Frontier Sets in Large Terrains

Shachar Avni James Stewart

School of Computing
Queen's University
Kingston, Canada

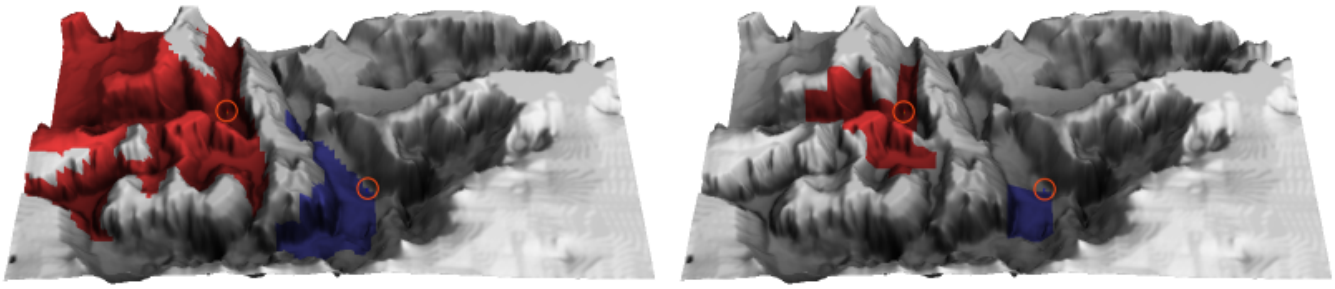

Figure 1: Frontier sets generated by our algorithm for two viewpoints (circled) on a 100,000 point Martian terrain for differing levels of PVS compression. *Left:* Hierarchical PVS compressed to 1.0 GB. Frontier sets computed in 78 milliseconds. Frontier sets total 27642 points. *Right:* Hierarchical PVS compressed to 0.1 GB. Frontier sets computed in less than one millisecond. Frontier sets total 6912 points.

Abstract

In current online games, player positions are synchronized by means of continual broadcasts through the server. This solution is expensive, forcing any server to limit its number of clients. With a hybrid networking architecture, player synchronization can be distributed to the clients, bypassing the server bottleneck and decreasing latency as a result. Synchronization in a decentralized fashion is difficult as each player must communicate with every other player. The communication requirements can be reduced by computing and exploiting frontier sets: For a pair of players in an online game, their frontier sets consist of the region of the game space in which each player may move without seeing (and without communicating to) the other player. This paper describes the first fast and space-efficient method of computing frontier sets in large terrains.

Keywords: frontier sets, decentralized online games, interest management, visibility, large terrain environments

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms; I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1 Introduction

Decentralized network architectures for online games have recently gained much interest in the research community. This trend is due to their significant promise of robustness (with the lack of a single point of failure), reduced latency, and greater scalability when compared to the traditional client-server model. However, the presence of yet unsolved design issues, including the lack of a central point of synchronization, an increased threat of cheating, and no persistence of the virtual world, has limited commercial online games to a centralized model.

In current games, player synchronization is accomplished by clients repeatedly sending position information to the server. The server then broadcasts the positions to the relevant players. This is unattractive as it adds latency between player renderings on the client's computer. This also becomes prohibitively expensive as the number of players becomes large, effectively capping the possible number of clients for any server.

Hybrid network architectures [15, 23] have recently been proposed which combine peer-to-peer and client-server methodologies. In such systems, the server maintains important game state while clients communicate directly in peer-to-peer fashion. The challenges mentioned above become moot as the server remains the central authority. Moreover, hybrid architectures have been shown to decrease the bandwidth required to host a game [15]. Decentralized online games may one day be viable commercially through the use of hybrid network architectures.

With a hybrid architecture, player synchronization can be distributed to the clients, freeing the server's resources and lowering latency in the process [23]. Synchronization done in this manner is difficult as each player depends on all other players to maintain position information. Hence, if there are k players, k^2 packets are sent at each time unit in the worst case. Frontier sets, introduced by Steed and Angus [21, 22] are an efficient network partitioning scheme that can be used to mitigate this problem. Frontier sets are based on the idea of the Potentially Visible Set (PVS) [2, 25].

With the frontier set method, each pair of players constructs between them an agreed-upon set of mutually invisible regions called frontier sets. As long as each player remains within his own region, no communication between the two players is necessary. These regions are in fact "update-free regions" [18], or areas where communication between pairs of players is not needed. Frontier sets are typically valid for several seconds, during which time the cost of synchronization is reduced to a local test to determine whether the player has left his respective region. If one player leaves his region, the other player is notified and the players once again generate a pair of frontier sets, if possible.

Frontier sets have been shown to be a viable solution in environments that can be partitioned into portals and cells [22, 23]. However, their use has not yet been shown to be practical in large open environments such as terrains. A terrain is commonly stored as a heightmap, consisting of the heights of discretized sample points over its surface. While a typical game level may consist of thousands of cells, the same size level as a terrain may require hundreds

of thousands of sample points.

The contribution of this paper is a fast algorithm for conservative frontier set generation in large terrain environments. The algorithm uses a quadtree-based data structure for the hierarchical storage of a terrain PVS along with a PVS merging operation and a PVS compression method. We show that our algorithm produces large frontier sets at interactive rates.

2 Background and Related Work

The general research problem we are addressing in this paper is that of *interest management* in networked virtual environments (NVEs). The goal of interest management is to predict which nodes in a network (or partition of a network) need to communicate, so as to avoid relaying useless information. The subject first received major attention in the research community in the late 1980s when NVEs were required for real-time combat simulation [19]. Interest management remained relevant through the 1990s as Multi-User Dungeons [4] were first graphically implemented and recently as Massively Multiplayer Online RPGs (MMORPGs) such as World of Warcraft have been gaining in popularity. In this section, we discuss interest management within the scope of networked online games.

The first solutions to interest management evolved for use in centralized (client-server based) online games. Approaches to interest management such as RING [12] and BrickNet [20] involve filtering requests from clients through the server as the requests occur. Unfortunately, since the server must deal with all requests, the filtering quickly becomes a major bottleneck. One popular solution is the static partitioning of the virtual world into regions and the restriction of a client's interaction only to other clients within their respective region. These regions would each be managed by a server or group of servers. The main drawback with this approach is that the static nature of the regions causes undue load on their respective server(s) when users cluster together in the same region (such as for guild meetings in MMORPGs). This problem necessitates the runtime allocation of users to more available servers or *dynamic load balancing* [7]. Dynamic partitioning approaches [9] have also been studied.

Apparent scalability and design issues in centralized online games have resulted in an increased focus on decentralized architectures. These architectures are, in turn, hampered by their own flavour of design issues [11], of which interest management is a prominent concern. One class of research involves the partitioning of the world into regions, wherein peers use a publish-subscribe architecture and communicate through an ad-hoc multicast group [1, 3, 17]. Another popular approach is the partitioning of the network using a Voronoi diagram [13].

The interest management scheme specifically related to our work is that of update-free regions. The computation of frontier sets [21, 22, 23] is one effective method of maintaining update-free regions. This scheme differs from the above schemes in that the peers negotiate the regions amongst themselves, as opposed to relying on a pre-determined partitioning of the world.

In this paper, we expand on the frontier set concept introduced by Steed and Angus. Those authors initially proposed to pre-compute frontier sets and to read them into memory for pairs of players at run-time [21]. Next, a more refined solution was introduced where an *Enhanced PVS* is pre-computed [22]. The frontier sets are generated at run-time by checking a distance criterion of each visibility cell with respect to both players and adding the visibility cells to the appropriate player's frontier set.

Our solution differs from that of Steed and Angus in that it is more applicable to terrains, which can contain a very large number of points. The method of Steed and Angus would, we expect, incur a high computational cost because it would iterate through a large number of points to build the frontier sets.

We focus our study on frontier set generation for pairs of players. Past studies have focused only on pairs of players as well [23]. However, judging by the research on network traffic in a fast-paced online game [21], simply maintaining a list of frontier sets should not pose an issue for groups of up to 32 players [23]. Theoretically, frontier sets in their current form are useful for online games such as Quake [14] where groups of players move around frenetically in a first-person shooter deathmatch. This research facilitates the inclusion of terrain environments and decentralized network architectures to this genre of games. Network trials, optimization, and validation for groups of players is the focus of future work.

3 Frontier Sets in Terrains

The problem we address is this: Given two player positions in a terrain of n points, compute "mutually invisible regions" around each position such that, for all pairs of positions, p from one region and q from the other region, players at p and q cannot see each other. The regions should be as large and contiguous as possible, since they will have to be recomputed when one of the players leaves his region.

The terrain is represented as a heightmap in a regular square grid and is typically stored as a rectangular texture. Since terrains can be quite large, a naïve region growing approach, which builds the regions by iterating through all the individual points on the terrain, will likely result in impractical runtimes. Thus, we propose to use a quadtree decomposition of the terrain and to grow the regions by iteratively adding adjacent quads to each region. This potentially allows the removal of many points from consideration in a single iteration.

A hierarchical PVS is built on the terrain. First, the PVS at each terrain point is computed. The point PVSs are then aggregated in a quadtree over the terrain, resulting in a *hierarchical PVS* where each quadtree node stores the conservative PVS of its corresponding terrain region.

3.1 PVS Computation, Storage, and Manipulation

The point PVSs can be computed off-line via raycasting or by a number of methods [5, 6, 10, 16]. However, the method may have to be adjusted to work on terrains and the viewpoints would have to be set to the player's height above the terrain. The choice of method for PVS computation is inconsequential to the performance of the algorithm. The method we chose for our implementation is discussed in the next section.

3.1.1 PVS Computation

The horizon is the boundary between the sky and the terrain surface when viewed from a particular point, x, in the $[0, 2\pi]$ range of azimuth directions around x. The key observation is that anything beyond and below the horizon is invisible from x. Since the complexity of the horizon in a meshed terrain of n points is $\mathcal{O}(n\,\alpha(n))$, we discretize the set of azimuth directions into a finite number of equal radial *sectors*, with the accuracy of the resulting horizon increasing with the number of sectors used.

Horizon points on their own do not describe all the necessary visibility information; namely, any occluded regions between the viewpoint and the horizon must still be determined (see Figure 2). The occluded regions of a sector with vertex x can be determined by inspecting the points along the midline of the sector and picking those points, p, that are (a) local maxima with respect to the elevation line from x to p and (b) are of a higher elevation (with respect to x) than all maxima that are closer to x. The resulting list consists of *intermediate horizon points*; an area of the terrain is invisible if it is beyond an intermediate horizon point, p, and below the corresponding elevation line passing through x and p, as shown in Figure 2. The areas of terrain beyond and below a horizon point

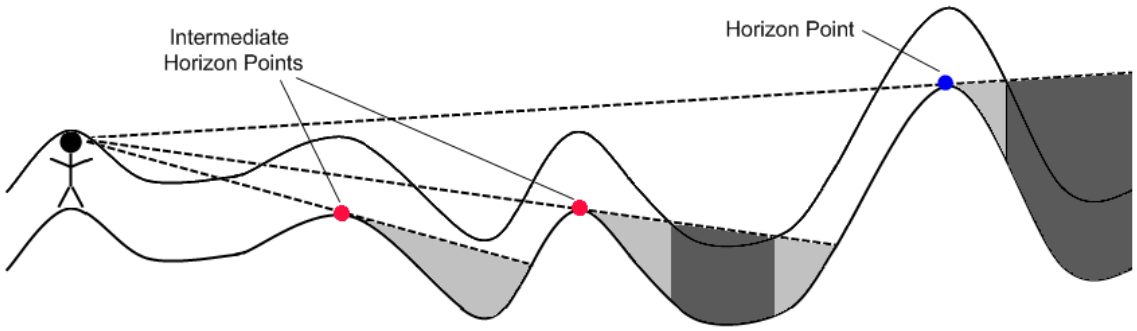

Figure 2: A cross-section of the terrain down the middle of a sector. The curve at the stick figure's feet represents the terrain; the offset curve above shows the position of the top of the player's head. The light gray regions are those that are hidden to the viewpoint but in which a player of the same height is still visible. The dark gray areas are those in which a player of the same height is hidden from the viewpoint.

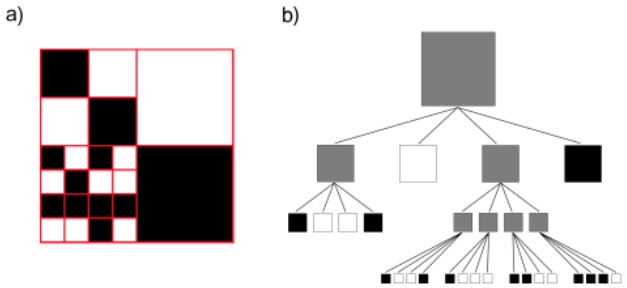

Figure 3: a) a PVS as a binary image. The white regions correspond to visible areas and the black regions correspond to hidden areas. b) resulting compressed PVS as a quadtree. The gray nodes correspond to regions having unknown visibility; further traversal is necessary to determine the visibility information for the sub-regions.

or intermediate horizon point are known as the "dead zones" [8] of the viewpoint.

However, the dead zones described by all horizon points may not be practical. For instance, if a player is two meters tall but the dead zone is only one meter deep at its deepest point, the player will still be visible when standing in the dead zone. To remedy this, set a threshold, τ, to be the maximum height of a player and include only the parts of each dead zone that are deeper than τ.

For each sector of a terrain point, x, we first compute the horizon p [24], then determine the intermediate horizons and dead zones along a Bresenham line passing between x and p. For a sufficient number of sectors (e.g. 1,000), this brute-force sampling is dense and any errors introduced are negligible. The PVS of x consists of the complement of the union of the dead zones of the sectors around x.

3.1.2 PVS Storage

The PVSs computed thus far can be stored in n bitmaps with the same dimensions as the terrain's heightmap. This is unattractive because, during the growing of a frontier set region around a player, the PVS of an adjacent point or quad must be merged with the PVS of the growing region. Such merging takes time proportional to the number of terrain points and can be too costly for large terrains. We thus describe a method to compress the PVS, which reduces storage costs and accelerates the merging of PVSs.

We treat the PVS as a binary image and store it as a compressed quadtree based on the predictive coding method of Wilson [26]: Each node in the quadtree is marked as "visible", "hidden", or "unknown", corresponding to its visibility status with respect to the viewpoint region of the PVS. The leaf nodes (corresponding to points in the heightmap) are initially marked as "visible" or "hidden" according to the results from the occlusion region computation of Section 3.1.1. From the finest level to the coarsest level, do the following at each quad: If the quad's four children are all "visible" or all "hidden", mark the quad with the same label and delete the children. Otherwise, label the quad as "unknown". For a typical PVS, the all-visible and all-hidden regions are fairly homogeneous, leading to large savings after compression. An example is shown in Figure 3.

3.1.3 PVS Merging

A key operation of the algorithm is to merge two terrain regions, each of which has its own PVS. The PVS of the merged region is computed by merging the PVSs of the two regions.

For two hierarchical PVSs, the merged PVS, M, is computed with a simultaneous depth-first traversal of both trees, comparing pairs nodes at corresponding positions in the two trees. The following cases arise when comparing quadtree node A from one PVS with quadtree node B from the other (both nodes correspond to the same region on the terrain):

- **Case 1 (both "unknown")**: Add a "unknown" node to M and recurse below that node.
- **Case 2 (both "hidden")**: Add a "hidden" node to M and don't recurse farther.
- **Case 3 (at least one "visible")**: Add a "visible" node to M and don't recurse farther. This is a conservative choice.
- **Case 4 (one "unknown" and the other "hidden")**: Add the subtree rooted at the "unknown" node to M and don't recurse farther.

A lossy merge is possible if, in Case 4, the visibility status of the node of M is set to "visible". This reduces the size of the merged tree at the expense of the visibility information stored in the "unknown" subtree.

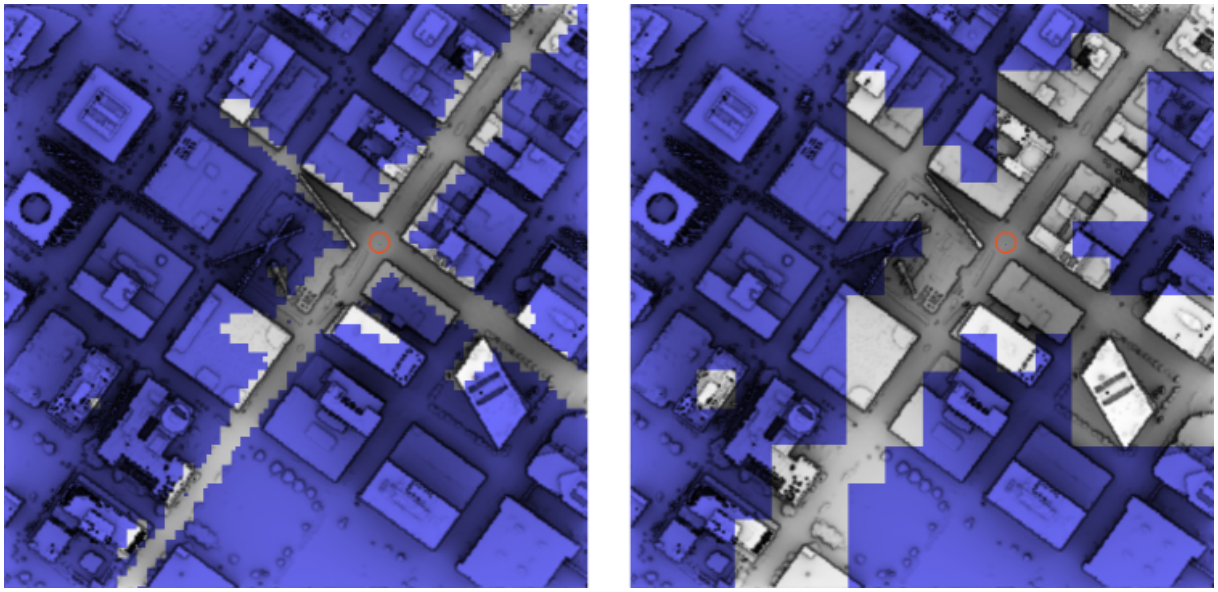

Figure 4: Invisible regions (shown in blue) for a particular point (circled) on the surface of downtown Houston. *Left:* Hierarchical PVS compressed to 1.0 GB. The PVS contains 216544 occluded points. *Right:* Hierarchical PVS compressed to 0.1 GB. The PVS contains 165228 occluded points.

3.1.4 PVS Compression

The hierarchical PVS stores an individual PVS in each of its nodes; each individual PVS is, itself, stored as a quadtree (Section 3.1.2). For large terrains, the hierarchical PVS is likely too large to fit into main memory. A typical 32-bit operating system only allows each process a maximum of 2 GB of virtual memory, of which approximately 1.5 GB may be addressed comfortably. Although this limit may be raised, and although 64-bit operating systems have been becoming more popular, it is not scalable to simply read in as much visibility information as possible. Moreover, the I/O involved with such memory sizes may take minutes or hours, far too long for a game player to wait. Hence, we present a scheme to compress the individual PVSs that are stored in the quads of the hierarchical PVS.

Choose an upper bound, N, on the number of nodes in an individual PVS, computed from an upper bound on the desired memory footprint. First, compute and store an individual PVS (Sections 3.1.1 and 3.1.2). Then, if the number of nodes in the PVS is larger than N, perform the following compression: Find a leaf node, L, of maximum depth. (Note that at least one of L's three siblings has a different visibility status than that of L; otherwise, L and its siblings would already be merged into the parent.) Set the value of L's parent node to "visible" and remove L and its siblings. If all of L's parent's siblings are now "visible", the visibility status of L's grandfather changes from "unknown" to "visible". In this case, propagate the change up the tree from L's grandfather, removing any subtrees rooted at newly-"visible" nodes. Repeat the compression with the next node of maximal depth until the size of the quadtree falls below N. This compression method is conservative.

We have tested this method with memory size limits of 0.1 GB, 0.25 GB, 0.5 GB, and 1 GB on two terrains. The first terrain is a 100,000–point sample of the surface of Mars; the second is a 250,000–point sample of downtown Houston. Our results show that most visibility information remains despite the drastic cut in total memory usage (see Figure 4 for an example of the occluded regions for a given point on the surface of downtown Houston). The uncompressed version of the hierarchical PVS of the Houston terrain takes up more than 17 GB of space. After compressing the PVS to 1.0 GB, the PVS for the point in Figure 4 contains 216544 occluded points, or 87% of the terrain. After further compression to 0.1 GB, the PVS has 165228 occluded points. That is a loss of only 24% of the visibility information for a 90% reduction in the size of the PVS.

In summary, the pre-computed hierarchical PVS provides a conservative PVS at each node of the terrain quadtree. The merging operation allows the PVS of an adjacent quad to be added to the PVS of a growing region. The PVS compression allows the hierarchical PVS to be loaded into main memory and the reduced PVS size keeps the merging operations to within reasonable execution times.

3.2 Frontier Set Computation

The frontier set computation, which uses the PVS storage, compression, and merging described above, is in fact quite simple and consists of the following three steps. Examples of frontier sets generated by our algorithm can be seen in Figures 1 and 5.

Step 1: Determine Maximal Mutually Invisible Quads

Let A and B be two players standing at points p_A and p_B on the terrain. First, determine if A and B are mutually invisible by querying p_B's PVS to determine whether p_A is within an invisible region. If the players are mutually visible, no frontier set can be created.

If the players are mutually hidden, begin a simultaneous traversal up the levels in the hierarchical PVS, starting at the parent quads of p_A and p_B and stopping when the current quads are not mutually invisible. An invisibility query between quads Q_1 and Q_2 is done by finding the PVS of Q_1 and determining whether the quad in that PVS corresponding to Q_2 is marked as "invisible". The last quads reached in this way are the maximal mutually invisible quads around the players' current positions, and are the initial quads from which to grow the frontier sets. Call these quads Q_A and Q_B.

If the last quads reached in this manner are the leaf quads corresponing to the starting points themselves, then a non-region-growing approach must be applied as region growing would be a

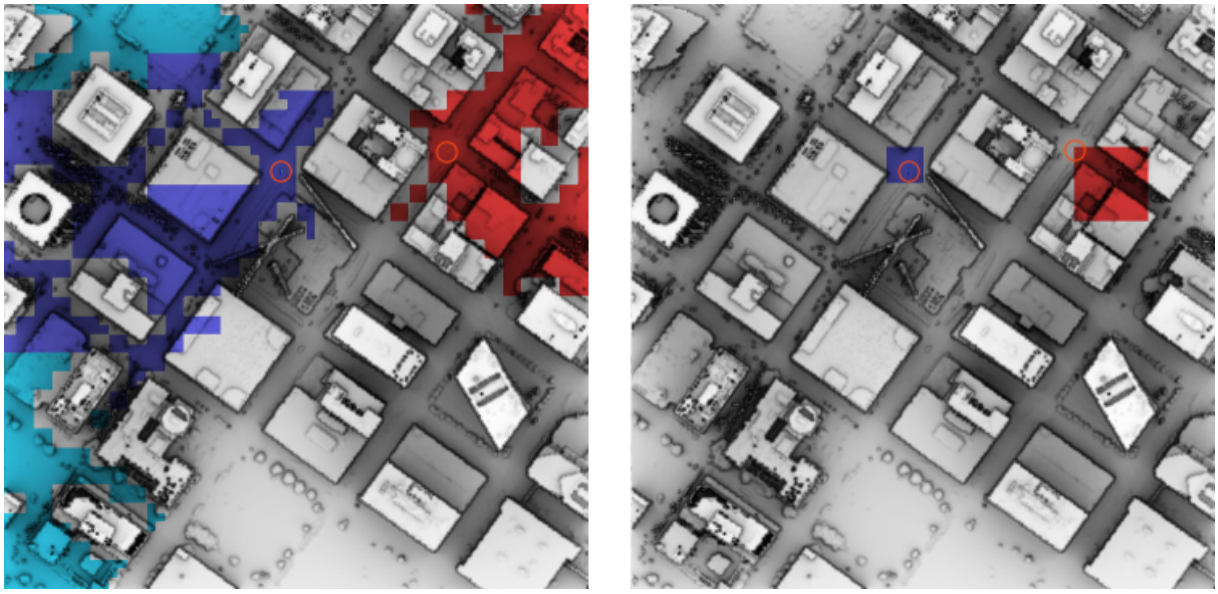

Figure 5: Frontier sets generated by our algorithm for two viewpoints (circled) on a 250,000–point sampling of downtown Houston for differing levels of PVS compression. This is a particularly challenging case as the larger quads contain viewpoints on the tops of buildings. On the left, the dark blue area shows the frontier set at the termination of step 2 in the algorithm and the light blue area shows the quads added in step 3. *Left:* Hierarchical PVS compressed to 1.0 GB. *With Step 3:* Frontier sets computed in 93 milliseconds. Frontier sets total 76712 points. *Without Step 3:* Frontier sets computed in 47 milliseconds. Frontier sets total 55332 points. *Right:* Hierarchical PVS compressed to 0.1 GB. Frontier sets computed in less than one millisecond. Frontier sets total 5120 points.

costly walk along the individual leaf nodes of the quadtrees. In this case, we simply set the frontier sets to the starting points. We also argue that frontier sets are not very useful for such a pair of points since the players are one point away from leaving their respective frontier set, resulting in a likely recomputation after a single time unit.

Step 2: Grow Regions

Add Q_A to A's frontier set and initialize the PVS of the frontier set of A to the PVS of Q_A. Do the same for B. Next, determine the adjacent quads of both Q_A and Q_B (with either a 4-neighbour or 8-neighbour approach) and place them on a FIFO queue for each player. Alternating between the two queues, extract the next quad from the queue of one player and query its visibility status in the PVS of the opposing player. If the visibility status is "unknown" and if this is not a lowest level quad, place this quad's children *that are adjacent to this player's frontier set* into the queue. If the visibility status is "invisible", add this quad to the frontier set, merge the PVS of the frontier set with the PVS of this quad, and place this quad's non-visited neighbours on the queue. If the visibility status is "visible", do nothing. Stop when one of the queues becomes empty.

This step may be customized depending on the terrain and the needs of the application. For instance, in a city environment, it is unreasonable to expect a player to go from a street corner to the top of a building in a single step. For such a situation, we propose to store the quads not in a FIFO queue, but in a priority queue ordered by distance from the ground. This would cause the regions to grow along the streets of the city before reaching higher ground.

Step 3: Grow Larger Frontier Set (optional)

At the end of Step 2, there is likely to be one frontier set which can no longer grow while the other frontier still has quads on its queue. Since larger frontier sets usually lead to fewer packets sent across the network, it may be beneficial to grow the frontier set with the non-empty queue. However, if the frontier set that can no longer grow is relatively small, the larger frontier set may grow very large as the smaller frontier set's PVS sees only a small portion of the terrain. This is unattractive as the time used growing this frontier set would be wasted if the other player leaves their region relatively quickly. Thus, deciding whether to grow at this stage is left up to the application. However, if it is beneficial for the larger frontier set to grow, the merging of PVSs with added quads is not necessary as the other player's frontier set can no longer grow and thus does not require the larger frontier set's visibility information.

The blue frontier set on the left in Figure 5 shows the difference between including and excluding step 3. The dark blue area shows the frontier set at the end of step 2 and the light blue area shows the quads added in step 3. In this particular case, the algorithm reached the end of step 2 in 47 milliseconds with a combined frontier set size of 55332 points. After step 3, the frontier set construction took a total of 93 milliseconds and the combined frontier set size reached 76712 points. Thus, in this case, it would have been wiser to stop after step 2 as step 3 contributed only 21380 points in 46 milliseconds.

Also notice that the frontier sets are roughly equal in size at the end of step 2. This is an attractive quality of a frontier set pair because, with unbalanced sizes, the player in the smaller frontier set would likely leave the region more quickly. This would waste all the time spent on building the larger frontier set.

4 Experimental Results

We tested the algorithm on the two terrains (Mars and Houston) at four different memory limits (0.1, 0.25, 0.5, and 1 GB). For each combination, we randomly and independently generated 10,000 mutually invisible pairs of points. The algorithm was executed (skipping the optional step 3) on all pairs.

Table 1 shows the distribution of frontier set computation times for the 0.1 GB and 1.0 GB memory limits. The distributions of

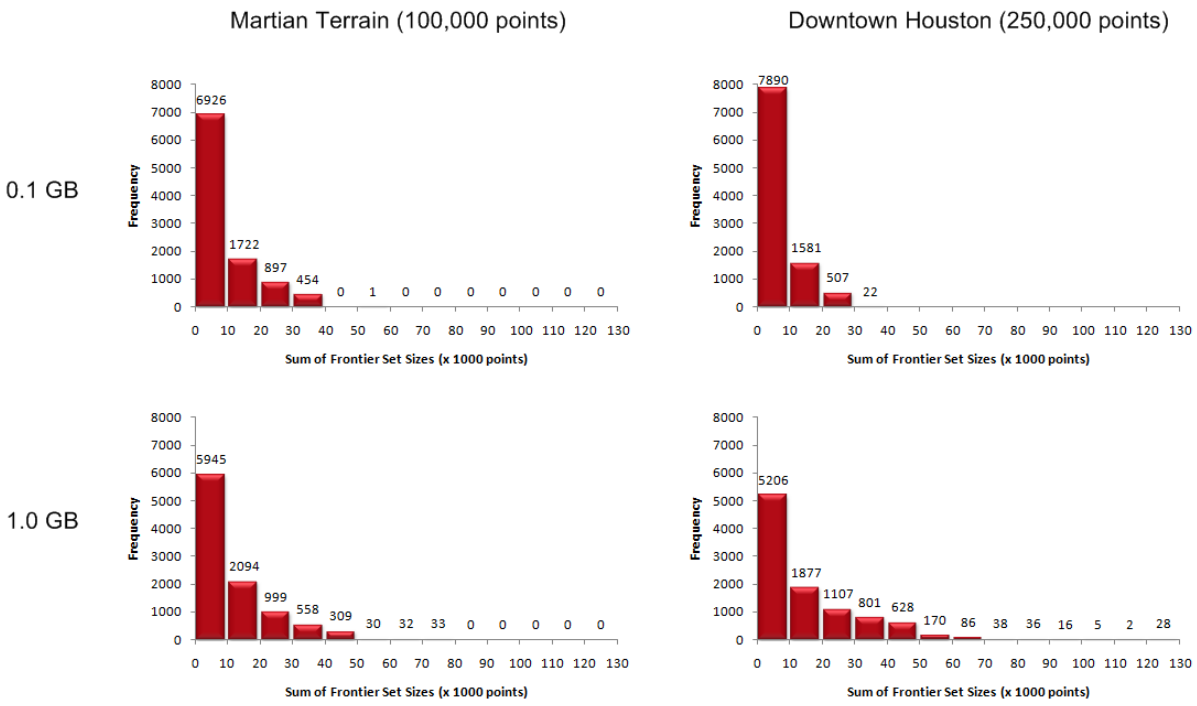

Figure 6: Distributions of frontier set sizes for a random sampling of 10,000 pairs of mutually invisible points.

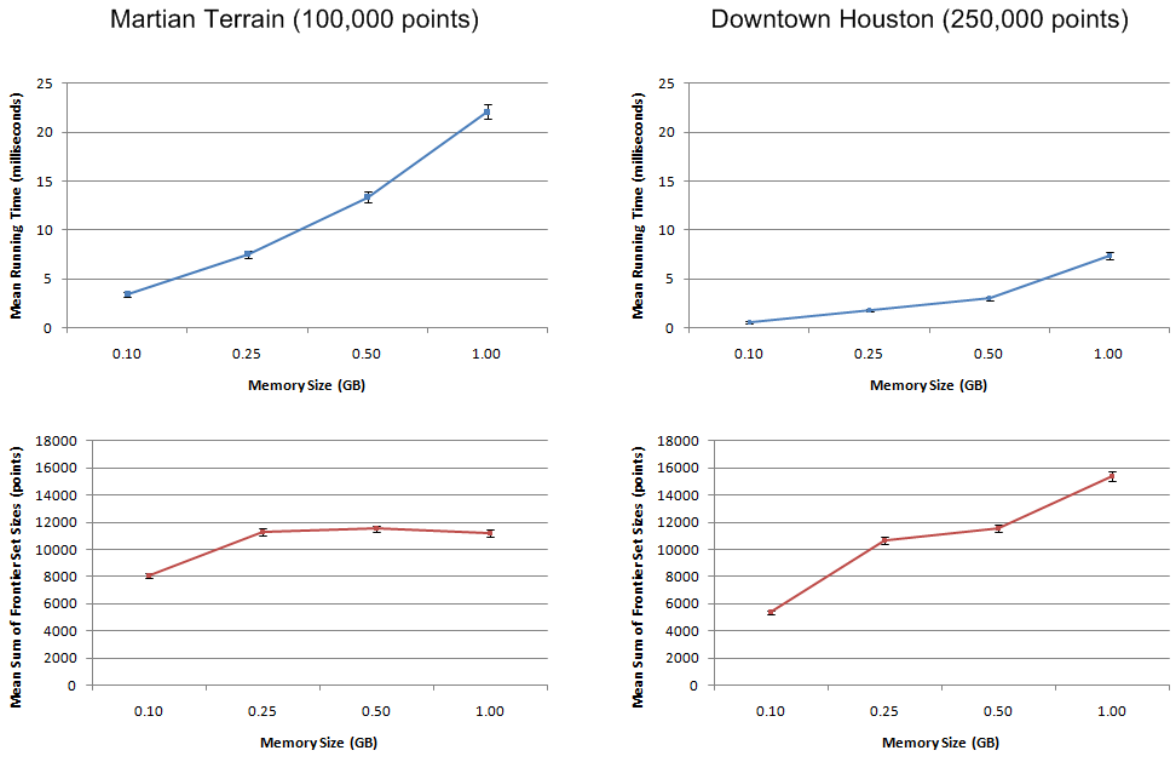

Figure 7: Mean running time and mean frontier set size for differing PVS compression sizes. Means were computed from a random sampling of 10,000 pairs of mutually invisible points. Error bars show the 95% confidence interval of the mean.

Table 1: Distribution of frontier set computation times for 10,000 pairs of mutually invisible points over both Mars and Houston terrains with memory limits of 0.1 GB and 1.0 GB.

Running Time (milliseconds)	Martian Terrain PVS Size		Downtown Houston PVS Size	
	0.1 GB Limit	1.0 GB Limit	0.1 GB Limit	1.0 GB Limit
0 - 8	8655	3942	9694	6972
9 - 24	1036	3027	279	2313
25 - 40	168	1639	16	405
41 - 56	44	521	6	138
57 - 104	65	559	4	129
over 104	32	312	1	43

the frontier set sizes are shown in Figure 6 for the 0.1 GB and 1.0 GB memory limits. Figure 7 shows the overall trends for the mean running times and the mean frontier set sizes at different memory limits. All running times were recorded on a system with 3 GB of RAM and four Intel Core 2 Quad CPUs each with a speed of 2.40 GHz (but only one core of one of the CPUs was used, as no parallel implementation was made).

The mean running times shown in Figure 7 suggest that the running time grows slightly superlinearly with the memory limit. The fastest running times occurred with the smallest memory limits for both the Martian terrain and downtown Houston. An important property to note is that the running time is proportional to the size of the PVS quadtrees involved in the PVS merging. A larger terrain has smaller PVS quadtrees than a smaller terrain if they both use the same compression size. This accounts for the fact that running times were faster for Houston even though Mars is a smaller terrain.

For the Martian terrain, the mean running time at the smallest memory limit was 3.4 ms and the standard deviation was 14.1 ms. For downtown Houston, the mean running time at the smallest memory limit was 0.6 ms and the standard deviation was 3.7 ms.

The mean frontier set size grows at a linear or sublinear rate with the memory limit. In the Martian terrain, increasing the memory limit beyond 0.25 GB didn't result in a substantial increase in the size of the frontier sets, so it would be best to stop at 0.25 GB.

For the Martian terrain, the mean frontier set size at 0.1 GB was 8076 points with a standard deviation of 9606 points. For downtown Houston the mean frontier set size at 0.1 GB was 5377 points with a standard deviation of 6867 points.

These results are promising as they show that the size of the hierarchical PVS can be efficiently controlled, resulting in interactive running times and large frontier sets.

5 Conclusion

With the advent of hybrid network architectures, decentralized online games may soon become commercially viable. We have presented a fast and space-efficient method of computing frontier sets in large terrains. These frontier sets can be used to reduce the burden of player synchronization in decentralized online games.

Our experimental results show (a) that the memory footprint of the hierarchical PVS can be efficiently controlled without a large reduction in PVS quality, and (b) that the frontier sets of two players can be computed at interactive rates in large terrains.

The focus of future work will be on validating the current frontier set solution for groups of players. Previous studies suggest simply maintaining a list of frontier sets is sufficient for groups of up to 32 players [23]. This claim will be validated through network trials.

Also of interest is to extend the notion of a frontier set to beyond pairs of players: Each player in the game should efficiently compute an overlapping group of frontier sets with respect to all other players, and update this group efficiently as the game proceeds. Other players can themselves be grouped spatially to accelerate the computation of the grouped frontier sets.

References

[1] H. Abrams, K. Watsen, and M. Zyda. Three-tiered interest management for large-scale virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 125–129, 1998.

[2] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, The University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, July 1990.

[3] J. W. Barrus, R. C. Waters, and D. B. Anderson. Locales and beacons: Efficient and precise support for large multi-user virtual environments. In *Proceedings of IEEE VRAIS*, pages 204–213, 1996.

[4] R. Bartle. Interactive multi-user computer games, December 1990. Unpublished research report, MUSE Ltd, British Telecom plc.

[5] J. Bittner, O. Mattausch, P. Wonka, V. Havran, and M. Wimmer. Adaptive global visibility sampling. *ACM Transactions on Graphics*, 28(3):1–10, 2009.

[6] J. Bittner, P. Wonka, and M. Wimmer. Visibility preprocessing for urban scenes using line space subdivision. In *PG '01: Proceedings of Pacific Graphics*, pages 276–284. IEEE, 2001.

[7] J. Chen. Locality aware dynamic load management for massively multiplayer games. Master's thesis, University of Toronto, Canada, January 2005.

[8] D. Cohen-or and A. Shaked. Visibility and dead-zones in digital terrain maps. *Computer Graphics Forum*, 14:171–180, 1995.

[9] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, New York, NY, USA, 2005. ACM.

[10] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. In K. Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 239–248. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[11] L. Fan, P. Trinder, and H. Taylor. Design issues for peer-to-peer massively multiplayer online games. In *MMVE '09: Proceedings of the 2nd IEEE International Workshop on Massively Multiuser Virtual Environments*, pages 1–11, March 2009.

[12] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–92, 209, 1995.

[13] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *CCNC '08: Proceedings of the 5th IEEE 2008 Consumer Communications and Networking Conference*, pages 1134–1138, 2008.

[14] id Software. Quake III Arena, 2000. http://www.idsoftware.com/games/quake/quake3-arena/.

[15] J. L. Jardine. The hybrid game architecture: Distributing bandwidth for mmorpgs while maintaining central control. Master's thesis, Brigham Young University, Provo, Utah, USA, December 2008.

[16] S. Laine. A general algorithm for output-sensitive visibility preprocessing. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 31–40, New York, NY, USA, 2005. ACM.

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

[18] Y. Makbily, C. Gotsman, and R. Bar-Yehuda. Geometric algorithms for message filtering in decentralized virtual environments. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 39–46, New York, NY, USA, 1999. ACM.

[19] A. Pope. The simnet network and protocols. Technical Report 7102, BBN Systems and Technologies, July 1989.

[20] G. Singh, L. Serra, W. Png, and N. Hern. Bricknet: A software toolkit for network-based virtual worlds. *Presence*, 3(1):19–34, 1994.

[21] A. Steed and C. Angus. Frontier sets: A partitioning scheme to enable scalable virtual environments. In M. Alexa and E. Galin, editors, *Eurographics 2004*, pages 13–17, 2004.

[22] A. Steed and C. Angus. Supporting scalable peer to peer virtual environments using frontier sets. In *VR '05: Proceedings of the 2005 IEEE Conference on Virtual Reality*, pages 27–34, Washington, DC, USA, 2005. IEEE Computer Society.

[23] A. Steed and B. Zhu. An implementation of a first-person game on a hybrid network. In *MMVE '08: Proceedings of the 1st IEEE International Workshop on Massively Multiuser Virtual Environments*, pages 24–28, March 2008.

[24] A. J. Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, 1998.

[25] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[26] R. Wilson. Quad-Tree Predictive Coding: A New Class of Image Data Compression Algorithms. Report LiTH-ISY-I-0609, Computer Vision Laboratory, Linköping University, Sweden, 1983.