Adaptive Load-Balancing for MMOG Servers Using KD-trees

CARLOS EDUARDO B. BEZERRA, JOÃO L. D. COMBA, and CLÁUDIO F. R. GEYER,
Universidade Federal do Rio Grande do Sul

In massively multiplayer online games (MMOGs) there is a great demand for high bandwidth connections with irregular access patterns. Such irregular demand is because players, who can vary from a few hundred to several tens of thousands, often occupy the virtual environment of the game in different ways with varying densities. Hence there is a great need for decentralized architectures with multiple servers that employ load-balancing algorithms to manage regions of the virtual environment. In such systems, each player only connects to the server that manages the region where the player's avatar is located, whereas each server is responsible for mediating the interaction between all pairs of players connected to it. Devising the proper load-balancing algorithm so that it takes spatial and variable occupations into account is a challenging problem which requires adaptive (and possibly dynamic) partitioning of the virtual environment. In this work, we propose the use of a kd-tree for partitioning the game environment into regions, and dynamically adjust the resulting subdivisions based on the distribution of avatars in the virtual environment. We compared our algorithm to competing approaches found in the literature and demonstrated that our algorithm performed better in most aspects we analyzed.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling

General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: Distributed server, kd-trees, load-balancing, MMOGs

ACM Reference Format:
Bezerra, C. E. B., Comba, J. L. D., and Geyer, C. F. R. 2012. Adaptive load-blancing for MMOG servers using KD-trees ACM Comput. Entertain. 10, 3, Article 5 (November 2012), 16 pages.
DOI = 10.1145/2381876.2381881 http://doi.acm.org/10.1145/2381876.2381881

1. INTRODUCTION

Massively multiplayer online games (MMOGs) is a genre of computer games that demands high network bandwidth to allow interaction among players. The main characteristic of MMOGs is the large number of players interacting simultaneously, which currently can reach up to tens of thousands [Schiele et al. 2007]. Client-server architectures are often configured to allow communication among players, with the server mediating the communication between each pair of players. To interact, each player sends commands to the server, which calculates the new game state and propagates it back to all players affected by the state change. This mechanism can lead to many state update messages sent by the server, which may be quadratic on the number of players in the worst case, that is, when all players interact with one another. Therefore, the

Invited extended version of the 2009 SBGames paper.
The development of this work was supported by the National Research Council (CNPq) and by the Coordination for Improvement of the Higher Education Personnel (CAPES).
Authors' address: C. E. B. Bezerra, J. L. D. Comba, and C. F. R. Geyer, Programa de pós-Graduação em Computação, Instituto de Informática, UFRGS Caixa Postal15064, 91501-970, Porto Alegre - RS - Brasil.

DOI 10.1145/2381876.2381881 http://doi.acm.org/10.1145/2381876.2381881

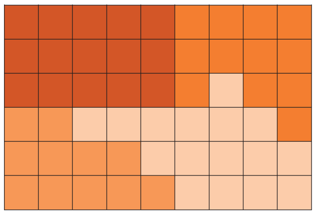


Fig. 1. Division into cells and grouping into regions.

cost of maintaining a centralized infrastructure like this is high when lots of players are connected, thus restricting the MMOG market to large companies with enough resources to pay the upkeep of the server.

To reduce this cost, several decentralized solutions have been proposed. Examples include peer-to-peer networks [Schiele et al. 2007; Rieche et al. 2007; Hampel et al. 2006; El Rhalibi and Merabti 2005; Iimura et al. 2004; Knutsson et al. 2004] or distributed servers [Ng et al. 2002; Chertov and Fahmy 2006; Lee and Lee 2003; Assiotis and Tzanov 2006], which are composed of low-cost nodes connected through the Internet. Common to both approaches is the fact that the "world", or the virtual environment of the game is divided into regions which are managed by a server (or a group of peers). Each region must not violate the network capacity of its respective server, and thus must not have more content than the load imposed on the corresponding server.

Players are assigned to servers based on their location. When an *avatar* (representation of the player in the virtual environment) is located in a region, the player controlling that avatar connects to the server associated to that region. This server becomes responsible for processing the input from that player and for sending update messages in response. If a server becomes overloaded due to an excessive number of avatars in its region, one way to reduce its load is by repartitioning the virtual environment.

Virtual environments are often divided into smaller cells, which are grouped into regions and distributed among servers. This approach has several limitations in its granularity due to cells of fixed size and position. More elaborate partitioning algorithms based in spatial data structures [Samet 2005] allow better player distribution among different servers. In this work, we use a kd-tree to dynamically partition the virtual environment. When a server is overloaded, it triggers a load balancing algorithm, which changes the limits of each region by changing the split coordinates stored in the kd-tree. We validate our proposal with several simulations and compare the results to previous work that usesthe cell division technique.

2. RELATED WORK

Different authors address the spatial partitioning of virtual environments in MMOGs for better distribution among multiple servers [Ahmed and Shirmohammadi 2008; Bezerra and Geyer 2009]. A simple approach is to partition the space into a static set of cells of fixed size and position, which are then grouped into regions and each region is then assigned to one of the servers (Figure 1). When a server becomes overloaded, part of the load is transferred to another server.

In the work of Ahmed and Shirmohammadi [2008], a cell-oriented load-balancing model is proposed. Their algorithm first enumerates all clusters of cells that are managed by the overloaded server. The smallest cluster is found, and from this cluster the cell with the least interaction with other cells on the same server is selected—the interaction between two cells A and B is defined by the authors as the number of pairs of avatars interacting with each other, one from A and the other from B. The selected cell

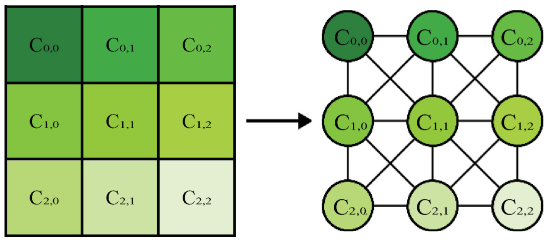


Fig. 2. Graph representation of the virtual environment.

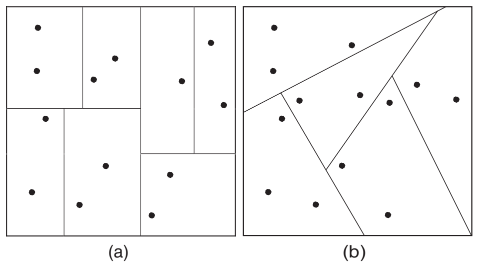


Fig. 3. Space partitioning using a kd-tree (a), and a BSP tree (b).

is then transferred to the server with the smallest load.[1] This process is repeated until the server is no longer overloaded or there is no more servers capable of absorbing more load—in this case, one option could be to reduce the frequency at which state update messages are sent to the players, as suggested by Bezerra et al. [2008].

Bezerra and Geyer [2009 also propose division into cells. To perform the division, the environment is represented by a graph, where each vertex represents a cell (Figure 2). Every edge in the graph connects two vertices representing neighboring cells. The weight of a vertex is the server's bandwidth used to send state updates to the players whose avatars are in the cell represented by that vertex. The interaction between any two cells defines the weight of the edge connecting the corresponding vertices. To form the regions, the graph is partitioned using a greedy algorithm: starting from the heaviest vertex, the vertex connected by the heaviest edge to any of the vertices already selected is added at each step until the total weight of the partition of the graph—defined as the sum of the vertices' weights—reaches a certain threshold related to the total capacity of the server that will receive the region represented by that partition of the graph.

This approach has a limited granularity distribution. If a finer granularity is desired, it is necessary to use smaller cells. This increases the number of vertices in the graph that represents the virtual environment and, consequently, the time required to perform load-balancing. Also, the control message with the list of cells designated to each server becomes longer. A possibility for circumventing this problem relies on partitioning the virtual environment using spatial data structures [Bentley 1975], for example kd-trees or binary space partitioning (BSP)-trees (Figure 3). Spatial data structures are widely used in computer graphics, and can be used to partition the virtual environments of MMOGs. There is a vast literature on techniques that aim to create spatial partitions with balanced "loads". In Luque et al. [2005], the goal is to reduce the time needed to calculate collision detection between pairs of objects in a dynamic

[1]Considering "load" as the bandwidth used to send state updates to the players whose avatars are positioned in the cells managed by that server.

environment. A BSP-tree is used to distribute the objects in the scene. Objects that are in different partitions do not collide and there is no need to perform a complex collision test. A dynamic readjustment of the tree is proposed, where, as objects move, their distribution on the leaf-nodes of the tree is balanced and, therefore, the time required to perform collision detection is minimized. Some of these ideas may be used for loadbalancing among servers in MMOGs.

3. PROPOSED APPROACH

In this section we describe our load-balancing algorithm. The proposal is based on two criteria: first, we define the *load* of a server as the bandwidth to send state updates to clients; and second, the system should be considered heterogeneous (i.e., every server may have a different amount of available bandwidth). Since every player may interact with many others, the load on each server should be proportional to the bandwidth required to send state update messages to the players connected to it. This allows each player to receive state updates from many other players, leading to a quadratic total number of state update messages to be sent by the servers. Also, upload rates are usually lower than download rates. Since the number of commands received by a server grows linearly with the number of players, we consider only the upload bandwidth to send state updates, ignoring the download rate used by the servers to receive commands from clients.

As mentioned before, we propose the utilization of a data structure known as the kd-tree to divide the environment of the game into regions. Despite having three-dimensional graphics, the vast majority of MMOGs (e.g., World of Warcraft [Blizzard 2004], Ragnarok [Gravity 2001], and Lineage II [NCsoft 2003] have a 2D simulated world—cities, forests, swamps, and points of interest in general. Hence we propose to use a 2D kd-tree.

Each node of the tree represents a region of the space defined by a coordinate used to split the space. Each of the two children of that node represents the two half-spaces obtained in the partitioning process. We alternate the splitting axis (in the case of two dimensions, the axes x and y at each level of the tree—if the first-level nodes store x-coordinates, the second-level nodes store y-coordinates, and so on. Every leaf node also represents a region of the space, but it does not store any split coordinates. Instead, it stores a list of the avatars present in that region. Finally, each leaf node is associated to a server of the game. When a server is overloaded, it triggers the load-balancing, which uses the kd-tree to readjust the split coordinates that define its region, thus reducing the amount of content.

The nodes of the kd-tree also store two other values: capacity and the load of the subtree. The load of a leaf node is equal to the load of the associated region. Similarly, the capacity of a leaf node corresponds to the network capacity of its associated server. For each nonleaf node, the load and capacity values correspond to the sum of those stored in its child nodes. Hence the tree root stores the total weight of the game and the total capacity—upload bandwidth—of the server system.

In the following sections, we will describe the construction of the tree, the calculation of the load associated with each server, and the proposed balancing algorithm.

3.1. Building the kd Tree

The initial kd-tree is constructed in such way that it represents a balanced tree. Algorithm 1 is used to create the tree, starting from instantiating a single root node which calls *node.build tree*(0*,* 0*, n*), where *n* is the number of leaf nodes (corresponding to the number of servers, in our case).

In Algorithm 1, *id* is used to calculate whether each node has children or not. This allows us to create a balanced tree where the difference between the number of leaf

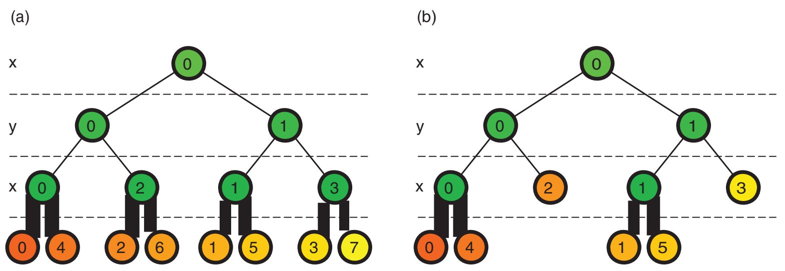


Fig. 4. Balanced kd-trees built with the algorithm described: (a) full, (b) incomplete.

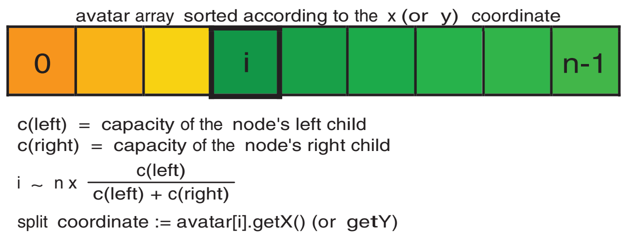

avatar array sorted according to the x (or y) coordinate
0
i
n-1
c(left) = capacity of the node's left child
c(right) = capacity of the node's right child
i ~ n x c(left) / c(left) + c(right)
split coordinate := avatar[i].getX() (or getY)

Fig. 5. Load-splitting, considering only the number of avatars.

ALGORITHM 1: node::build tree(id, level, n)

```
$this.node\_id \leftarrow id$;
**if** id $+ 2^{level} \geq n$ **then**
    $left\_child \leftarrow NIL$;
    $right\_child \leftarrow NIL$;
**else**
    $left\_child \leftarrow new\_node()$;
    $left\_child.parent \leftarrow$ **this**;
    $right\_child \leftarrow$ new_node();
    $right\_child.parent \leftarrow$ **this**;
    $left\_child.build\_tree(id, level + 1, n)$;
    $right\ child.build\ tree(id + 2^{level}, level + 1, n)$;
**end if**
```

nodes in the two subtrees of any given node is at most one. In Figure 4(a), we have a full kd-tree formed with this simple algorithm. and in Figure 4(b), an incomplete kd-tree with six-leaf nodes. As we can see, every node of the tree in (b) has two subtrees whose number of leaf nodes differs by one in the worst case.

3.2. Calculating the Load of Avatars and Tree Nodes

The definition of the split coordinate for every nonleaf node of the tree depends on how the avatars will be distributed among regions. An initial idea might be to distribute players in a way such that the number of players on each server is proportional to its upload bandwidth. To calculate the split coordinate it is enough to sort the n avatars based on their coordinates along the split axis (x or y) and assign the first n^t avatars to the left child and the last $n - n^t$ ones to the right child, where n^t and $n - n^t$ are proportional to the capacity of the left and right child, respectively (Figure 5). At each node, this operation costs $O(nlog\ n)$ due to sorting. Assigning all avatars to the root

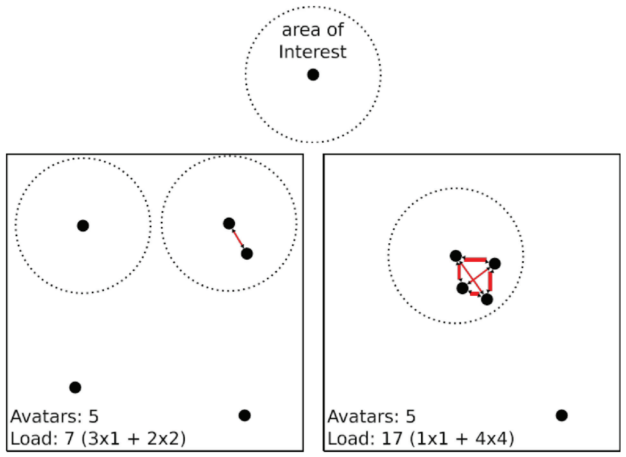


Fig. 6. Relation between avatars and load.

node and recursively executing this operation would have a complexity of $O(nlog^2n)$, as it must be repeated for each of the $O(logn)$ levels[2] of the kd-tree.

However, this distribution is not optimal, since the load imposed by players depends on how they are interacting with one another. For example, if the avatars of two players are distant from each other, there will probably be no interaction between them, and so the server will update each of these players only about its own avatar's state. In this case, the growth in the number of messages is linear with the number of players. On the other hand, if the avatars are close to one another, each player should be updated not only about the outcome of his own actions but also about the actions of every other player—in this case the number of messages may grow quadratically with the number of players (Figure 6). For this reason it is not enough to consider the number of players only when dividing them among servers.

A more appropriate way to divide the avatars is by considering the load imposed by each one on the server. A brute-force method for calculating the load is to get the distance separating each pair of avatars and, based on their interaction, calculate the number of messages that each player should receive by unit of time. This approach has complexity $O(n^2)$. One way to speed-up this calculation is to sort avatars according to their coordinates on the axis used to divide the space in the kd-tree.

For this, two nested loops are used to sweep the array of avatars, where each of the avatars contains a *load* variable initialized with zero. As the vector is sorted, the inner loop starts at the smallest index that indicates that no avatar a_j has relevance to the one referenced in the outer loop, a_i. It uses a variable *begin*, with the initial value of zero: if the coordinate of a_j is smaller than a_i, with a difference greater than the maximum view range of the avatars, the variable *begin* is incremented. For every a_j that is at a distance smaller than the maximum view range, the *load* of a_i is increased according to the relevance[3] of a_j to a_i. When the inner loop reaches an avatar a_j, such that its coordinate is greater than a_i, with a difference greater than the view range, the

[2]Although the height of the tree is $O(logS)$, where S is the number of servers, we can represent it as $O(logn)$, since $n > S$, i.e., the number of avatars is greater than the number of servers.

[3]The relevance of an object to an avatar may be used to define how many state update messages its player must receive about that object per time unit [Bezerra et al. 2008]—for that matter, avatars are also considered objects. One possible approach is to define the relevance of an object to an avatar as their proximity in the virtual environment.

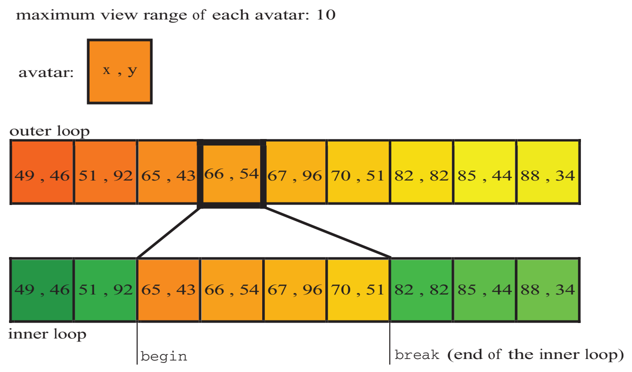


Fig. 7. Sweep of the sorted array of avatars.

outer loop moves immediately to the next step, incrementing a_i and setting the value of a_j to that stored in *begin* (Figure 7).

To evaluate the complexity of the algorithm, we refer to *width* as the length of the virtual environment along the axis used for splitting; *radius* as the maximum view range of the avatars, and n, the number of avatars. The number of relevance calculations, assuming that avatars are uniformly distributed in the virtual environment is $O(m \times n)$, where m is the number of avatars compared in the internal loop, that is, $m = \frac{2 \times radius \times n}{width}$. Sorting avatars along one of the axes takes $O(nlogn)$. Although it is still quadratic, the execution time is reduced significantly, depending on the size of the virtual environment and on the view range of the avatars. The algorithm could also sort each set of avatars a_j which are closer (in one of the axes) to a_i according to the other axis and, again, perform a sweep eliminating those that are too far away, in both dimensions. The number of relevance calculations would be $O(p \times n)$, where p is the number of avatars closer to a_i, considering the two axes of coordinates,

$$p = \frac{(2 \times radius)^2 \times n}{width \times height}.$$

In this case *height* is the extension of the environment in the second axis taken as reference. Although there is a considerable reduction in the number of relevance calculations, it does not compensate the time spent in sorting the subarray of avatars selected for each a_i. Adding up all the time spent on sorting operations, we obtain a complexity of $O(nlogn + n \times mlogm)$.

After calculating the load generated by each avatar, we use it to recursively set the split coordinate of the tree nodes so that each child node has a fair portion of its parent's load—if a node p has c and c^t as child nodes, and the capacity of c is the same of c^t, then each one of them gets half the load of p, which may correspond or not to half the avatars, since the load of these avatars may vary. As mentioned before, the capacity value stored on the leaf nodes corresponds to the upload bandwidth of the servers they represent, and the rest of the nodes have their capacity value recursively calculated.

3.3. Dynamic Load Balancing

The overloading of a server indicates that part of its load must be assigned to a different server. To accomplish this, the overloaded server collects data—the list of avatars, their positions and loads—from other servers and, using the kd-tree, adjusts the split coordinates of the regions.

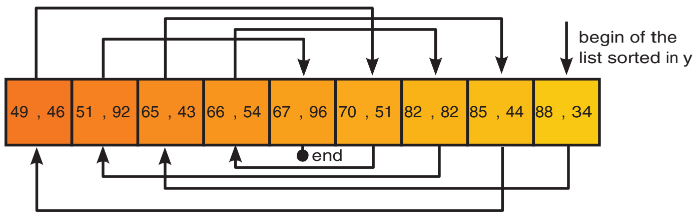


Fig. 8. Avatar array sorted by x, containing a list sort by y.

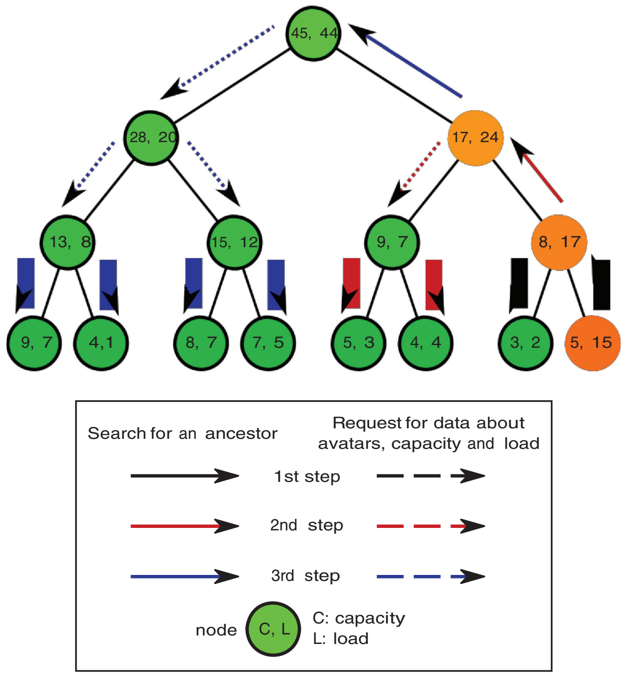
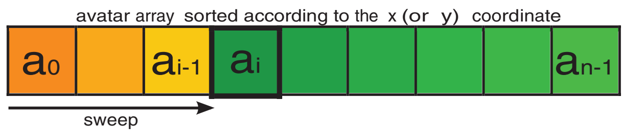


Fig. 9. Search for an ancestor node with enough resources.

Every server maintains an array with the avatars located in its respective region, sorted by the x coordinate. Also, each element of the array stores a pointer to a linked list that is ordered by the y coordinated of avatars (Figure 8). By using a local sorted avatar list on each server, the time required for balancing the load is reduced, since there is no need for the server to again sort the avatar lists sent by other servers. It only needs to merge all avatars lists received from the other servers in a unique list used to define the limits of the regions, which is done by changing the split coordinates used to partition the space.

To perform the rebalance algorithm, the overloaded server runs an algorithm that traverses up the kd-tree starting from the leaf node associated to it. This traversal moves one level up at each step until it finds an ancestor node with a capacity greater than or equal to the load. At each step, the algorithm visits a node that has a subtree containing other servers (leaf nodes), whose avatar list, load value, and capacity value are probably unknown or outdated to the initiating server. A request is sent to these servers, requiring the current avatar list and the current values of load and capacity (Figure 9). With this data, and the initiating server's own list of avatars and values of



c(left) = capacity of the node's left child
c(right) = capacity of the node's right child
split coordinate := avatar[i].getX() (or getY), where i is such that:

$$\frac{\sum_{j=0}^{i-1} \text{load}(a_j)}{\sum_{j=0}^{n-1} \text{load}(a_j)} \sim \frac{c(\text{left})}{c(\text{left}) + c(\text{right})}$$

Fig. 10. Division of an avatar list between two brother nodes.

load and capacity, the algorithm can calculate the load and capacity of the ancestral nodes visited in the kd-tree, which were not known beforehand—these values are sent on-demand to save up some bandwidth of the servers and to keep the system scalable.

Reaching an ancestral node with capacity greater than or equal to the load—or the root of the tree if no such node is found—the initiating server adjusts the split coordinates of the kd-tree nodes. For each node, it sets the split coordinate in a way such that the avatars are distributed according to the capacity of the node's children. For this, the load fraction is calculated that should be assigned to each child node. The avatar list is then swept, stopping at the index i such that the total load of the avatars before i is approximately equal to the value defined as the load to be designated to the left child of the node whose split coordinate is being calculated (Figure 10). The children nodes also have, in turn, their split coordinates readjusted recursively, so that they are checked for validity—the split coordinate stored in a node must belong to the region defined by its ancestors in the kd-tree—and readjusted to follow the balance criteria defined.

As the avatar lists received from the other servers are already sorted along both axes, it is enough to merge these structures with the avatar list of the server that initiated the rebalance. Assuming that each server already calculated the weight of each avatar managed by it, the rebalance time is $O(nlogS)$, where n is the number of avatars in the game and S is the number of servers. The communication cost is $O(n)$, caused by the sending of data related to n avatars. The merging of all avatar lists has $O(n)$ complexity, since the avatars were already sorted by the servers. In the worst case, at each level of the kd-tree, $O(n)$ avatars are swept in order to find the i index whose avatar's coordinate will be used to split the regions defined by each node of the tree (Figure 10). As this is a balanced tree with S leaf nodes, it has a height of $[log\ S]$, explaining the $O(nlogS)$ overall complexity of the rebalancing algorithm.

4. SIMULATIONS

A virtual environment with several moving avatars was simulated to evaluate the proposed dynamic load-balancing algorithm. Starting from a random point in the environment, each avatar moves according to the random waypoint model [Bettstetter et al. 2002]. To force a load imbalance and stress the tested algorithms, we defined some *hotspots*—points of interest to which the avatars moved with a higher probability than to other parts of the map. This construction allows for a high concentration of avatars in some areas. Although the model used to control player movement is not very realistic in

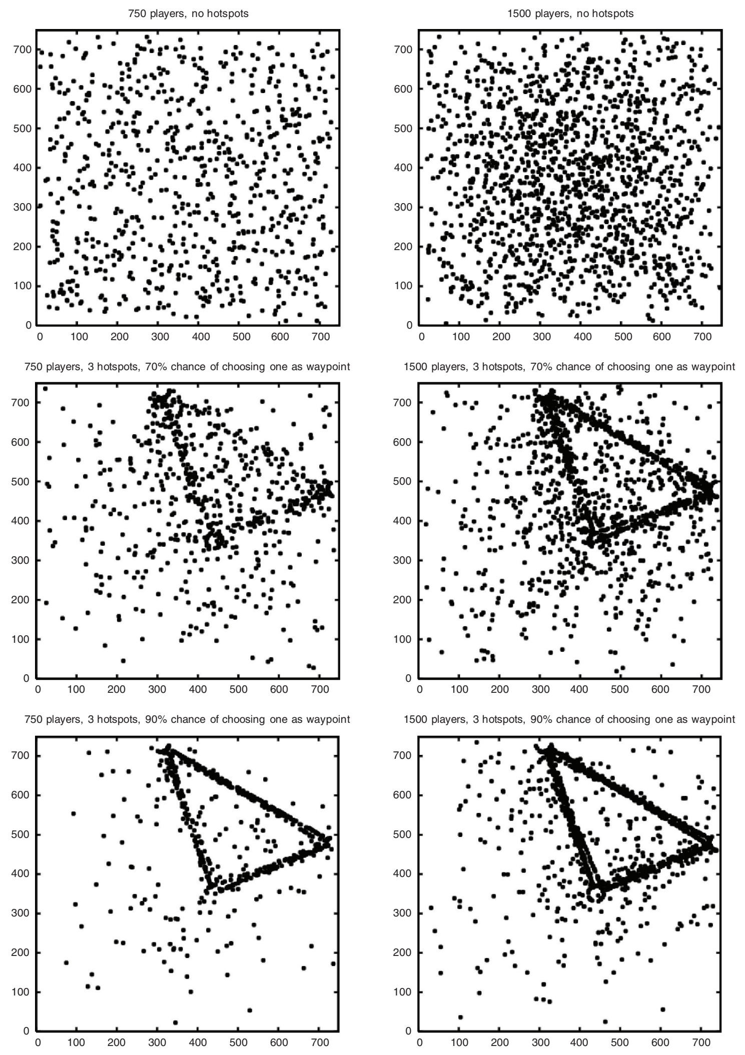


Fig. 11. Scenarios in which the algorithms were simulated.

terms of the way the players move their avatars in real games, it was only used to verify the load-balance algorithms being simulated. For each algorithm tested, we simulated scenarios with 750 and 1500 players, with and without hotspots. In the hotspot cases, we simulated different probabilities (70% and 90%) for each player, selecting one of the hotspots as the next waypoint to move. Combining all these parameters, we simulated six different scenarios for each algorithm; Figure 11 illustrates each situation.

We chose three kinds of avatar distributions—no hotspots, mildly concentrated around hotspots (70%), and densely concentrated (90%)—because we wanted to make realistic simulations. In most commercial MMOGs, there are points of interest, like towns and dungeons, through which players wander looking for adventure and treasure—for example, going to a dungeon and coming back to town once in a while—thus creating a mildly concentrated avatar population around these places. Frequently, a game company creates special game events that attract most players to some locations in the game world, such as a very powerful monster invading a town or a very rare and valuable treasure in a dungeon, creating a population density in these areas that is much higher than usual, to which the heavily concentrated simulation scenarios refer. It is rare to have absolutely no hotspots in a real MMOG (however, this case was not simulated to cover this kind of no-hotspot situation only, but to also serve as a reference for the results of the scenarios with hotspots).

The proposed approach was compared to those presented in Section 2. It is important to observe that the model employed by Ahmed and Shirmohammadi [2008] considers hexagonal cells, while in our simulations we used rectangular cells. Furthermore, Ahmed and Shirmohammadi consider that there is a transmission rate threshold that is the same for all servers in the system. As we assume a heterogeneous system, their algorithm was simulated considering that each server has its own transmission rate threshold, depending on the upload bandwidth available for each one of them. However, we kept what we consider the core idea of the authors' approach, that is, the selection of the smallest cell cluster managed by the overloaded server, then choosing from this cluster the cell with the lowest interaction with other cells of the same server, and finally transferring this cell to the least loaded server. Besides Ahmed's algorithm, we also simulated some of the ones proposed in Bezerra and Geyer [2009]: Progrega and BFBCT.

The simulated virtual environment consisted of a 750×750 two-dimensional space filled with moving avatars, each with a view radius of 50, and whose players were divided among eight servers ($S_1, S_2, \ldots, S_8$), each of which was assigned to one of the regions determined by the balancing algorithm. For the cell-oriented approaches simulated, the space was divided into a 15×15 cell grid—or 225 cells. The capacity[4] of each server S_i was equal to $i \times 20000$, forming a heterogeneous system. This heterogeneity allowed us to evaluate the loadbalancing algorithms simulated according to the criterion of the proportionality of the load distribution on the servers. In addition to evaluating the algorithms according to the proportionality of the load distribution, it also considered the number of player migrations between servers. Each migration involves a player connecting to the new server and disconnecting from the old one. This kind of situation may occur in two cases: where the avatar moved, changing the region in which it was located and, consequently, changing the server to which its player was connected; or where the avatar did not move, and its player still had to migrate to a new server. In the latter case, the player's transfer was obviously due to a rebalancing. An ideal balancing algorithm performs the load redistribution requiring the minimum possible number of player transfers between servers, while keeping the load on each server under (or proportional to) its capacity.

Finally, the interserver communication overhead[5] was also evaluated. It occurs when two players are interacting, but each one of them is connected to a different server. Although the algorithm proposed in this work does not address this problem directly, it would be interesting to evaluate how the load distribution performed by it influences the communication between the servers.

5. RESULTS

Figure 12 presents the average load (plus the interserver communication overhead) on each server for each algorithm tested. The first two graphics show the values in a situation without hotspots, and so present a lower total load. The next two graphics, in turn, present the load distribution when the players tend to move towards one of

[4]The network capacity is a value proportional to the upload bandwidth of each simulated server.

[5]The interserver communication overhead here represents the amount of bandwidth wasted by the servers when intermediating interactions of players who are in different regions. This happens because, in each of these interactions, there is more than one intermediate server.

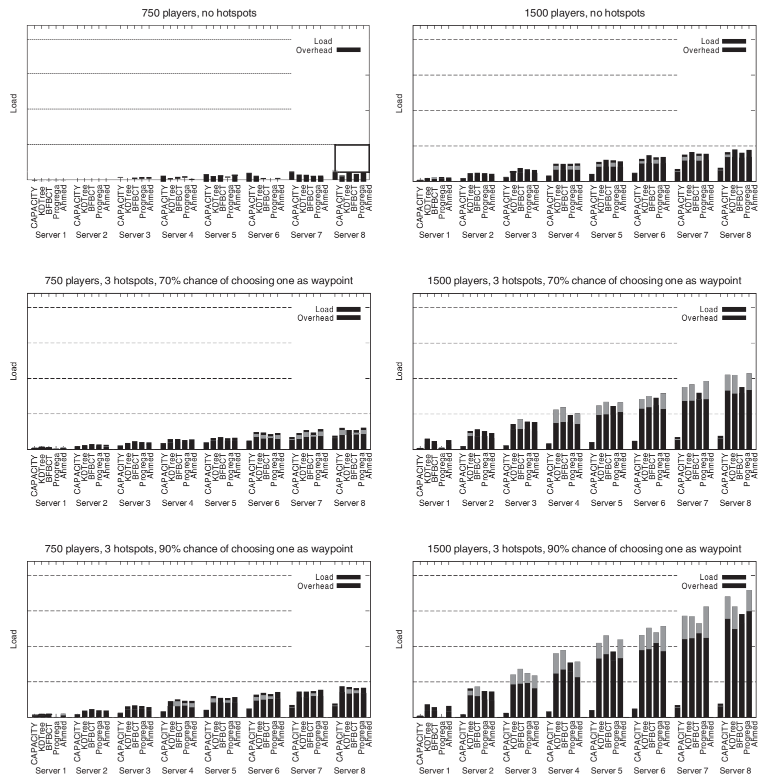


Fig. 12. Average load on each server in each scenario.

the hotspots, increasing the number of interactions, and thus the load on the servers. Finally, the two bottom graphics show the load on the servers when there are hotspots and the probability of a player moving to one of them is considerably high (90%). Although this is not the worst case—all players interacting with one another—the load increase on the servers is significant.

We can see that in all the situations, all the algorithms met the objective of keeping the load on each server lower than or proportional to its capacity. When the server system is overloaded (as in most graphics in Figure 12, apart from the first), it was demonstrated that all the algorithms managed to dilute, in a more or less proportional manner, the load excess on the servers. It is important to observe, however, that the load shown is only theoretical. Each server will perform some kind of "graceful degradation" to keep the load under its capacity.

For example, the update frequency might be reduced and access to the game could be denied for new players attempting to join, which is a common practice in most MMOGs.

In Figure 13, it is shown how much the balance generated by each algorithm deviates from an ideal balance—that is, how much on average the load on the servers deviates from a value exactly proportional to the capacity of each one of them—over time. It is possible to observe that, at least in three of the situations of greatest overload, the algorithm that uses the kd-tree has the least deviation. This is the due to the fine granularity of its distribution, which, unlike the other approaches tested, is not limited by the size of a cell. Where there is a heavy overload, the algorithm that uses the kd-tree is particularly effective because rebalance is needed. Where the system has

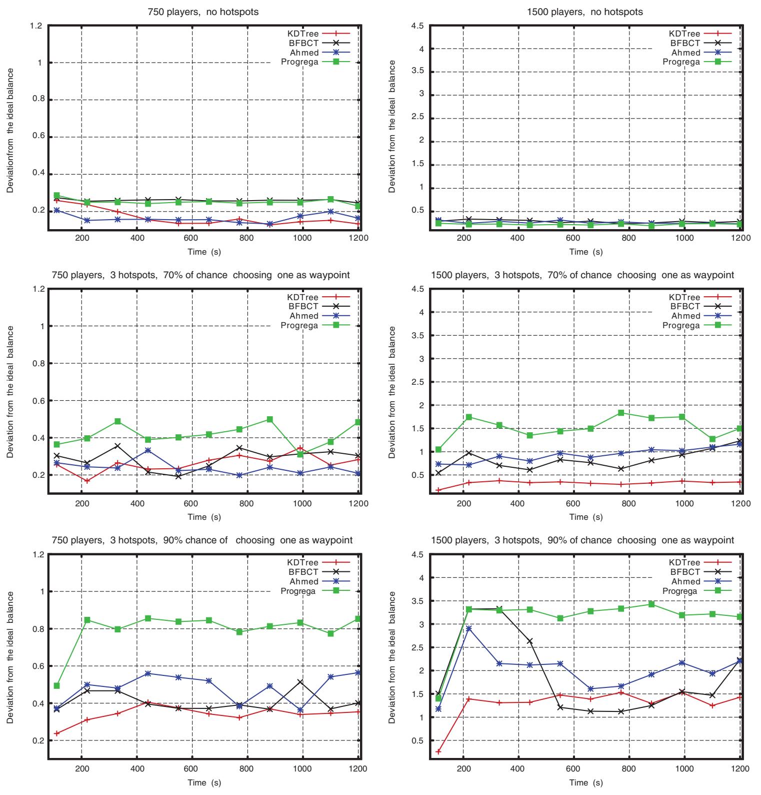


Fig. 13. Average deviation of the ideal balance of the servers in each scenario.

enough resources, proportionality of the distribution is not an important issue: it is enough that each server manages a load smaller than its capacity.

Regarding player migrations between servers, the proposed approach not only performed considerably better than the others (Figure 14), it also demonstrated a greater scalability, increasing its advantage over the other algorithms as the simulated scenario became more heavily loaded. This is due, in the first place, to the fact that the regions defined by the leaf nodes of the kd-tree are necessarily contiguous, and each server was linked to only one leaf node. An avatar moving across an environment divided into very fragmented regions constantly crosses the borders between these regions, and hence causes its player to migrate repeatedly from server to server. Another reason for this is that each rebalancing executed with the kd-tree gets much closer to an ideal distribution than the cell-based algorithms.

—again, thanks to the finer granularity of the kd-tree-based distribution, requiring less future rebalancing, and thus causing fewer player migrations.

Finally, it is shown the amount of communication between servers for each simulated algorithm over time. As we can see in Figure 15, when there are no hotspots, the algorithm which uses the kd-tree is slightly better than the others. This is also explained by the fact that the regions are contiguous, minimizing the number of boundaries between them, and consequently reducing the probability of occurring interactions between pairs of avatars, each one in a different region. However, it is also possible to see that the interserver communication with Progrega was considerably

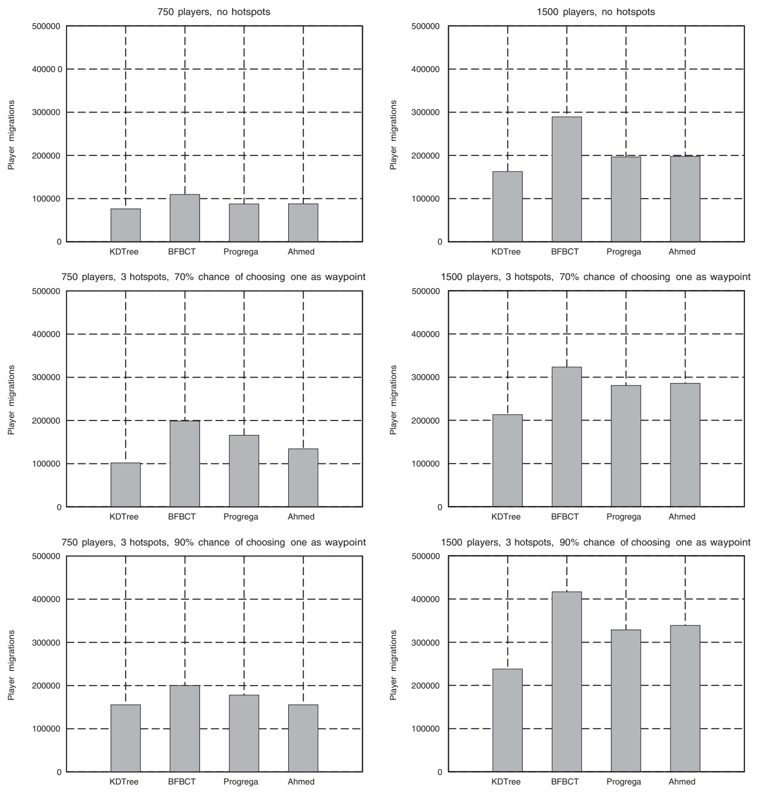


Fig. 14. Player migrations between servers in each scenario.

lower than with the other algorithms in all the situations of system overload, except when there were 750 players with 70% chance of an avatar moving to a hotspot, where Progrega and the kd-tree-based algorithm alternated in the first place. The reason is that the main goal of Progrega—besides balancing the load—is precisely to reduce the communication between servers. However, even though it was not built specifically for reducing the overhead due to interserver communication, the algorithm proposed here got second place overall in this criterion.

6. CONCLUSIONS

In this work, we proposed the use of a kd-tree to partition the virtual environment of MMOGs and to perform the load-balancing of servers by recursively adjusting the split coordinates stored in the nodes of the kd-tree. One of the conclusions reached is that the use of such data structures to make this partitioning allows a fine granularity of the load distribution, while the readjustment of the regions becomes simpler—by recursively traversing the tree—than the usual approaches, based on cells and/or graph partitioning.

The finer granularity allows for a better balancing, so that the load assigned to each server is closer to the ideal value that should be assigned to it. This better balance also helped to reduce the number of migrations, by performing less rebalancing operations. The fact that the regions defined by the kd-tree are necessarily contiguous was one of the factors that contributed to the results of the proposed algorithm, which was better than the other algorithms simulated in most of the criteria considered.

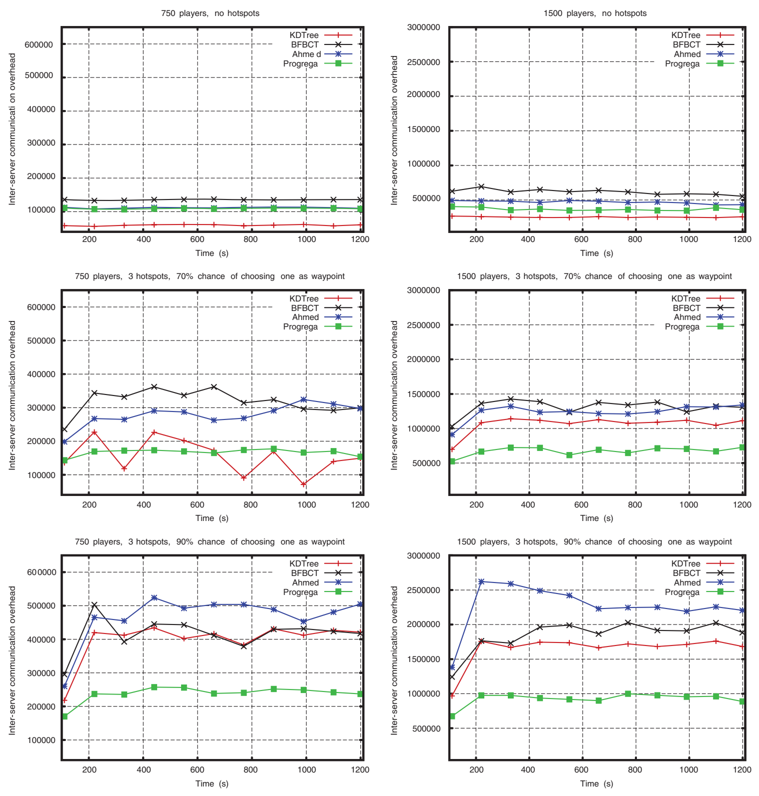


Fig. 15. Interserver communication over time for every algorithm in each scenario.

We have made an extensive set of simulations, comparing all the algorithms in six different scenarios, varying the number and distribution of avatars throughout the virtual environment of the game. By performing these tests, we concluded that the proposed algorithm not only performs better than the others, but also that it scales better, at least in the kinds of situations we considered, which we tried to make as close as possible to real MMOG scenarios—although we defined very high load values in order to push the simulated algorithms and to verify their behavior in worst-case scenarios.

In conclusion, it was possible to use methods that can reduce the complexity of each rebalancing operation. This is due, first, to the reduction of the number of operations for calculating the relevance between pairs of avatars by sweeping a sorted avatar list; and, second, to keeping at each server an avatar list already sorted in both dimensions, saving the time spent on sorting the avatars when they were received by the server executing the rebalance.

REFERENCES

Ahmed, D. and Shirmohammadi, S. 2008. A microcell oriented load balancing model for collaborative virtual environments. In *VECIMS*. 86–91.

Assiotis, M. and Tzanov, V. 2006. A distributed architecture for MMORPG. In *Proceedings of the ACM SIGCOMM Workshop on Network and System Support for Games (NetGames 5)*, ACM, New York, 4.

Bentley, J. 1975. Multidimensional binary search trees used for associative searching.

Bettstetter, C., Hartenstein, H., and Peŕez-Costa, X. 2002. Stochastic properties of the random waypoint mobility model: Epoch length, direction distribution, and cell change rate. In *Proceedings of the ACM*

International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems, 5, ACM, New York, 7–14.

Bezerra, C. E. B., Cecin, F. R., and Geyer, C. F. R. 2008. A3: A novel interest management algorithm for distributed simulations of MMOGS. In *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT),* IEEE, Washington, D.C., 35–42.

Bezerra, C. E. B. and Geyer, C. F. R. 2009. A load balancing scheme for massively multiplayer online games. *Massively Multiuser Online Gaming Systems and Applications,* Special Issue of *J. Multimedia Tools Appl.*

Blizzard. 2004. World of Warcraft. http://www.worldofwarcraft.com/.

Chertov, R. and Fahmy, S. 2006. Optimistic load balancing in a distributed virtual environment. In *Proceedings of the ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video* (NOSSDAV), ACM, New York, 1–6.

El Rhalibi, A. and Merabti, M. 2005. Agents-based modeling for a peer-to-peer MMOG architecture. *Comput. Entertain. 3,* 2, 3–3.

Gravity. 2001. Raganar online. http://www.ragnarokonline.com/.

Hampel, T., Bopp, T., and Hinn, R. 2006. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of the ACM SIGCOMM Workshop on Network and System Support for Games*, ACM, New York.

Iimura, T., Hazeyama, H., and Kadobayashi, Y. 2004. Zoned federation of game servers: A peer-to-peer approach to scalable multi-player online games. In *Proceedings of the ACM SIGCOMM Workshop on Network and System Support for Games,* ACM, New York, 116–120.

Knutsson, B. et al. 2004. Peer-to-peer support for massively multiplayer games. In *Proceedings of the IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, IEEE, Washington, D.C., 96–107.

Lee, K. and Lee, D. 2003. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, ACM, New York, 160–168.

Luque, R., Comba, J., and Freitas, C. 2005. Broad-phase collision detection using semiadjusting BSP-trees. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, ACM, New York, 179–186.

Ncsoft. 2003. Lineage ii. 2003. http://www.lineage2.com/.

Ng, B. et al. 2002. A multi-server architecture for distributed virtual walkthrough. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST)*, ACM, New York, 163–170.

Rieche, S. et al. 2007. Peer-to-peer-based infrastructure support for massively multiplayer online games. In *Proceedings of the 4th IEEE Consumer Communications and Networking Conference (CCNC)*, IEEE, Washington, D.C., 763–767.

Samet, H. 2005. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, San Francisco, CA.

Schiele, G. et al. 2007. Requirements of peer-to-peer-based massively multiplayer online gaming. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, IEEE, Washington, D.C., 773–782.

Received February 2011; accepted June 2012