Geometric Algorithms for Message Filtering in Decentralized Virtual Environments

Yohai Makbily
Computer Science Dept.
Technion
Haifa 32000, Israel
myohai@cs.technion.ac.il

Craig Gotsman[1]
Virtue Ltd.
P.O.Box 199
Tirat Carmel 30200, Israel
gotsman@virtue.co.il

Reuven Bar-Yehuda
Computer Science Dept.
Technion
Haifa 32000, Israel
reuven@cs.technion.ac.il

Abstract

Distributed virtual environments impose a heavy load on the network upon which they reside. Bandwidth is a potential bottleneck because *n* users imply $O(n^2)$ update messages per time unit, which is prohibitive for a large number of users. Efficient message filtering is called for, in both centralized systems, having a central server, and decentralized systems, having no central server. We solve the message filtering problem for decentralized multi-user systems based on geometric virtual worlds, popular in interactive 3D graphics applications. This is achieved by exploiting the visual relevance relationship (based on proximity, visibility and direction criteria) between pairs of users to compute mutually irrelevant regions in user parameter space. These regions are then used as update-free regions (UFR's); i.e. no communication between users is required while they are in their respective regions. Geometric algorithms for computing UFR's for the proximity, visibility and direction relevance criteria are described. Our implementation and experimental results show that the message filtering algorithm is output-sensitive. Use of our algorithms is especially effective where messages are sent through a slow communication network, such as the Internet.

Keywords: distributed systems, virtual reality, message filtering.

1. Introduction

Distributed multi-user systems, where many agents roam within a virtual environment, have recently gained popularity. Two major technical advances drive this trend: computer graphics and computer networks. With the advent of the Internet and the virtual reality modeling language (VRML) [1], 3D shared environments are a reality. Commercial VRML-based examples are Blaxxun's Community Server [2] and Sony's Community Place [3].

Distributed environments can be implemented on two kinds of system architectures: centralized or decentralized. Centralized systems are typically based on a server (or servers) connected to a large number of clients. All information sent between clients passes through the server. Despite its simplicity, the client-server architecture has serious drawbacks. The first is latency: using an intermediate relay between two clients can cause significant communication delays (consider two clients located in the same city and the server in another continent). The second is scalability: handling the multitude of messages generated by a large number of clients can seriously overload any single server machine. $O(n^2)$ messages per time unit must flow through the server (between all pairs of clients) to keep *n* clients up-to-date. To make this manageable,

[1] On sabbatical leave from the Technion – Israel Institute of Technology.

the server must reduce the number of outgoing update messages by employing message filtering techniques (e.g. the RTIME "Intelligent message filtering" technology [4]). The third drawback is the vulnerability of the system to server failures, which may paralyze the entire system. Because of these drawbacks, a decentralized implementation is to be preferred.

In decentralized systems, or so-called point-to-point systems (e.g. Earnshaw et al's VRMUD environment [5]), communication between clients is done directly. This minimizes latency. However, the scalability problem remains, since requiring that all clients have an up-to-date knowledge of the states of other clients implies that each of *n* clients will receive *n-1* messages per time unit. Here too, message filtering techniques are required to reduce the load to the bare minimum. This is potentially possible since not every message from every client is relevant to every other client. Examples of relevance relations are proximity ("is B too far from A for A to care about ?"), direction ("is A looking at B ?") and visibility ("is B visible to A ?"). The main question then becomes how can each client be sent *only* what it needs and not much more than this.

Existing message filtering algorithms in distributed systems are suitable primarily for the centralized model. The filtering is done relatively easily by the server which constantly has an up-to-date knowledge about all the clients. In this system there is one fully-updated entity, the server, and a collection of partly-updated entities, the clients, who are guaranteed to have all *relevant* up-to-date data. In decentralized systems, with no central server, message filtering is much more complicated. In these systems, contradicting requirements are to be met: intelligent decisions are to be made at each client, but yet no client is guaranteed to have full knowledge about the world.

The following example demonstrates why the decentralized message filtering problem is difficult: Consider a simple environment consisting of two agents (clients) A and B. Furthermore, assume that A and B cannot see each other (for example, they are occluded from each other), namely, they are mutually irrelevant. In this situation B should not update A (of its movements), because A has no use for these updates until B becomes visible. However, for B to continue to make the correct decision whether to update A, B must always know where A is, namely, A must always send messages to B. The same holds for A: In order to decide correctly, B must always update A. The result is a paradox: A and B must *always* update each other in order to be able to decide when *not* to update each other. For an example of an erroneous situation which might arise if care is not exercised in these situations, see Fig.1.

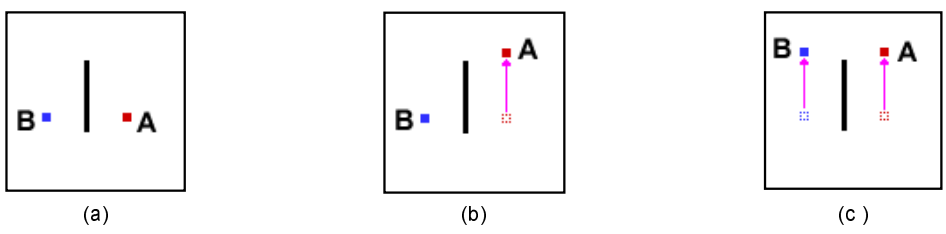


Figure 1: An erroneous situation arising in decentralized multi-user systems: (a) A and B are located on opposite sides of an occluder. (b) A moves upwards, not updating B since A's trajectory is invisible to B. (c) B also moves upwards, not updating A, since B is invisible to A according the information B has about A. The result is inconsistent: A and B think they do not see each other, even though they do.

Achieving absolute message filtering, i.e. sending *only* relevant updates, is probably impossible, but it is possible to decrease the messages traveling in the network to almost only the relevant ones. This paper shows how.

The general technique is as follows: if a pair of agents are irrelevant, it is possible to compute two regions in their parameter space containing the agents' respective states, such that as long as each agent is in its respective region – they remain irrelevant. These regions are called *update-free regions* (UFR's) (see Fig. 2). The protocol for updates between each pair of agents will then be as follows: as long as each agent is inside its UFR - updates need not be sent, and the agents are silent. When an agent exits its UFR, a mutual update is required. From that time on - updates are transmitted between the agents and relevance is tested for at every time step. As soon as the agents become irrelevant again, new UFRs are computed, the agents fall silent, and the process iterates. See Fig. 3 for an example. The key is to design the UFRs as cleverly as possible, so as to maximize the time until each agent exits his respective region. This will minimize the number of updates.

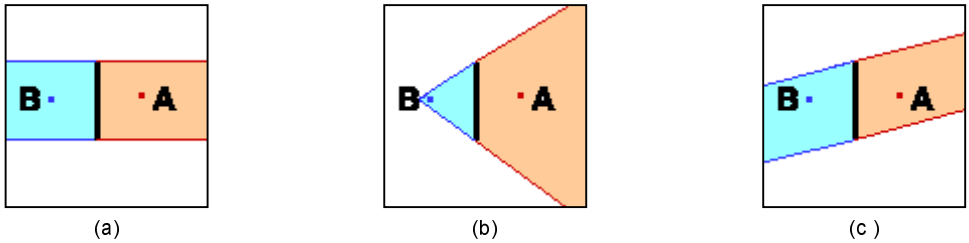


Figure 2: Possible UFR's for the example in Fig.1. As long as A and B do not exit their respective UFR's, no communication is required between them.

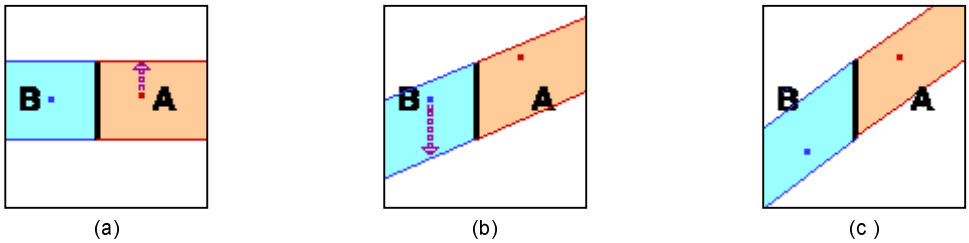


Figure 3: UFR's for message filtering purposes, computed during agent movement: (a) Initial states and UFR's. A moves upwards. (b) A exits UFR, resulting in new UFR computation. B then moves downwards. (c) B exits UFR, resulting in another UFR computation.

When the relevance relation is symmetric (e.g. proximity or visibility), either both agents should be silent (while irrelevance holds) or constantly transmitting (when relevance holds). If the relation is asymmetric (e.g. direction), situations may arise when just one agent is constantly transmitting, and the other silent. In this case, since at least one agent (the silent one) is always up-to-date, UFR's are not required. In effect, the up-to-date agent functions as a mini-server.

The remainder of this paper is organized as follows: The next section surveys previous work related to message filtering. Section 3 describes the general UFR method and a user behavioral model on which the UFR computations are later based. Section 4 describes concrete geometric algorithms to compute 2D UFR's for the proximity, direction and visibility relevance relations. Not surprisingly, the most interesting, and

complex, algorithm emerges from the visibility relation. We measure the performance of our algorithms empirically in Section 5, and conclude in Section 6.

2. Previous Work

This section surveys existing message filtering methods, designed mostly for centralized multi-user systems, and discusses if and how they are applicable to decentralized systems.

Consider, for example, the visibility relation. A central server which must decide if an update originating at A must be sent to B, has to perform a geometric line-of-sight (LOS) calculation between A and B. To perform optimal message filtering, the server must perform this computation online for all pairs of agents at each time step, which is prohibitive. Distance computations for determining the proximity relation, as in COMMIC [6], DIVE [7] and Virtual Society [3] are less expensive, so more tolerable, but still a significant load on any server.

In order to save online relevance computations, but still be able to perform reasonable message filtering, the RING [8], CLOVES [9], NPSNET [10] and SPLINE [11] systems partition the 3D virtual world to cells (with various regular or irregular structures). Two cells are said to be *relevant* if at least one point in each cell are relevant. In a preprocessing phase, the relevance relationship between every pair of cells is calculated and stored in a map at the server. Note that this is applicable only to static geometric worlds, and, for a typical virtual world, most pairs of cells will be relevant, especially if the cells are large. The relevance map is exploited at run-time when the server sends the updates originating at agent A only to other agents in cells that are relevant to the cell containing A. However, the technique is conservative: the agents might be irrelevant even though they are located in relevant cells. This results in unnecessary updates, so is suboptimal. Increasing the map cell resolution will increase the effectiveness of the map, at the expense of more storage space.

At first glance, it might seem that cell-to-cell relevance techniques are applicable to message filtering in decentralized systems by storing the relevance map at each client. Besides this technique being limited to only static scenes, a second glance reveals that the only benefit that may be reaped from the relevance map is that two agents may be silent as long as they are in mutually irrelevant cells, *and they both know that they are in their respective cells*. This is the simplest form of UFR's. Once an agent A exits its cell, it must break its silence to B, lest A's new cell is relevant to B's.

The temporal bounding volume (TBV) visibility algorithm for dynamic scenes, proposed by Sudarsky and Gotsman [12], is applicable to message filtering in decentralized virtual environments. The TBV is a volume guaranteed to contain a dynamic agent from its creation until some expiration date based on some knowledge of the agent's behavior (for instance, its maximal velocity). To perform message filtering, an agent A maintains a TBV for each other agent in the system. Once the TBV associated with B expires or becomes relevant (e.g. visible), A requests from B an update on B's status, as B might be relevant now. Until then B was silent. The problem with this technique is that it is concerned with notions of *time* ("expiration date"), while the visual relevance relation is determined by only *spatial* data (e.g. the

location). Moreover, using this technique, agent A computes a TBV for agent B based only on B's data, as opposed to UFR's, which are computed based on both A's and B's data. Hence, the TBV solution is suboptimal.

When agent behavior is somewhat predictable, *dead reckoning* [10] may be used to further reduce the message load. Here both agents jointly assume some predictable behavior of each other, and are silent as long as they indeed behave that way and are irrelevant based on that behavior. The first agent to break the behavior pattern transmits an update, and this reiterates. It is clear that dead reckoning is useful only for scenarios where most agent movements are predictable.

3. The General UFR Method

To perform message filtering in decentralized systems, we must provide each agent with a decision procedure on what to send to other agents in the system, or, in other words, which of the potential *n-1* updates may be filtered (not transmitted).

Our general message filtering algorithm, run at each agent independently, is outlined in Fig. 4, and can be described as follows: A region is a continuous set of states an agent can have in its parameter space (location, direction, etc.). An *Update-Free-Region* of an agent A, associated with another agent B (A.UFR[B]), is a region such that as long as A's state is located within the region, it does not need to communicate with B. Computing UFR's on the fly for each pair of agents is the key to the message filtering algorithm. For polygonal environments, these UFR's turn out to be polygonal, so checking containment in a UFR is a simple point-in-polygon query. While each agent is in its respective UFR, messages between the two agents may be filtered, because the agents are mutually irrelevant. Once one of the agents exits its UFR, updates must be sent. As soon as irrelevance holds again, the UFR's are recomputed, and the process iterates. Optimal UFR's are those containing the agents for as long as possible. This implies that the UFR's should have maximal areas, namely, there will be no possibility of enlarging any UFR while keeping the regions mutually irrelevant. The pair of UFR's should also be balanced (if the agents' behavior is identical). Since the UFR must be computed in near real-time, a fast algorithm is helpful. Constraints on the agents behavior may simplify the UFR's computation. We present algorithms for the case where the motion of the two agents are unconstrained, and their behavior identical.

```
while (1)
    if (A.state not in A.UFR[B]) then       // communicate
        A.send(B,A.state);
         st = A.receive(B);
        if not relevant (a.state,st) then A.calc_UFR[B](st);
    end
    if (st=A.receive(B)) then          // received update  ?
         if (B is waiting) then A.send(B,A.state);
        if not relevant (a.state,st) then A.calc_UFR[B](st);
    end
    A.change_state();
End
```

Figure 4: Message filtering algorithm for agent A updating agent B.

The effectiveness of the UFR algorithm depends on the agent's behavior. If the agent advances consistently in one direction, it will soon approach the UFR's boundary and stray the UFR, resulting in communication and possibly UFR recomputation. If the agent moves in a random walk, it tends to stay close to its initial state and remain within the UFR, staying silent for a long period of time. The ideal, but least interesting case, is when the agents do not move at all, forever remaining within their UFR's, hence silent ad infinitum. The following model captures possible agent behavior: at each step the agent advances a fixed distance in a direction picked at random within a fixed range of directions, denoted by the parameter *a*, relative to its initial state. See Fig. 5 for examples. Larger values of *a* describe more random behavior. Values of *a* less than 180 describe movement in which the agent's distance from the initial state can only increase. We believe the model most representative of a typical (human) agent's behavior is *a* =180. In this case there is a good balance between the agents' tendency to distance itself from its origin, but yet have some random behavior.

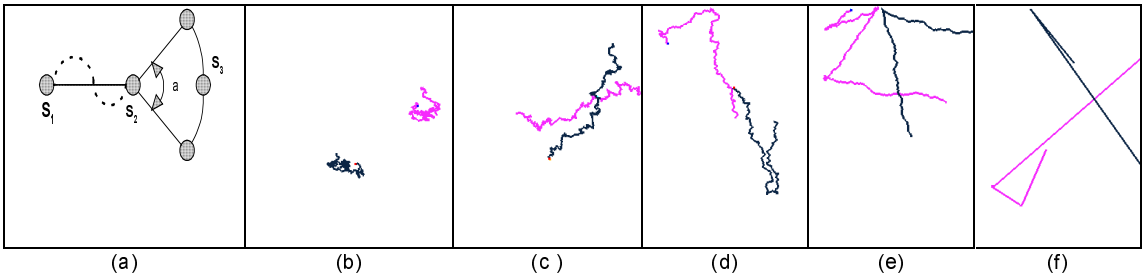


Figure 5: The agent behavior model (a) The agent can advance from its current position *s2* in any direction contained in an arc of *a* degrees around the line between s2 and its initial position s1. (b) Trajectories of two agents for *a*=360 (random walk) (c) *a*=270 (d) *a*=180 (e) *a*=90 (f) *a*=0 (deterministic walk).

As Fig. 2 shows, given states of two agents there are many possible ways to define UFR's. Which is to be prefered ? We answer this with the following argument:

Given the initial location in parameter space of an agent A, what can another agent B deduce about the whereabouts of A after some time t ? Only that A will have moved some distance from the initial state, Since many directions are equi-probable, B may assume that A has "moved" simultaneously in all directions. The same holds for A's knowledge of B. Since the agent continuously distances itself from the initial state, this movement may be simulated by a imaginary simultaneous expansion of the agents in space from their initial states. We call this process the *mutually expanding* process, and use it to define optimal UFR's: Each agent, starting from its initial state, "conquers" a state in the space if the state is irrelevant to all other states conquered so far by the other agent, and it reaches it before the other. The process stops when no more states may be conquered. Each agent's conquered region can then function as its UFR.

Fig. 6 is an sequence of snapshots of the expanding process in the scenario of Fig. 1. It shows that the optimal UFR's in this scenario are those of Fig. 2(a).

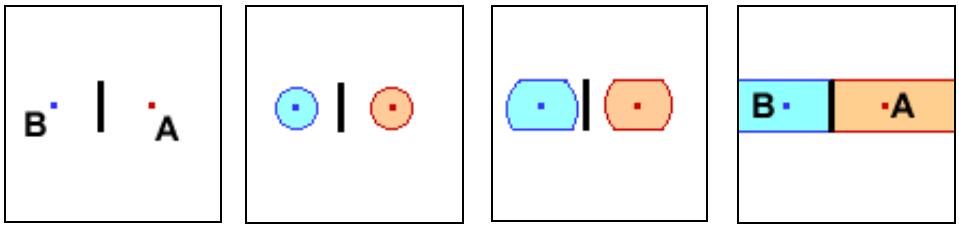


Figure 6: Visibility-based mutual expanding process used to define optimal UFR's: (a) Initial states (b) Some time later, the agents are simultaneously expanding and still are invisible from each other. (c) Some time later, the boundaries of the UFRs start to form. (d) Final UFR's - the regions cannot be extended.

The UFR Definition

Let *Dist(agent, state)* be the distance for an *agent* to travel from its current state to the given *state* (this is not necessarily the Euclidean distance if there are obstacles)
Let *R(state1, state2)* be the relevance relation, between agent2 in state2 to agent1 in state1. *R(state, path*) holds iff there exist $s \in path$ such that *R(state,s).* The path is a collection of states.

The Update-Free Region of A relative to B (A.UFR[B]) is the maximal set of states such that:

$$\forall p \in A.UFR, \forall q \in World: \left[R(q,p) \rightarrow \exists s \in A.UFR\left[R(s, Path(B,q)) \rightarrow Dist(A,s) \geq Dist(B,q)\right]\right]$$

In words: Point *p* is in A's UFR if for any other point *q* that *p* is relevant to, there exists a point *s* in A's UFR such that the path from agent B to *q* is relevant to the point *s* and the time for agent A to reach *s* is at least the time for agent B to reach *q*.

Note that since equal velocities are assumed for the agents, we have replaced the notion of time with distance ("the time to reach is less than" is replaced by "the distance to is less than").

4. UFR Algorithms

We now present geometric algorithms for computing UFR's for three different relevance relations: proximity, direction and visibility. The algorithms operate on 2D ("flatland") scenes, which, in many scenes, may be considered as the 2D projection of a 3D scene (e.g. in buildings with walls). The proximity and direction-based UFR algorithms may be generalized easily to true 3D scenes.

4.1 Proximity-Based UFR's

The proximity-based relation is defined as follows:

$$\forall agent, state \in W : R(agent, state) \Leftrightarrow (dist(agent.location, state.location) \leq agent.relevanceDist)$$

To compute the UFR, a perpendicular is extended through the midpoint of the segment AB. The UFR boundaries are then parallel to this perpendicular, at equal distances from it.

```
Line = linethrough(A.location,B.location);
perpLine = line.getPerendicularAt(middleOf(line));
A.UFRborder = perpLine.move(A.side, relevanceDist/2);
B.UFRborder = perpLine.move(B.side, relevanceDist /2);
```

Figure 7: The proximity-based UFR algorithm.

Fig. 8(a) shows a sample output from the proximity-based algorithm. It is obvious that the regions computed by this algorithm are mutually irrelevant: it is not possible to draw a line segment between two points in each region respectively that is longer than the distance between the parallel lines. This algorithm requires O(1) time.

(a) (b)

Figure 8: Outputs from the proximity-based UFR algorithm: (a) Agents A and B have same relevance distance. (b) When A is less mobile than B, the algorithm may be generalized to yield this result.

4.2 Direction-Based UFR's

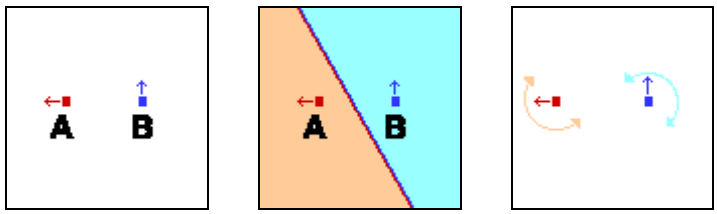
The direction-based relation, a function of both agent view direction and location, is defined as follows:

$$\forall agent, state \in W: R(agent, state) \Leftrightarrow (state \in agent.fieldOfView)$$

Since the direction relation is not symmetric, the UFR's for direction-based relevance are a combination of two types of regions: a 2D region in location space, and a 1D region in direction space, which is a sector of angles. This means that an agent will not have to send update messages only as long as it is in both these regions, namely, it has freedom to move in the location space UFR, and turn in any direction (along its own axis) in the direction UFR. Since the regions are not independent, the maximal UFR is not well-defined, as each UFR may be enlarged at the expense of the other: We can allow more freedom of movement if the directional movement is limited. However, it is always possible to allow a direction UFR of at least 180 degrees without compromising the area of the location UFR. Fig. 9 outlines the algorithm and Fig. 10 some sample output.

```
l1 = linethrough(A.location , B.location);
mid = middleOf(line);
angle = middleOf(A.viewDirection, B.viewDirection);
UFR.location.border = lineAtAngle(mid,  angle);
UFR.direction.border = angle;
```

Figure 9: The direction-based UFR algorithm. The direction UFR's are 180 degrees wide, and the location UFR's are two half planes.



(a) (b) (c)

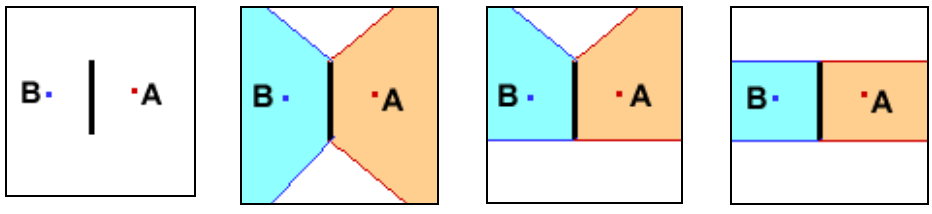
Figure 10: Output of the direction-based UFR computation algorithm: (a) Agent locations and view directions. (b) UFR's in location space. (c) UFR's in direction space.

4.3 Visibility-Based UFRs

The more interesting, and complex, relevance relation is visibility, i.e. the existence of an unobstructed line of sight (LOS) between two agents. The definition of the visibility-based relation is as follows:

$$\forall agent, state \in W : R(agent, state) \Leftrightarrow LOS(agent.location, state.location)$$

The UFR computation algorithm for this relation actually mimics the "dual" of the mutually expanding process. For a given pair of agents, each agent computes the UFR of *both* agents simultaneously, and afterwards discards the UFR of the other. The computation proceeds as follows: A finite set of strategic *viewpoints* is maintained for each agent, initialized to their initial locations. As the viewpoints are accumulated, the UFR's shrink to correspond to the intersection of all the regions occluded from these viewpoints. Hence, the UFR of A is initially the region occluded from B, and vice-versa. The viewpoints are generated by tracking discrete *events,* which are points in time where changes in the behavior of the mutually expanding process occur. See Fig. 11 for examples of such events. The events are related to the occluders' vertices (end-points if the occluder is a line segment). The process terminates when all events have been exhausted.



(a) (b) (c) (d)

Figure 11: Computation of visibility-based UFR's: (a) Agent locations. (b) First approximation to UFR's are occluded areas. (c) Partition event shrinks the UFR's by setting their lower boundaries to a line through the bottom occluder vertex. (d) Partition event again shrinks UFR's by setting their top boundaries analogously.

There are four kinds of events: add, partition, watch and watch-prevention.

Add: This is the event in which the expanding process behavior changes by encountering an occluder vertex. This generates a new viewpoint (see Fig. 12).

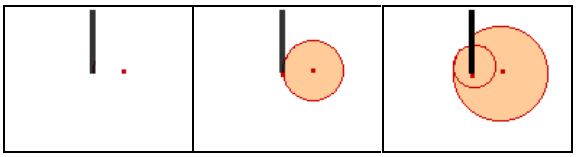


Figure 12: The Add event: Snapshots taken from an expanding process (a) Initial state. (b) Hitting an occluder vertex generates a new viewpoint (c) The two viewpoints continue to expand simultaneously.

Partition: This is the event in which mutual visibility is about to occur. It is subsequently prevented by creating a boundary line for the UFR's (see Fig. 13).

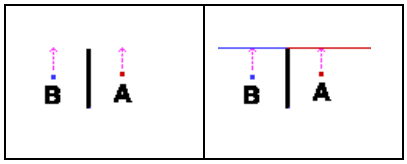


Figure 13: The Partition event: (a) Agents moving in expansion process and about to become mutually visible. (b) When they become mutually visible, a new viewpoint is generated for each agent, and so a partition line is formed.

Watch: This is the event in which an agent can see a vertex, i.e. reach a point from where a formerly occluded vertex is now visible.

Watch Prevention: This is the event in which an agent reaches a point from where it prevents the watch event by the other agent.

The complete UFR computation algorithm is outlined in Fig. 14.

```
// Initialize agent viewpoint sets.
A.viewpoints = { A.location };
B.viewpoints = { B.location };
// Generate events and their locations.
Event_set = {};
for (each occluder vertex v)  Event_set.add(world.calcFirstEvent(v));
// Initialize agent UFR's.
A.UFR = world.calcOccludedRegionFrom(B);
B.UFR = world.calcOccludedRegionFrom(A);

while (not Event_set.isEmpty() ) {              // does not contain events with infinite priority.
    event = Event_set.getElementWithMaxPriority();                    // event closest in time.
    event.owner.companion.updateUFR(event. Location);       // update UFR by intersection.
    If (event.type == Add)
       event.owner.viewpoints += event. Location;            // Add event generates viewpoint.
    if (event.type == Add || event.type== Partition) Event_set.remove(event);
       else event.update();                                                  // other types of events.
    for (each event in Event_set)  event.update();  //update priorities of all remaining events.
}
```

Figure 14: The visibility-based UFR algorithm.

Observations

1. Since an agent's UFR is occluded from the other agent, the boundaries of this region consist also (besides occluder edges and world boundaries) of lines through occluder vertices.
2. Only *connected* UFRs are computed, since moving between two unconnected occluded areas forces the agent to be visible to each other.

Fig. 15 shows some examples of UFR's computed by the algorithm in some multi-occluder environments.

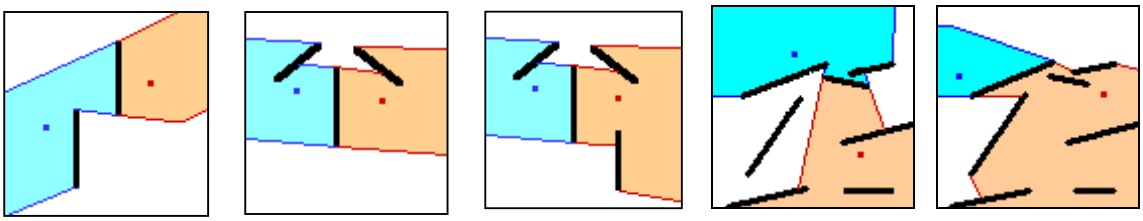


Figure 15: Sample UFR's computed by the visibility-based UFR algorithm.

Proof of the correctness of this algorithm is beyond the scope of this paper. We will just mention that its complexity is $O(n^3)$, where n is the number of occluder vertices in the scene.

For static scenes, some of the complexity may be reduced by using a pre-computed cell-to-cell visibility map stored at each client. Online, the cell-to-cell visibility map is first used to determine mutually invisible cells. These are immediate parts of the UFR's. For pairs of cells which are mutually visible, use our UFR computation method with the cells' occluding walls and other occluders between the cells. This approach minimizes the number of occluders involved and can save much of the computing time.

5. Experimental Results

Experimental tests were made by running implementations of the distance-based and visibility-based UFR computation algorithms. Since updates must be sent continuously if relevance holds, and none must be sent if irrelevance holds, the success of the algorithm is measured by the ratio of the number of updates sent to the number of irrelevant time steps. Ideally, it should be zero.

The performance of the UFR algorithm depends on the agent's behavior. In our experiments, we simulated agents in a scenario with 3-4 occluders, moving according to the model described in Section 3, with a taking values in the set {0,90,180}.

More than 400,000 agent trajectory pairs were simulated for each a. Each trajectory pair contains periods of relevance and irrelevance. While relevance holds, messages are constantly sent. While irrelevance holds, the UFR algorithm prevents many messages from being sent, but some still leak through, counted by our monitor. The results are shown in Fig. 16. The conclusion from these graphs is that the number of messages sent during the irrelevance period tends to a fixed low value as the period grows longer. This means that where filtering is needed most, namely in long periods

of irrelevance, the success is higher. In situations where the irrelevant period is very short - the filtering is less successful.

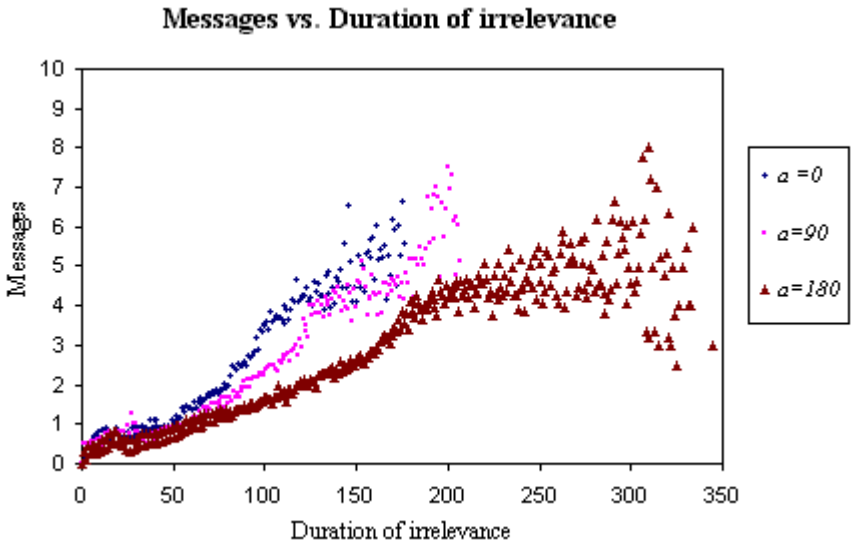


Figure 16: Experimental results. The number of updates is far less than the duration of irrelevance (for *a*=180 it levels off at less than 2%).

6. Conclusion

We have described a scheme for message filtering in decentralized multi-user virtual environments. The scheme is based on defining and computing on-the-fly update free regions (UFRs) in agent parameter space. These regions are mutually irrelevant regions: no communication is required while the agents are in their respective regions. Algorithms to compute 2D UFR's for the proximity, direction and visibility criteria were described. The algorithms are suitable also for dynamic scenes.

Experimental results show that in cases where filtering is very much needed - the algorithm achieves impressive update savings, and, for a pair of agents, is output-sensitive, namely that almost all the irrelevant messages are filtered. Our message filtering technique may be combined with existing techniques and used in a hybrid client-server and client-to-client systems.

The UFR technique, as described here, is applicable only to a pair of agents. In a multi-user environment, applying this technique for every other agent is a significant burden on any agent. The agent must determine at every time step whether its state is contained in the UFR associated with every other agent (a point-in-polygon operation). In most cases, the result will be negative. It seems possible, using appropriate geometric data structures, to speed up these computations by exploiting the temporal coherence present in the agents' behavior. This implies a fully output-sensitive algorithm for the entire system, but requires further research to formulate a complete solution.

References

[1] J. Hartman and J. Wernecke. The VRML 2.0 Handbook, Addison-Wesley, 1996.
[2] http:\\www.blaxxun.com
[3] R. Lea, Y. Honda, K. Matsuda, S. Matsuda. Community Place: Architecture and performance. *Proceedings of Second Symposium on the Virtual Reality Modeling Language (VRML '97),* pp 41-50, 1997.
[4] http:\\www.rtimeinc.com
[5] R.A. Earnshaw, N. Chilton and I.J.Palmer. Visualization and virtual reality on the Internet. *Proceedings of the Visualization Conference*, Jerusalem, Israel, Nov. 1995.
[6] S. Benford, L. Fahlen, C. Greenhalge, and J. Bowers. Managing mutual awareness in collaborative virtual environments. *Proceedings of ACM SIGCHI Conference on Virtual Reality and Technology* (VRST '94), 1994.
[7] C. Clarson and O. Hagsand. DIVE - a platform for multi-user virtual environments. *Computers & Graphics*, 17(6):663-669, 1993.
[8] T.A. Funkhouser. RING: A client-server system for multi-user virtual environments. *Proceedings of the 1995 ACM Symposium on Interactive 3D Graphics*, pages 85-92, 1995.
[9] M. Capps, S. Teller. Communication visibility in shared virtual worlds. *Proceedings of the 6th Workshop on Enabling Technologies: Infrastructure for Collaborative Environments*. Cambridge, MA, June 1997.
[10] M.R. Macedonia, D.P. Brutzman, M.J. Zyda, D.R. Pratt, P.T. Barham, J.Falby and J. Locke. NPSNET: A multi-player 3D virtual environment over the Internet. *Proceedings of the 1995 ACM Symposium on Interactive 3D Graphics*, pages 93-94, 1995.
[11] J. W. Barrus, R.C. Waters and D.B. Anderson, Locals and Beacons: Efficient and precise support for large multi-user virtual environments. *Proceedings of IEEE Virtual Reality Annual Intl. Symp. (VRAIS). 1996.*
[12] O. Sudarsky and C. Gotsman. Output-sensitive visibility algorithms for dynamic scenes with application to virtual reality. *Proceedings of Eurographics '96,* 1996. Blackwell. *Computer Graphics Forum*, 15(3).