CCP pattern); *perimeter*, where all the emergency exits are uniformly distributed along the four borders of the virtual world (HP-Near); *up*, where there are only a few exits and they are located at the top border of the world (HP-All); and *down*, where there is only one exit located at the bottom border of the world (HP-All with a single hot-point). In order to illustrate these configurations, Fig. 11 shows four snapshots of the virtual world with these configurations at half of the simulation time. In this figure, the 2D world is viewed from above, and agents are represented as grey dots.

The second scenario considers an urban environment where the population size remains constant during the whole simulation. In this way, the complexity of the partitioning problem does not decrease with the simulation time. This scenario contains twenty target locations randomly distributed within the virtual world. Each agent randomly selects one of these targets and approaches it. Once the target has been reached, then the agent randomly selects

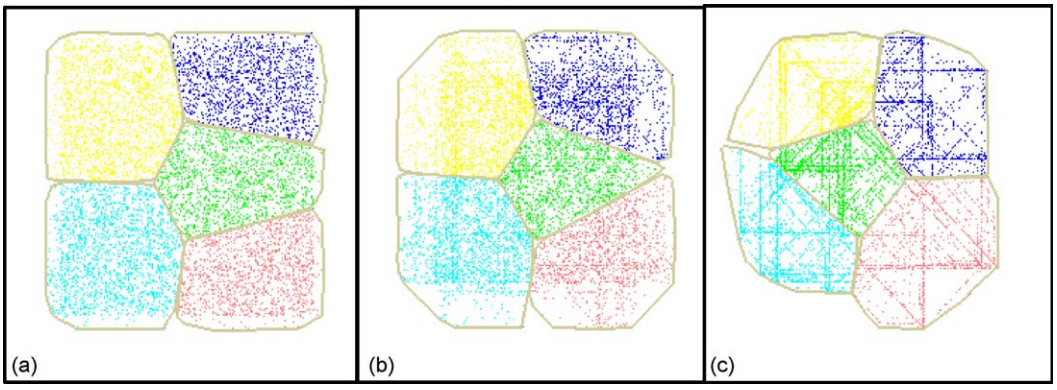

Fig. 10. Snapshots of the partitions provided by QHull at different simulation stages (a) beginning; (b) middle; (c) end.

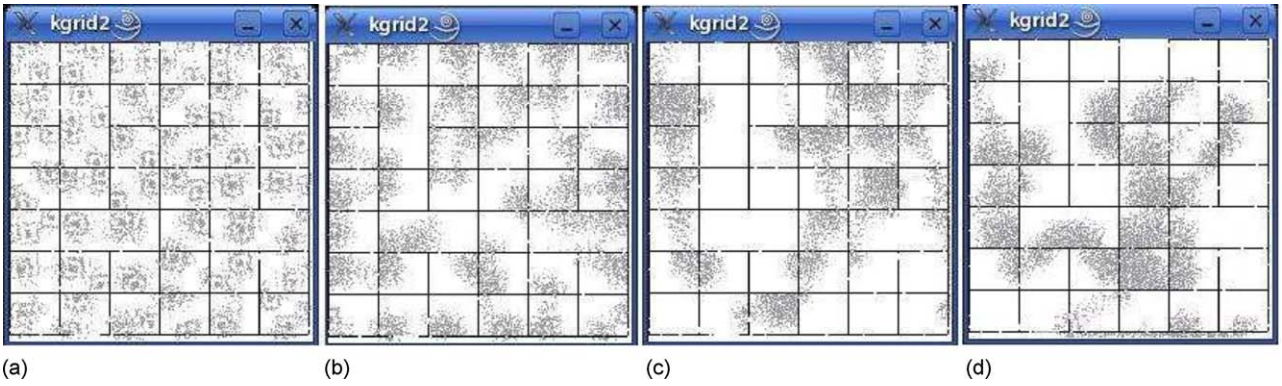

Fig. 11. Movement patterns: (a) full; (b) perimeter; (c) up; (d) down.

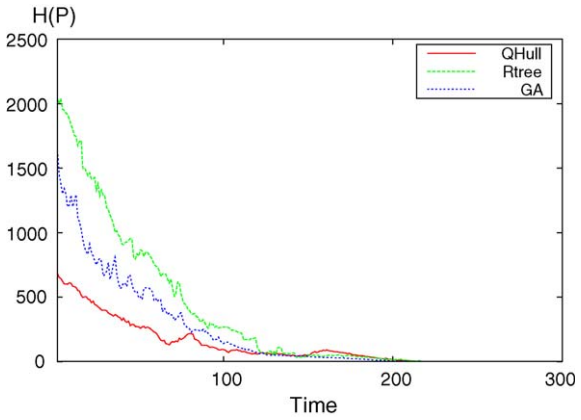

Fig. 12. Fitness function values provided by the partitioning methods for the full simulation.

the next target and repeats the process, until all the targets have been reached. We have denoted this configuration as *urban*.

In order to measure the actual improvement that the different methods can provide to real systems, we have simulated the five configurations described above. For each simulation we have calculated both the execution time and the fitness function. By execution time we refer to the amount of time required by each method for computing the provided partition in each cycle of the simulation.

Firstly, we analyze the results obtained for the evacuation scenario. Fig. 12 shows the fitness function values provided by each partitioning method for the *full* configuration. In this figure, the *X*-axis represents the simulation time (measured in simulation cycles) while the *Y*-axis represents the fitness function values. As can be seen, the values provided by the QHull method are around one half of the values provided by the GA method and around one third of the ones obtained by the R-Tree method. These differences progressively decrease towards the end of the simulation. The reason is that the population size decreases while the agents exit the virtual world, and so does the complexity of the partitioning problem.

On the other hand, Fig. 13 shows the execution time values. In this figure, the *X*-axis refers to the simulation time (measured in simulation cycles), and the *Y*-axis refers to the execution times needed to compute the partition (in milliseconds). Similarly to the fitness values, the QHull method requires the shortest execution times. Specifically, this method requires one third of the execution time required by the GA method and one fourth of the time

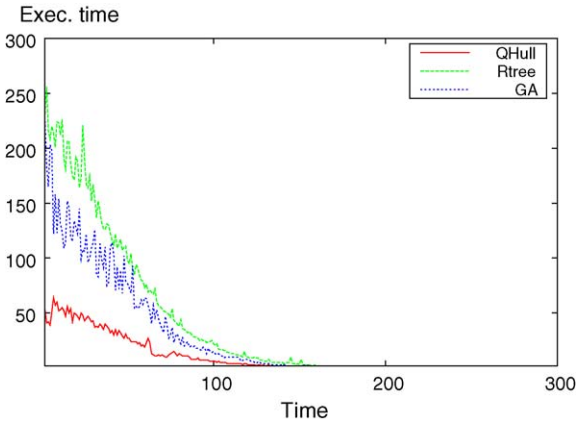

Fig. 13. Execution times required by the partitioning methods for the full simulation.

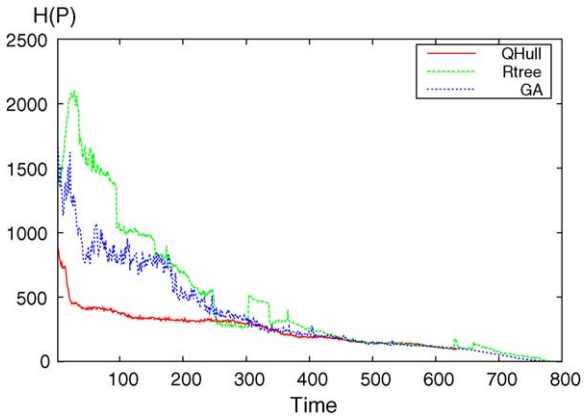

Fig. 14. Fitness function values provided by the partitioning methods for the perimeter simulation.

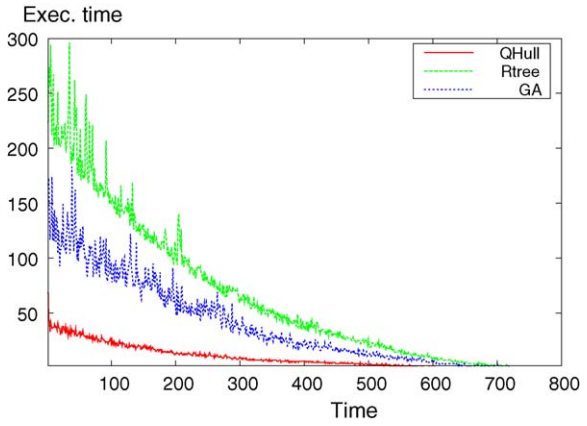

Fig. 15. Execution times required by the partitioning methods for the perimeter simulation.

required by the R-Tree method. The ratio between the different execution times remains constant during the whole simulation. Once more, the execution times required for all methods decrease as the simulation proceeds, since the agents exit the virtual world.

Figs. 14 and 15 compare the three partition methods for the *perimeter* configuration. In this type of evacuation pattern, the QHull method better fits the partition during the first half of the simulation (see Fig. 14). Afterwards, the fitness values are equal for the three partition methods. However, as Fig. 15 shows, the QHull method outperforms its competitors with respect to the execution time. In this case, the QHull method is around three and a half times faster than the GA method and around six times faster than the R-Tree method. Again, even though this ratio do not vary, the execution times decrease as the agents evacuate the scenario.

The comparison of the three partitioning methods for the *up* configuration can be seen in Figs. 16 and 17. The fitness function values in Fig. 16 show how QHull's partitions are better than the ones provided by the GA and the R-Tree methods, regardless of concrete deviations due to this particular movement pattern. Although the QHull plot reaches the values shown by the R-Tree plot at some points, this is a eventual behavior. Furthermore, the execution time values of Fig. 17 reflect that the QHull is once more the fastest one of the three methods being considered. Concretely, the QHull's execution time is one third of the GA's execution time and one sixth of the R-Tree's execution time.

The results for the *down* configuration are analyzed in Figs. 18 and 19. Fig. 18 shows the fitness function values provided by each partitioning method for the *down* simulations. This figure shows

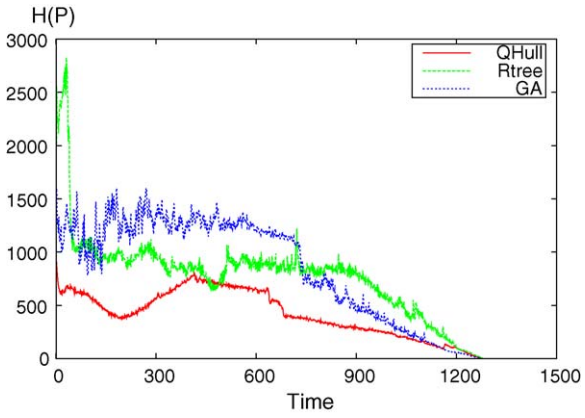

Fig. 16. Fitness function values provided by the partitioning methods for the up simulation.

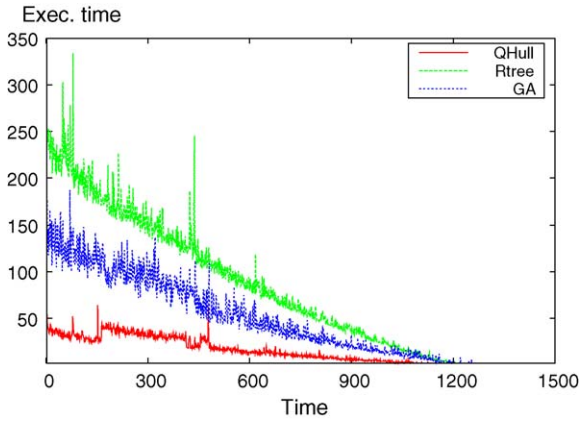

Fig. 17. Execution times required by the partitioning methods for the up simulation.

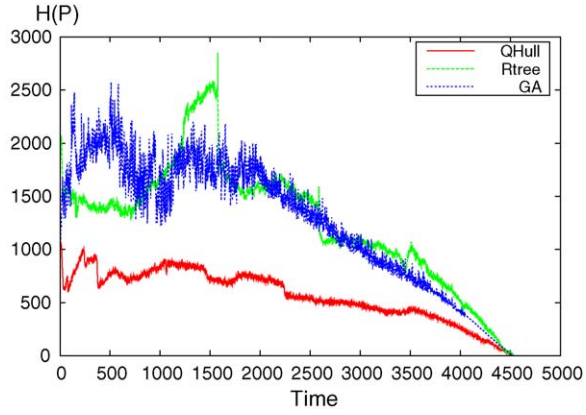

Fig. 18. Fitness function values provided by the partitioning methods for the down simulation.

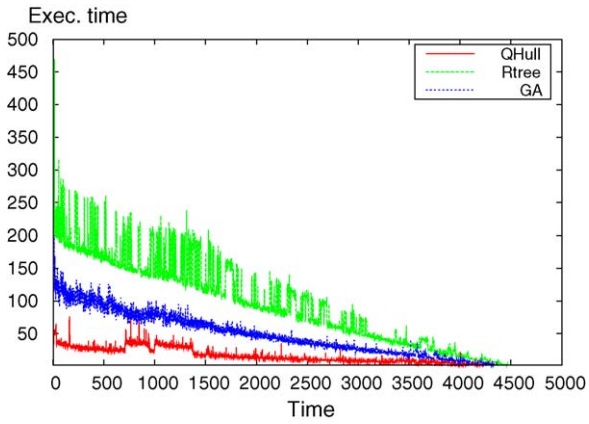

Fig. 19. Execution times required by the partitioning methods for the down simulation.

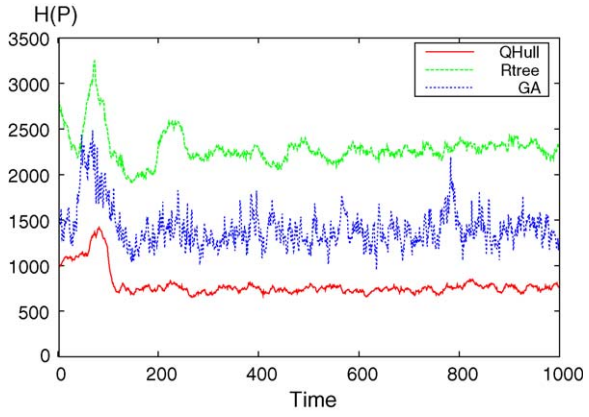

Fig. 20. Fitness function values provided by the partitioning methods for the urban simulation.

that the QHull method provides the best fitness function values during the whole simulation. Concretely, the values provided by this method are about 50% (or less) of the values provided by the other two methods in the first two thirds of the simulation. These differences decrease towards the end of the simulation due to the emptying of the scenario, as stated above.

In turn, Fig. 19 shows once again that the QHull method requires the shortest execution times. Applied to this pattern, the QHull method is around three times faster than the GA method and around five times faster than the R-Tree method. Fig. 19 again shows how the execution times required for all the methods decrease as the simulation proceeds, since the agents exit the virtual world.

In order to show the performance of the partitioning methods when the population size remains constant during the whole simulation we use the *urban* scenario. On the one hand, Fig. 20 shows the fitness function values provided by each partitioning method for the *urban* simulation. In this case all the plots have similar shapes, and after a stabilizing period (about 200 ms) all of them show a flat slope. Fig. 20 clearly shows that again the QHull method provides the best fitness function values, being around 50% lower (better) than the values provided by the GA method and around one third of the values provided by the R-Tree method.

On the other hand, Fig. 21 shows the execution times required by each partitioning method for the *urban* simulation. This figure also shows great differences in the execution times required by each method. Again, the QHull method requires execution times that are around one third of the times required by the GA method and around one fifth of the R-Tree method.

Summing up, these results show that the QHull method provides the best fitness function values while requiring the shortest execution times. However, the actual benefits that each method provides to the crowd simulation should be measured. In accordance with this, Table 1 shows the performance of the partitioning methods in terms of the average number of locking requests produced in each AS cycle among the computers hosting the database. Each value shown in this table is the average value for all the AS cycles of the

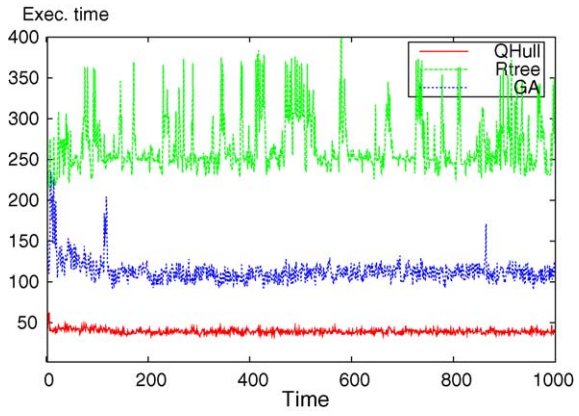

Fig. 21. Execution times required by the partitioning methods for the urban simulation.

Table 1
Actual performance provided by the different methods.

Method	Locks	Std. deviation
Full		
GA	357.98	210.12
RTree	691.57	127.06
QHull	180.05	157.64
Perimeter		
GA	345.24	391.52
RTree	610.29	277.97
QHull	136.03	469.21
Up		
GA	580.61	1277.7
RTree	1163.84	227.2
QHull	337.89	563.04
Down		
GA	1133.76	1471.68
RTree	1903.52	327.39
QHull	723.19	351.49
Urban		
GA	1895.97	694.31
RTree	3156.02	989.74
QHull	1022.98	439.6

simulation time. Moreover, this table shows the average standard deviation for the average number of agents assigned to each server. That is, how balanced the generated partitions are.

Essentially, Table 1 shows that the behavior of each method does not depend on the movement pattern of the agents, since similar results are obtained for the four patterns considered in the evacuation scenario as well as for the urban scenario. As Table 1 shows, the GA method provided an intermediate number of locking requests and the highest standard deviations (i.e. the worst balanced partitions) for the five movement patterns. On the contrary, the R-Tree method provided the highest number of locking requests and the lowest standard deviations (i.e. the best balanced partitions). Finally, the QHull method provided the lowest number of locking requests and also the lowest standard deviation. Thus, we can conclude that this method is the most appropriate one for solving the partitioning problem in distributed crowd simulations.

6. Conclusions and future work

In this paper, we have analyzed the use of irregular shape regions (convex hulls) for solving the partitioning problem. Concretely, we have compared partitioning methods (with both a heuristic method and an algorithmic method) that use rectangular regions with a partitioning method based on convex hulls.

The performance evaluation results show that the method based on convex hulls outperforms the rest of the considered methods in terms of both fitness function values and execution times, regardless of the movement pattern followed by the agents. As a result, this method provides partitions containing regions with lower overlapping area, thus reducing the computational cost required for managing the crowd. These results indicate that the shape of the regions in the partition has a major influence on the performance of the partitioning method, rather than the search method used.

As a future work to be done, we plan to apply optimization techniques to the QHull method in order to increase the quality of the partitions provided by this method.

Acknowledgments

This work has been jointly supported by the Spanish MEC, the European Commission FEDER funds, and the University of Valencia under grants Consolider-Ingenio 2010 CSD2006-00046, TIN2009-14475-C04-04, and UV-BVSPIE-07-1788.

References

[1] C.B. Barber, D.P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, ACM Trans. Math. Softw. 22 (4) (1996) 469–483.
[2] D.E. Diller, W. Ferguson, A.M. Leung, B. Benyo, D. Foley, Behavior modeling in commercial games, in: Behavior Representation in Modeling and Simulation (BRIMS), 2004.
[3] S. Dobbyn, J. Hamill, K. O'Conor, C. O'Sullivan, Geopostors: a real-time geometry/impostor crowd rendering system, ACM Trans. Graph. 24 (3) (2005) 933.
[4] T. Frank, K. Bernert, K. Pachler, Dynamic load balancing for lagrangian particle tracking algorithms on mimd cluster computers, in: PARCO'2001—International Conference on Parallel Computing 2001, 2001.
[5] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: SIGMOD'84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 1984.
[6] R.L. Haupt, S.E. Haupt, Practical Genetic Algorithms, Wiley, 1997.
[7] A. Iglesias, F. Luengo, New goal selection scheme for behavioral animation of intelligent virtual agents, IEICE Trans. Information Syst., Special Issue on 'Cyber-Worlds' E88-D (5) (2005) 865–871.
[8] T. C. International Business Machines Corporation, Sony Computer Entertainment Incorporated, Cell Broadband Engine Programming Handbook, April 2007.
[9] P.A. Kruszewski, A game-based cots system for simulating intelligent 3d agents, in: BRIMS'05: Proceedings of the 2005 Behavior Representation in Modelling and Simulation Conference, 2005.
[10] M. Lozano, P. Morillo, D. Lewis, D. Reiners, C. Cruz-Neira, A distributed framework for scalable large-scale crowd simulation, in: Virtual Reality, Second International Conference, ICVR 2007, Held as part of HCI International 2007, Beijing, China, July 22–27, vol. 4563 of Lecture Notes in Computer Science, Springer, 2007.
[11] M. Lozano, P. Morillo, J.M. Orduña, V. Cavero, On the design of an efficient architecture for supporting large crowds of autonomous agents, in: Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA'07), 2007.
[12] M. Lozano, J.M. Orduña, V. Cavero, A genetic approach for distributing semantic databases of crowd simulations, in: Proceedings of the 21st International Parallel and Distributed Symposium, IEEE Computer Society Press, 2007.
[13] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer, 1994.
[14] P. Morillo, J.M. Orduña, M. Fernández, J. Duato, Improving the performance of distributed virtual environment systems, IEEE Trans. Parallel Distributed Syst. 16 (7) (2005) 637–649.
[15] H. Nakanishi, T. Ishida, Freewalk/q: social interaction platform in virtual space, in: VRST'04: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, ACM Press, New York, NY, USA, 2004.
[16] N. Pelechano, J.M. Allbeck, N.I. Badler, Controlling individual agents in high-density crowd simulation, in: SCA'07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, Aire-la-Ville, Switzerland, 2007.
[17] M.J. Quinn, R.A. Metoyer, K. Hunter-Zaworski, Parallel implementation of the social forces model, in: Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, 2003.
[18] C. Reynolds, Big fast crowds on ps3, in: Sandbox'06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames, ACM, New York, NY, USA, 2006.
[19] C.W. Reynolds, Flocks, herds and schools: a distributed behavioral model, in: SIGGRAPH'87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, 1987.

[20] K. Sims, Particle animation and rendering using data parallel computation, in: SIGGRAPH'90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, 1990.
[21] S. Singhal, M. Zyda, Networked Virtual Environments, ACM Press, 1999.
[22] A. Steed, R. Abou-Haidar, Partitioning crowded virtual environments, in: VRST'03: Proceedings of the ACM Symposium on Virtual Reality Software and Technology, ACM, New York, NY, USA, 2003.
[23] M. Sung, M. Gleicher, S. Chenney, Scalable behaviors for crowd simulations, in: Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, ACM Press, 2004.
[24] F. Tecchia, C. Loscos, Y. Chrysathou, Visualizing crowds in real time, Computer Graphics Forum 21.
[25] A. Treuille, S. Cooper, Z. Popovic, Continuum crowds, in: SIGGRAPH'06: ACM SIGGRAPH 2006 Papers, ACM, 2006.
[26] G. Vigueras, M. Lozano, C. Perez, J. Orduña, A scalable architecture for crowd simulation: implementing a parallel action server, in: ICPP'08. 37th International Conference on Parallel Processing, 2008.
[27] B. Zhou, S. Zhou, Parallel simulation of group behaviors, in: WSC'04: Proceedings of the 36th Conference on Winter Simulation, Winter Simulation Conference, 2004.