Constant-Time Complexity Interest Management for Online Games

Francisco Gallego[1], Pawan Kumar[2], Qasim Mehdi[2], Abel Bernab[1] and Faraón Llorens[1]

[1]Department of Computer Science and Artificial Intelligence
University of Alicante
Ctra. San Vicente del Raspeig s/n, Alicante, Spain
{fgallego, faraon}@dccia.ua.es

[2]School of Computer and Information Technology
University of Wolverhampton
Wolverhampton WV1 1SB
{Pawan.Kumar, Q.h.Mehdi}@wlv.ac.uk

KEYWORDS

Interest Management, Computer Games

ABSTRACT

Data Distribution Management (DDM) services are very important to online services in general and, in particular, to online Computer Games. Many previous works have addressed the problem of minimizing bandwidth usage by avoiding sending unnecessary data to clients, which is often referred to as Interest Management. Many good algorithms have been developed to calculate clients' interests, having $O(n^2)$ complexity in worst case, but none of them have paid attention to the time dependencies of the data. In this paper we present a novel algorithm for Interest Management which reaches a $O(1)$ complexity by profiting from time dependencies of data related to clients' interests. Our results show that this approach improves previously existing ones in time performance, and it is specially suitable for its use in online computer game servers.

INTRODUCTION

Online games allow multiple players at geographically dispersed location to interact in real-time in a common virtual environment. In these, a player's node will contain some subset of the shared virtual world whose state is influenced and maintained by the player. In order to have a mutual consistent view of the virtual world, events (messages) are exchanged between player nodes (either directly or indirectly through a server). However, an update occurring at one node is likely to have an immediate significance for only a subset of other nodes in the system. The techniques that exploit this interest of each node to minimize the number of messages sent are referred as interest management (IM) schemes. Interest management schemes have been in existence in several large scale distributed simulators (Morse, 2000; HLA, 1998), collaborative virtual environments (Macedonia et al., 1995; Miller and Thorpe, 95; Greenhalgh and Benford, 1995) and multi player online games (Unreal, 1999; Yu and Vuong, 2005; Liu et al., 2005). These have been incorporated mainly to allow systems to scale seamlessly and efficiently in terms of users and simulation entities. Without an IM scheme, it would entail every update or state changes at one node to be communicated to all other nodes. This could significantly increase the bandwidth usage, messages sent per second and computational requirements at processing these messages.

Several IM schemes have been researched in which a node expresses its interest to some subset of the world where information pertinent to that node gets forwarded to it. Most of these schemes use a filtering mechanism to cull the information exchange among the nodes. These include *grid based*. approaches (Macedonia et al., 1995; Miller and Thorpe, 95; Tan et al., 2000) that partition the world into n-dimensional grid cells of fixed or variable size resolution. Each node subscribes to some set of cells and updates are sent only between nodes whose subscriptions fall into the same grid cell. Another scheme uses *class based* filtering (HLA, 1998; DIS, 1995) where a node expresses interest through subscription to some set of classes and exchange information regarding those classes only. Other schemes combines the class and grid based filtering with *region based* approaches (HLA, 1998; Greenhalgh and Benford, 1995; Liu et al., 2005) where simulation nodes or entities specify interest areas in form of updater-subscriber regions where an intersection between the two represents potential communication between the entities. These regions can be specified in form of multidimensional routing spaces (HLA, 1998; Liu et al., 2005), auras (Greenhalgh and Benford, 1995) or as view cones. Our work focuses on the region matching algorithm for interest management that uses multidimensional routing spaces and extends the previous work done in the area to make region matching more efficient and scalable. In our earlier paper (Kumar and Mehdi, 2006), we highlighted some of the related works in the field and presented a recursive algorithm that is scalable and efficient with the computational time complexity of $O(n \log n)$. This work provides another alternative that achieves the same region matching in constant time by using the time dependence of the updaters-subscribers. Figure 1 shows an example of a game world that is partitioned into 2

dimensions representing a *2D routing space*. Here you can see four regions (extents) where each has a bounded range defined along each dimension of the routing space. For IM, node's specifies game entities attributes or interactions that it wishes to update or subscribe and associate regions with those subscriptions. When updater regions overlap with subscriber regions, connectivity is established between the two nodes and finally the information is transmitted. Thus for scalable implementation of IM, it is important that region matching and connectivity is established as efficiently as possible. In the next section we detail our constant-time algorithm for this purpose.

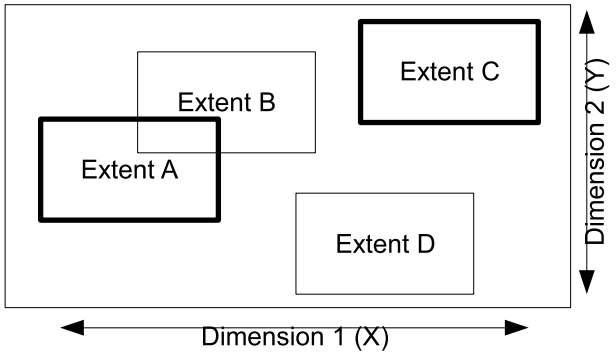

Figure 1: Routing space with defined areas of interest (extents)

IMPROVING THE SORT-BASED ALGORITHM

Our algorithm extends the basic idea of the Sort-Based DDM Matching Algorithm which was very well conceived by (Raczy et al., 2005). Therefore, the main idea remains the same: the routing space is composed of dimensions. For each dimension there is a point-vector. These point-vectors store start and end points of the projections of the extents in the corresponding dimension, as shown in Figure 2. These point-vectors are used for calculating intersections between different extents in each dimension. We consider that two extents are colliding only when they are overlapping in all the possible dimensions. Finally, a collision between two extents means that they have a shared area of interest and, therefore, data produced by the entity represented by one of them is potentially interesting for the other.

Starting with this idea, we have added a new consideration that Raczy et al. did not take into account. Multidimensional extents in any virtual world where Interest Management should be applied, correspond directly to data sources and clients. Most of the time, interests are expected to remain the same during short periods of time. Furthermore, they are not expected to vary dramatically, but just to change a few intersection relationships with most closely related extents.

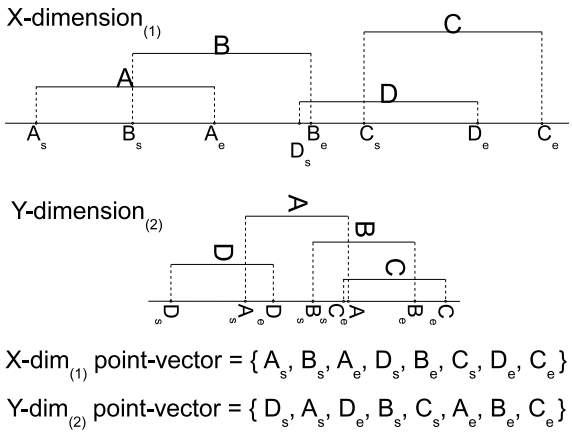

Figure 2: Dimension projections and point-vectors

This observations can be very well appreciated in the Computer Games domain. Interest Management in this domain is mostly related to proximity relations among different players. Players which are close enough to each other need to know about their actions in order to render results in their respective screens properly. Of course, players are not expected to move chaotically through the virtual world (i.e. rapidly switching from one location to a distant one in a constant fashion). They are expected to move in lines or forming some kind of curves.

This means the data is strongly time-related and we can profit from this characteristic to improve the performance of the algorithm. Our proposal consists in creating a look-up table where to store the information about present intersection relationships. Once we have stored this information, all we have to do is update it in next time step instead of recalculate it. This is based on the hypothesis stated before that updating changes should cost less than recalculating everything each timestep.

Time-Dependent Sort-Based Algorithm

As we have stated before, we consider two extents to be colliding (and, therefore, sharing interests they represent) only if they intersect in all the possible dimensions. Therefore, we are interested in knowing the number of dimensions in which two extents are now intersecting. This value is stored in a two-dimension look-up table which is called the Collision Look-up Table (CLT). Table 1 represents the CLT for our example in Figure 1. In this table extents A & C are supposed to be updaters whilst extents B & D are supposed to be subscribers. As can be seen, A & B are colliding because they are intersecting in the 2 possible dimensions (X and Y), whilst A & D, C & B

and C & D are not colliding because they are only intersecting in 1 out of the 2 possible dimensions. Intersections in each dimension can be seen in Figure 2.

	B	D
A	2	1
C	1	1

Table 1: Collision Look-up Table (CLT)

As can be seen in the class diagram in Figure 3, our algorithm considers the routing Space to be decomposed in a list of Dimensions. In each dimension, pairs of points are grouped into Regions (i.e. Intervals), which represent projections of extents (see again Figure 2) in the considered dimension. A group of regions (one region from each dimension) forms a Extent. Finally, all the extents, which correspond to the different interest areas, are also part of the routing space.

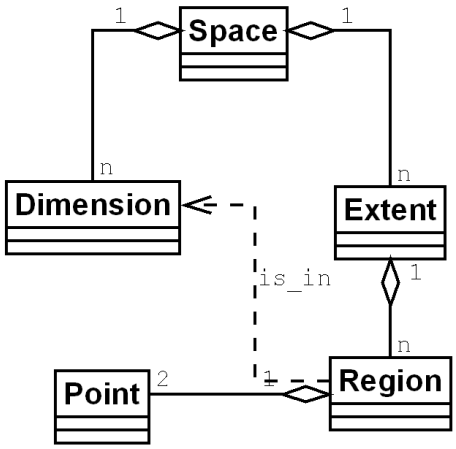

Figure 3: Class Diagram

This decomposition of the problem lets us create different routing spaces, just by instantiating the Space class. This lefts open the possibility to create space partitions just by instantiating sets of routing subspaces, which could be used for improving the algorithm. Moreover, as extents are formed by different instances of Region, they could be differently sized, which is a very convenient feature in many applications, such as Computer Games, for instance.

The core of the algorithm are the extents. Then, the most important functions to implement it are 2: adding a new extent to the world and moving an extent. Take note that, for simplicity reasons, this versions of the algorithms do not take into account extent classes (i.e. updater and subscriber). If you want to implement a fully functional version, you should add the necessary comparisons to distinguish between updaters and subscribers wherever needed.

- *Adding a new extent:* This function adds a new area of interest (i.e. extent) to the routing space. The steps to follow for this can be seen in the algorithm 1. When adding the new extent is important to insert all the points of the regions which define the extent in order. While inserting this points, the algorithm is also looking for the intersections between the new extent and the already existing ones.

Algorithm 1 Add a new extent to the routing space

Require: E = the new Extent to insert
1: Let S_{RDE} be the set of regions describing E
2: Let S_{IE} be the set of intersecting Extents
3: $S_{IE} \Leftarrow \emptyset$
4: **for all** R in S_{RDE} **do**
5: Let D_R be the Dimension containing R
6: $R_s \Leftarrow getStartPoint(R)$
7: $R_e \Leftarrow getEndPoint(R)$
8: $V \Leftarrow getPointVectorfromDimension(D_R)$
9: $i \Leftarrow 0$
10: **while** $i < size(V) - 1$ and ($V[i] < R_s$ or ($V[i] = R_s$ and $isStartPoint(V[i])$)) **do**
11: **if** $isStartPoint(V[i])$ **then**
12: add $extentOf(V[i])$ to S_{IE}
13: **else**
14: remove $extentOf(V[i])$ from S_{IE}
15: **end if**
16: $i \Leftarrow i + 1$
17: **end while**
18: $insertIn(V, i, R_s)$
19: **while** $i < size(V) - 1$ and ($V[i] < R_e$ or ($V[i] = R_e$ and $isStartPoint(V[i])$)) **do**
20: **if** $isStartPoint(V[i]$ **then**
21: add $extentOf(V[i])$ to S_{IE}
22: **end if**
23: $i \Leftarrow i + 1$
24: **end while**
25: $insertIn(V, i, R_e)$
26: $addToCLTIntersectionsBetween(E, S_{IE})$
27: **end for**

- *Moving an extent:* Changes in the interests of the entities in the world are reflexed as movements of the n-dimensional extents in the routing space. The easiest example would be a player moving inside the virtual world created by a computer game. Whenever players move,

their area of interest changes. This algorithm lays down on maintaining the CLT, which is no other thing than a look-up table with previously calculated values. Each interest change will be potentially affecting the entries in this table, so the algorithm will look for these changes as it is shown in algorithm 2.

As can be seen from the algorithms 1 and 2, the two core functions are responsible for updating the CLT, using $addToCLT(E_1, E_2)$, $removeFromCLT(E_1, E_2)$ and $addToCLTIntersectionsBetween(E_1, E_{set})$. These functions add or subtract 1 to/from the counter in the specified cell in the CLT. The first one adds 1 to the cell where E_1 and E_2 meet, indicating that these two extents are now intersecting in 1 more dimension than before. The second one, as expected, subtracts 1 telling that the two extents are not intersecting in 1 dimension anymore. The third one is an extension of the first, which adds 1 to all the cells that meet E_1 with each extent in the set E_{set}.

Of course, for online games and applications where areas of interest are likely to disappear (e.g. in the event of a player disconnecting from the server), there is the need to implement a function to remove extents from the routing space. But, as long as this function is similar to the one for inserting extents, we are not considering it for this discussion.

Analysis of Complexity

As long as the CLT is always kept up-to-date, the time needed to know if two extents E_1 and E_2 are sharing interests (i.e. they are colliding) will be the time needed to consult the value in the CLT and compare it with the number of dimensions N_d of the routing space. The two extents will only be colliding when $CLT_{E_1E_2} = N_d$. And, of course, the time needed for this operation is not depending on the number of extents N_e, meaning that this is a constant time operation $O(1)$. This is a big improvement with respect to previous works in which the complexity was always dependent of the number of Extents and, in most of the cases, $O(N_e^2)$.

Of course, this constant complexity is referred to a static state of the routing space. While the areas of interests remain unchanged, there is no need to make any calculations and the algorithm will suffice by consulting the CLT. But this algorithm has a new kind of dynamic complexity, related to some kinds of changes to the routing space. There are changes that require calculations, as can be easily deducted from the algorithms 1 and 2, and there are others that do not. The movements that require calculations are those that change the order of the points in some point-vector of any dimension. In other words, the

Algorithm 2 Move an extent in the routing space

Require: E = the Extent be moved
Require: V_{mov} = Vector of displacements
Require: V_{dim} = Vector of dimensions
Let P be a point and V_P be a points-vector
Let S_{RDE} a set of regions
$S_{RDE} \Leftarrow getSetOfRegionsDescribing(E)$
for all D_i in V_{dim} **do**
  $V_P \Leftarrow \emptyset$
  $d \Leftarrow V_{mov}[D_i]$
  $insertIn(V_P, 0, startPoint(S_{RDE}[D_i]))$
  $insertIn(V_P, 1, endPoint(S_{RDE}[D_i]))$
  for all P in V_P **do**
    $j \Leftarrow getIndexOf(P)$
    $P \Leftarrow P + d$
    if $d > 0$ **then**
      while $j < size(V_P) - 1$ and ($V_P[j] < P$ or ($V_P[j] = P$ and not($isStartPoint(P)$ and $isEndPoint(V_P[j])$))) **do**
        $E_{V_P[j]} \Leftarrow extentOf(V_P[j])$
        if $isStartPoint(P)$ **then**
          if $isEndPoint(V_P[j])$ **then**
            $removeFromCLT(E, E_{V_P[j]})$
          end if
        else {P is endPoint}
          if $isStartPoint(V_P[j])$ **then**
            $addToCLT(E, E_{V_P[j]}))$
          end if
        end if
        $j \Leftarrow j + 1$
      end while
    else
      while $j > 0$ and ($V_P[j] > P$ or ($V_P[j] = P$ and not($isEndPoint(P)$ and $isStartPoint(V_P[j])$))) **do**
        $E_{V_P[j]} \Leftarrow extentOf(V_P[j])$
        if $isStartPoint(P)$ **then**
          if $isEndPoint(V_P[j])$ **then**
            $addToCLT(E, E_{V_P[j]})$
          end if
        else {P is endPoint}
          if $isStartPoint(V_P[j])$ **then**
            $removeFromCLT(E, E_{V_P[j]})$
          end if
        end if
        $j \Leftarrow j - 1$
      end while
    end if
  end for
end for

movements that make any change to the intersections between regions in any dimension.

This means that the complexity of our algorithm depends on the number of extent movements, N_m, and the number of dimensions N_d. Intuitively, this represents an improvement with respect to depending on the number of extents, N_e, because the tendency is that $N_m \ll N_e$. However, there are two possible worst-case scenarios, which are discussed below:

1. *All the extents moving at the same time*: If all the extents are moving at the same time, and we consider all the movements to be meaningful, the complexity, attending to algorithm 2 depends on N_m, N_d and the mean number of changes (i.e. swapping points in point-vectors and updating CLT) to do for each movement, $\bar{N}_c$. This mean number is in turn depending on the density of the extents. Assuming a great density, this number could be $\bar{N}_c \approx \ln N_e$. In this case, complexity would be $O(N_d N_m \ln N_e)$. As N_d is always constant during execution, and $N_m = N_e$, because we assume all the extents to be making 1 movement each, the resulting complexity is $O(N_e \ln N_e)$.

   For instance, in an online game, this scenario is completely unlikely to occur. If we pay close attention to the requirements, not all the extents are always able to move. For instance, lots of them are usually from static entities, like a control tower, others are from non-playing characters, which could be static most of the time, etc. Moreover, it is unlikely that they were able to make meaningful movements at the same time. And, at the end, it is even more unlikely that they were able to repeat this in every timestep. So, in a real case scenario, some movements are not meaningful, and some others are. On average, we can consider the number of changes per movement to be constant, thus obtaining an $O(N_m)$ linear dynamic complexity.

2. *Chaotic movements of all extents every timestep:* Movements that require most changes are those which move an extent from one extreme of the routing space to the other in one timestep. These movements will require for each one N_e operations, as can be inferred from the algorithm 2. So, from a theoretical point of view, if it were possible to move all the extents chaotically from boundary to boundary each timestep, the complexity will be $O(N_m N_e) = O(N_e^2)$. This is an important result that tells us the performance improvement of this approach: this worst scenario should be not possible to reach in practice and has the same complexity than most previous-works worst scenarios, that are sometimes reached.

It is interesting to point out that the first worst case scenario can sometimes happen in a real execution, although the probability is very rare. The second one is studied here purely from the theoretical point of view, as it makes no sense in most of real, practical scenarios.

Of course, this analysis is only about movement of extents (algorithm 2). Inserting a new extent, as shows the algorithm 1 consist in performing a kind of insertion sort of the projection points of the extent in the point-vectors of the dimensions. As inserting extents is not the core of the method, we are not analysing the complexity deeply and just stating that inserting points in order has a complexity of $O(2N_e)$.

Although time complexity for this method represents an improvement, there also some drawbacks one should take into account when using it. In order to achieve this time performance, the algorithm is using a look-up table which could be very big, depending on the number of extents N_e. Most concretely, memory requirements for storing this table will be growing in squared relation to N_e. Extents are divided into updaters and subscribers ($N_e = N_{upd} + N_{sub}$) and, for each pair updater-subscriber you need to store the number of dimensions in which they are intersecting. So, depending on the maximum number of dimensions that a concrete application needs, the minimum store requirements (MSR) in bytes for the CLT will be represented by equation 1.

$$memory(CLT) = \frac{N_{upd} N_{sub} \lceil \log_2 (\arg\max N_d) \rceil}{2^3} \; bytes \tag{1}$$

There are also some other storage needs besides CLT's, for instance, point-vectors (PVs) for every dimension. The algorithm needs to store a point-vector for each dimension considered. As long as dimensions are supposed to be constant during execution, storage needs for point-vectors are kept linear with the number of extents. Assuming that M_p is the constant number of bytes needed to store one point, equation 2 shows point-vectors storage needs concretely.

$$memory(PVs) = 2 M_p N_e N_d \; bytes \tag{2}$$

Keeping the number of dimensions to the minimum, there should be no problem with most of todays applications to afford these memory requirements. But for some applications and, of course, for future scalability, this is a drawback of the algorithm. Further improvements are needed to reduce memory needs or to give the user the chance to switch between time performance and memory requirements in order to suit concrete needs.

RESULTS

In this section we present the results of three experiments, which give evidence to validate theoretical analysis made in this paper. For obtaining these results we have measured the time needed for performing the specific calculation tasks of inserting or moving extents. These tasks have been previously shown in algorithms 1 and 2, and the calculations involved are focused on keeping the CLT up-to-date. We have deliberately not included networking issues and application logic or any interaction between our algorithm and other piece of code required for running a real application or computer game, as it is beyond the scope of this paper. This lets us show accurate results about the main topic we are discussing in this paper: the calculations involved in knowing intersection between areas of interests of entities in a routing space.

We acknowledge that it is important to drive experiments for comparing this method with other previously studied ones. Our intention here is just to proof that our theoretical complexity results are real in practice. Knowing that these theoretical results are valid, we are also validating any theoretical comparison. Because of this, we just want to show here the evidence that supports this point. Future works on the subject should extend this and make strong comparisons with other methods to reveal improvements and weaknesses.

In our experiments we have used a standard PC with a Intel Pentium IV processor, running at 3.2 Ghz and having 2 GB of available RAM. This machine was running Debian GNU Linux 2.4.27-2-386 with gcc 3.3.5 and libc6 2.3.2.ds1-2. The source code used for this experiments is available at `http://abp.i3a.ua.es/frs/?group_id=75` under GNU GPL license.

The first experiment is about the insertion algorithm. Figure 4 shows the results of inserting 30000 extents (15000 subscribers and 15000 updaters) at random locations in a $3D$ world. Each extent has a volume of $50u^3$ (being u the unit of length) in a routing space of $10000u^3$. Firstly, we introduce the 15000 subscribers. In this first part of the graph (from 0 to 15000), we see a linear complexity growing slowly. It is growing so slowly because subscribers do not have to take care of intersections between themselves, just taking care of inserting projected points in order in point-vectors. This is not the case of the second part. There we see an step and a different pattern of grow, still linear, but more closer to $O(N_e \ln N_e)$, if we pay attention to the $\mu + 2\sigma$ line (mean + 2 times standard deviation). It is important to notice that, inserting a new extent when there are already 30000 in the world, should take less than 20 millisecond in mean, and rarely more than 45 milliseconds. This is a good, affordable, scalable measure for a computer game wishing to have such a number of players.

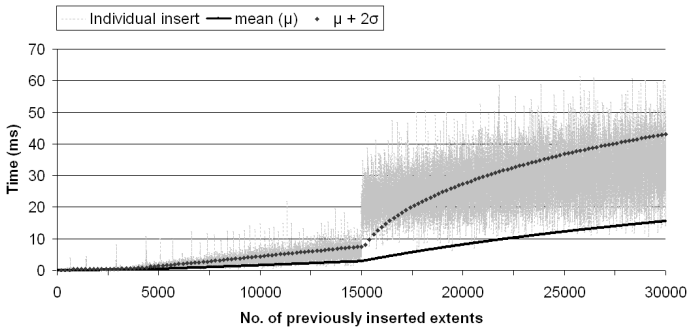

Figure 4: Time required to insert a new extent

The Figure 5 shows the results of doing 1000 extent movements in a extremely spread $3D$ world. The size of the world is $18 \times 10^8\ u^3$ whilst extents are of size $10\ u^3$. This extreme sizes are for making almost 0 the probability for two extents to overlap in any possible dimension. Data gathered also told us that no extent was overlapping neither before nor after the movements have been done. Each extent was assigned a random, constant velocity vector, with module $\left|\vec{v}\right| \in [-2u, 2u]$. Results shown in the graph make clear that the complexity of the algorithm is not depending on the number of extents, but on the number of movements. Doing the same number of movements requires the same number of processor cycles if there is no overlapping between extents at all. For this experiment, each individual time measure is the mean of 3 runs of the algorithm. Noisy measures are mainly due to context changes that the operative system performs during execution. We also may warn that it is necessary to run the experiments giving full priority to the process. It is also advisable that, depending on slight variations in the implementation, cache misses due to big amount of memory being allocated and scatterly accessed can fake performance of the algorithm, which could appear to be linear instead of constant.

Of course, not only the number of movements affects the performance of the algorithm, also the number o meaningful movements (the ones that change overlapping status of extents). Every time a extent starts or ends overlapping another one in whichever dimension, swaps between points in the correspondent point-vector are made. This is done in the lines 17, 21, 31 and 35 in the algorithm 2, within the functions for adding and removing overlaps to the CLT. Swaps have to take place to maintain points sorted when they move (see Figure 2 to get the idea). These swaps affect performance depending on the characteristics of the extents and their moves. We

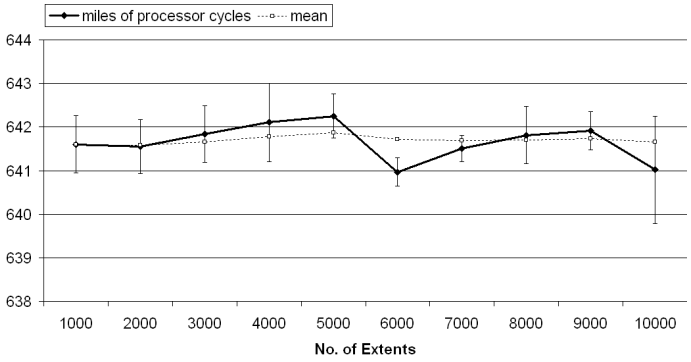

Figure 5: Processor cycles required for 1000 moves

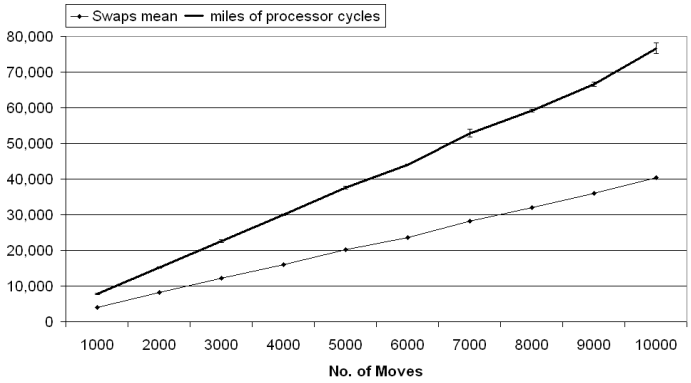

Figure 6: Processor cycles required to move extents

are mainly concerned about what might happen in a normal situation.

Figure 6 shows the increase in the number of processor cycles to process increasing number of extent moves in the same timestep. The size of the $3D$ world is $10000\ u^3$ and the extent size is $100\ u^3$ (1% of the size of the world). There are 5000 updaters and 5000 subscribers and they move with $\left|\overrightarrow{v}\right| \in [-2u, 2u]$. As our theory results predicted, in a normal scenario the complexity grows linear with the number of moves ($O(N_m)$), because not all the movements are to be meaningful and the number of swappings per movement is to be considered constant. For each value obtained in this experiment, the average of 3 runs of the algorithm was made. It is important to notice that, even if all the extents are moving the same time (10000 movements), it is feasible to invest 80 million clock cycles in calculations, which correspond to 26.7 milliseconds in a 3 Ghz computer. This could be possibly done up to $\frac{1000}{26.7} = 37.47$ times within 1 second.

CONCLUSION AND FURTHER WORK

In this paper we present a novel approach for dealing with the problem of Interest Managment in online virtual worlds. Our approach takes advantage from the time-dependencies that areas of interest exhibit in the virtual world to achieve a constant-time complexity, not depending on the number of extents. This represents a great improvements with respect to previously developed algorithms, which usually exhibit quadratic-time complexity.

The complexity is constant while the areas of interest are static. When extents move, the complexity of our approach becomes linear with the number of movements. It has been shown that the complexity also depends on the density of extents and the chaos of their movements, arriving to quadratic complexity in worst case. But it has been also shown that this worst case is completely unlikely to present in an actual scenario, and real scenarios should exhibit a linear dependency with the number of moves.

We also have stated that, for achieving this temporal performance, our algorithm creates a very big look-up table named Collision Look-up Table (CLT). This table stores a number for each subscriber-updater pair, which stands for the number of dimensions in which they are overlapping. This has the drawback of requiring a huge ammount of memory to maintain this table.

Our results confirm the theoretical predictions and show the performance of the algorithm in a usual scenario, which could be the case of a real online game. In this case, the complexity has proven to be linear and good enough as to let a 3 Ghz computer deal with 10000 extents moving at the same time in each timestep, with an output performance of almost 38 fps. Taking into account that online servers usually run at 10 fps, we assume this measure to be very good.

Although theoretical results drive us to think that this approach is faster than previous ones, practical evidence is needed to further support this claim. Therefore, our future work will focus on comparing the performance of our approach with the previously published works in practical scenarios.

Regarding the algorithm itself, several improvements could be made. It is possible to combine this algorithm with a grid-based to speed it up and, more important, to reduce the ammount of space needed for the CLT, having a smaller CLT for each cell. This approach could also bring paralellism to the algorithm, making it suitable for running in a distributed server.

REFERENCES

DIS (1995). *DIS: IEEE 1278 Standard for Distributed Interactive Simulation*.

Greenhalgh, C. and S. Benford (1995). Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, Washington, DC, USA, pp. 27. IEEE Computer Society.

HLA (1998). *USA Defence Modelling and Simulation Office High Level Architecture (HLA)- Interface Specification version 1.3*.

Kumar, P. and Q. Mehdi (2006). Recursive interest management for online games. In *CGAMES USA: proceedings of 8th International Conference on Computer Games*, Louisville, Kentucky.

Liu, E. S., M. K. Yip, and G. Yu (2005). Scalable interest management for multidimensional routing space. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, New York, NY, USA, pp. 82–85. ACM Press.

Macedonia, M. R., M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham (1995). Exploiting reality with multicast groups. *IEEE Comput. Graph. Appl. 15*(5), 38–45.

Miller, D. and J. A. Thorpe (95). Simnet: The advent of simulator networking. *IEEE 83*(8), 1114–1123.

Morse, K. (2000). *An Adaptive, Distributed Algorithm for Interest Management*. Ph. D. thesis, University of California, Irvine.

Raczy, C., G. Tan, and J. Yu (2005). A sort-based ddm matching algorithm for hla. *ACM Transactions on Modeling and Computer Simulation 15*(1), 14–38.

Tan, G., R. Ayani, Y. Zhang, and F. Moradi (2000). Grid-based data management in distributed simulation. In *SS '00: Proceedings of the 33rd Annual Simulation Symposium*, Washington, DC, USA, pp. 7. IEEE Computer Society.

Unreal (1999). *The Unreal Networking Architecture: http://unreal.epicgames.com/Network.htm*.

Yu, A. P. and S. T. Vuong (2005). Mopar: a mobile peer-to-peer overlay rrchitecture for interest management of massively multiplayer online games. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, New York, NY, USA, pp. 99–104. ACM Press.

BIOGRAPHY

Francisco J. Gallego works as assistant professor at University of Alicante and has long experience in designing and programming Computer Games. He graduated in Computer Science in 2003 at the University of Alicante and he is currently working on his PhD Thesis about Neuroevolution, Generic Machine Learning and Computer Games. His research interests cover the fields of Computer Games, Serious Games, Human-Level AI, Game-AI, Machine Learning and Softcomputing. He is a member of the AEPIA (Spanish Society for Artificial Intelligence).

Pawan Kumar is a PhD candidate at University of Wolverhampton, UK and is pursuing research in the area of Online Games. Prior to joining University of Wolverhampton, he graduated with MSc degree specializing in Games Programming from University of Hull, UK. In addition, he holds an engineering degree specializing in Computer Technology from Nagpur University, India.

Dr. Quasim Mehdi is head of the Multimedia and Computer games in the school of computing and information technology, University of Wolverhampton, UK. His research interests include AI, Computer Simulation and Modelling, Mobile Devices, Graphic and visualisation, Computer Games and Multimedia Technology. He had published more than 90 papers in international conferences and journals. Dr. Mehdi is the founder member and the general conference chair of annual Game-On Conference on Intelligent Games and Simulation, and International Conference on Computer Games.

Faraón Llorens received his degree in Computer Science from Polytechnic University of Valencia in 1993 and his PhD from the University of Alicante in 2001. His research activity in the Industry and Artificial Intelligence (I3A) Group has been centered in the field of Artificial Intelligence (Logics, Reasoning and Intelligent Agents) and the application of digital technologies to education (Didactics of Computing, Intelligent Tutoring…). He is a member of the AEPIA and the AENUI (Spanish Society for Computer Science Education). Since January 2005, he is pro-vicechancellor of "Technology and Educative Innovation" at University of Alicante.