Realistic Fireteam Movement in Urban Environments

Christian J. Darken, Daniel McCue, and Michael Guerrero

MOVES Institute, Naval Postgraduate School, Monterey, California, USA
{cjdarken, djmccue, mjguerre} @ nps.edu

Abstract

Realistic simulations of the movement of infantry in urban environments with 3D models and at interactive rates is of wide and enduring interest. Many video games have attempted it, and military simulations are increasingly doing the same. Previous attempts have fallen short in two areas: properly coordinating movement, and adequate modeling of the detection of hostile targets. Novel algorithms to simulate fireteam movement and visual scanning appropriate to urban environments are described. Measurements of the computational performance of the most expensive components of the approach are provided.

Introduction

Because of their complex geometry, urban environments are some of the most dangerous for dismounted infantry. The four men of an infantry fireteam appear to an outsider to move chaotically. This is intentional; predictable movement makes a sniper's job easy. In actuality, they are moving according to a list of fairly well defined priorities: taking advantage of cover afforded by walls and other objects, staying far enough from teammates to avoid simultaneous destruction by a grenade or bomb yet close enough for mutual support and to receive commands from the team leader. All the while, each fireteam member is scanning the surroundings for any potential threat.

Simulating the movement of infantry in urban environments with 3D models and at interactive rates is of wide and enduring interest. Many video games have attempted it, and military simulations are increasingly doing the same. Previous attempts have fallen short in two areas: properly coordinating movement, and adequate modeling of the detection of hostile targets. Both of these areas are critical for many military training and analysis applications. Furthermore, a solution to these problems would afford designers of infantry-themed video games the choice of greatly increased realism.

From a technical point of view, three main obstacles must be addressed. Realistic infantry modeling requires an unprecedented amount of visibility testing. Line of sight tests are not adequate to support state-of-the-art target detection algorithms, which require knowledge of both how much of the target's surface is visible and how much target to background contrast is present. Visibility checks are not only necessary for target detection. They are also needed for movement modeling. Visibility checks are needed to evaluate positions for safety, the ability to see targets in areas of interest, the ability to see other members of the fireteam, and more. Because of the large number of required visibility checks and the need to achieve interactive rates, we have developed an off-line visibility precomputation and caching scheme.

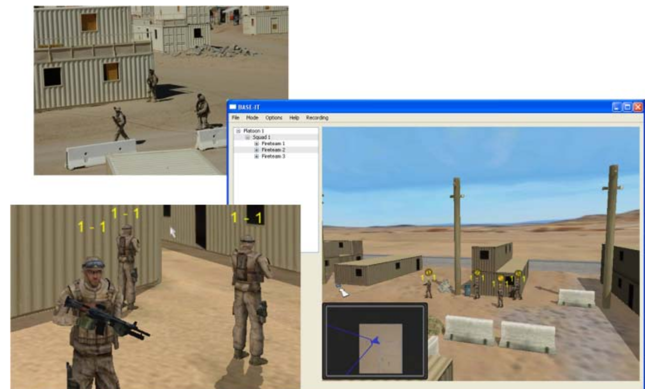

Figure 1: *Montage containing live Marines on patrol during training (upper left) and simulated Marines in our prototype (right and detail lower left).*

Secondly, an algorithm for determining where a fireteam member looks at any given time must be developed. A fireteam does not even begin to detect a target before some team member looks in its vicinity.

Third, a realistic model of how the individual infantryman's movement coordinates with that of the rest of the fireteam is needed. We have developed a hierarchical search model of fireteam movement. An approximate path for the fireteam as a whole is generated by a cover-weighted A* search. Then, alternating greedy searches incorporating several additional factors are periodically performed for each member of the fireteam to determine their movement in detail.

We have developed a demonstrable advanced prototype that meets these requirements. Below, we describe the

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

previous work on which our approach is based. We then describe our approach to each of the above issues in detail.

Related Work

Most video games and real-time military simulations use a line-of-sight algorithm to determine whether a target is detected or not. To save computation at run-time, lines-of-sight may be precomputed for all pairs of navigation graph nodes (Liden 2002). The state of the art in off-line military simulations is to use the ACQUIRE algorithm, which gives a probability of detection (and detection time, in the case of positive detection) based on both the exposed solid angle of the target and its contrast to its background. We base our work on a previously existing adaptation of ACQUIRE to detailed 3D environments (Darken and Jones 2007). These are the values that we precompute and cache.
The model of potential threats in the environment that we use is an occupancy map-based (Darken and Anderegg 2008) variant of that described in Alt and Darken (2008), a line of research that originated with Isla and Blumberg (2002). As in Alt and Darken, a critical difference from standard occupancy maps is that visible waypoints do not generally have a zero possibility of being occupied by a threat, since we are using a probabilistic model of target detection, i.e. just because someone looks at a location and does not see a threat does not provide certainty that no threat is there.

Path planning in video games and military simulations has a very large literature. Reece (2003) is very close to our paper in spirit. Not only does he specifically address fireteam movement, and model threat as a movement cost, but he uses a two tiered architecture for determining movement. However, the second tier he uses is steering. Steering, as Reece points out, is subject to failure sometimes, especially in cluttered environments, as city streets often are. As Jurney (2008), Reece treats only the movement of units moving in fixed formations, which is not how infantry move in urban environments.

Straatman et. al. (2005) also deserves mention for showing the extent to which various types of consideration may be modeled as a movement cost, and also for highlighting the selection of end-points of movement as an important sub-problem of movement. Like Straatman et. al., we use scoring to select the best end-points, though our scoring criteria are different.

Visibility Precomputation

One simple method to determine the intervisibility between two points in a 3D space is to cast a ray from the first point toward the second point. If no collisions occur between the two then both points are deemed as visible from the other. The result is Boolean since the object will either be visible or not. This method is often deceived (Darken and Jones 2007).

Instead, we use a method that depends upon the exposed surface of the target (subjective to the viewer, i.e. equivalent to the exposed solid angle of the target) as well as its contrast with its background. We call the exposed surface and contrast to background the "Intervis" parameters. Even given a relatively uniformly spaced grid of points across a terrain, the visibility as defined by contrast and exposed surface between any pair is asymmetric. A hidden person exposing only his eye may nonetheless have a perfectly clear view of his target. For n points, there will be a total of $n*(n-1)$ ordered pairs. In contrast, the ray cast approach does not account for this aspect of visual perception (as well as others) and exhibits symmetry. For large n, the possibility of adapting the described technique to use cached depth and framebuffers is intriguing (vam der Leeuw 2009).

While ACQUIRE style detection is certainly more accurate than lines-of-sight, precomputation can only be done for a finite set of locations. If at run time arbitrary locations are approximated by precomputed ones, this will introduce some amount of error. Changes in the geometry in run time will introduce additional errors, unless the set of possible changes is small and can be included in the precomputation.

Algorithm

We compute the Intervis parameters (contrast and visible surface values) for each point as follows:

GenerateVisibilityForScene()
  For each waypoint *n*:
    GenerateVisibilityForWaypoint(*n*)

GenerateVisibilityForWaypoint(*sourcePoint*)
  For each sector *d*:
    Set camera position to position(*sourcePoint)*
    Set camera orientation to direction(*d*)
    Render scene
    Store scene color texture Reference_{color}
    For each waypoint *w* in current field of view:
      RenderAndHarvestVisibility(*w*)

RenderAndHarvestVisibility(*w*)
  Let s(i,j) be the stencil buffer value of the pixel at (i,j).
  Clear the stencil buffer s.t. $s(i,j) = 0\ \forall(i,j) \in s$
  Place target at position(*w*)
  Set up the stencil function s.t.:
    The stencil test always passes
    On depth test pass, set $s(i,j) = 127$
    On depth test fail, set $s(i,j) = s(i,j) + 1$
  Render the scene
  Store scene *color* and *stencil* textures
  GetBoundingBox (*stencil*)
  Compute n_t, the number of target pixels in the bounding
    box such that $s(i,j) \geq 127$
  Compute n_o, the number of occluded target pixels in the
    bounding box such that $0 < s(i,j) < 127$

Compute n_b, the number of background pixels in the bounding box such that $s(i,j) < 127$

Compute the brightness of the target, B_t. Sum $r^2+b^2+g^2$ over pixels such that $s(i,j) \geq 127$. Divide by n_t.

Compute the brightness of the background, B_b. Sum $r^2+b^2+g^2$ over pixels in the bounding box such that $s(i,j)$!= 127. Divide by n_b.

Compute and store the contrast $C = |B_t - B_b| / B_b$

Compute and store the size of exposed surface n_t

Compute and store the occlusion $O = n_o / (n_t + n_o)$

GetBoundingBox(*stencil*)

Determine the bounding i and j values for the target ($i_{min}, i_{max}, j_{min}, j_{max}$), the min and max values of i and j such that $s(i,j)$!= 0.

Let target $i_{size} = i_{max} - i_{min}$, likewise for j

Compute the limits of a 10% padded bounding box:

$bb_{imin} = max(0, i_{min} - 0.05 * i_{size})$,

$bb_{imax} = min(max_i, i_{max} + 0.05 * i_{size})$

likewise for j.

Using the stencil buffer allows us to determine the pixels that belong to the object of interest as well as those that would have been rendered had they not been occluded. The number of occluded pixels can be used to provide the percentage of the object that was actually rendered and will equal $n_t / (n_t + n_o)$ where n_o is equal to the number of occluded pixels. The stencil function used for passing the depth test is straightforward as it just replaces the current value with one that is identified with the object. The case where the depth test fails is less intuitive as failure does not necessarily imply that the object is not present at that location. In cases where the object exhibits concavity or is rendered without backface culling, overdraw will occur. This may cause the object to occlude itself. In these cases, incrementing the current value instead of replacing it allows us to infer that the depth test passed for values within a certain range. Since we have 2^8 possible values for each pixel in the stencil buffer, an id value should be chosen so that the id plus the number of possible overdraws is less than 256.

For our purposes, starting with the mean value of 127 and incrementing as described above was sufficient as the maximum overdraw experienced for any given pixel of our posed skeletal mesh was 8. In cases where the amount of overdraw exceeds the capacity of this method, rendering the back faces with the stencil operation set to decrement (2-sided stencil) should keep the values contained. Contrast values from a single point are visualized in figure 2.

Performance

Contrast and exposed surface values are valuable parameters for modeling intervisibility but current approaches and hardware may not be suitable for computation in realtime. Tabel 1 shows the computation time required on our system for the visibility of a single ordered pair.

	CPU Time	GPU Time
Ray Casting	~0.00015s	0.0s
Intervis	~0.00027s	~0.00561s

Table 1: ***Single frame computation (Intel Q6600 @ 2.40GHz, 2.00 GB RAM).***

Given a target frame rate of 60 frame / sec, a single frame allows for ~16.7ms. The GPU cost for a single Intervis frame is ~5.61ms or about 33.5% of the total frame budget. This is largely consumed by the data transfer from the GPU to the CPU of the color and stencil buffers. Since this operation would only need to be performed once in a single frame, no further GPU cost would be incurred for additional calculations. Cost will scale in multiples of the CPU time for both methods.

If an application needs intervisibility information for every waypoint combination, the data will need to computed offline and made available for quick lookup by the application. Due to symmetry, the total number of waypoints requiring computation for ray casting and Intervis differ by a factor of 2. Ray casting will require (n*(n-1)) / 2 whereas Intervis will require n * (n-1) where n ≥ 2. Table 2 shows the total computation time needed for a set of 3,784 waypoints (2m spacing over 100m by 122m).

	Waypoints Processed	CPU Time	GPU Time
Ray Casting	7,157,436	~0.298hrs	0.0hrs
Intervis	14,314,872	~1.08hrs	~22.3hrs

Table 2: ***Total time for n=3784 waypoints (Intel Q6600 @ 2.40GHz, 2.00 GB RAM).***

Assuming that the data will be precomputed and stored in memory, data size becomes the next issue. Note that when occlusion is total, no data storage is required for waypoints that are not visible. This can result in a dramatic reduction in the amount of storage required. For instance, in our example with n = 3784, the resulting file size (which includes several additional pieces of data per waypoint) was 28.141MB which is ~75% reduced.

	Waypoints Processed	Atomic Data Size	Total Size
Ray Casting	7,157,436	1 bit	.89468MB
Intervis	14,314,872	8 bytes	114.519MB

Table 3: ***Data storage.***

A current computational bottleneck is the transfer of texture data (diffuse and stencil) from video memory to system memory. This could be eliminated if the data was generated on the GPU and only returned the results instead of the source textures.

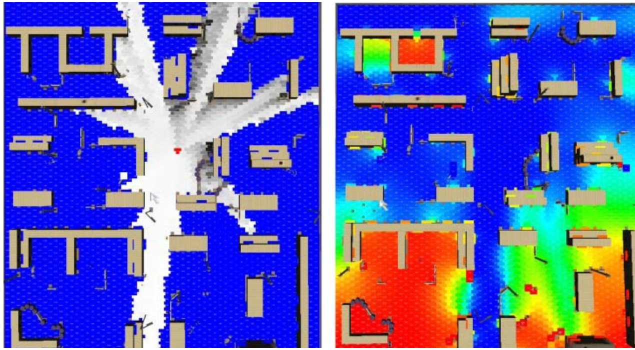

Figure 2: ***Precomputed contrast values from a single (red) point, whiter is better contrast (left), Threat Probability Map of one fireteam that has explored much of the upper region (right)***

Visual Search

A dynamic model of the distribution of perceived threat is maintained for purposes of directing gaze (visual search) and aiming behavior. Even in sophisticated military simulations such as Combat XXI, search is typically modeled as a simple sweep from left to right. This is a very poor model of how infantry actually search for targets in an urban environment. The goal of the Threat Probability Model (TPM) is to provide a basis for synthetic behavior to more rationally scan an area, prioritizing locations that are most likely to contain a threat. Threat in unobserved areas accrues over time, so that when a fireteam enters the area they are driven to scan it thoroughly until the act of looking reduces the probability of threat in the immediate area. In this fashion the TPM and looking behavior affect each other in a feedback loop (Alt and Darken 2008).

Threat Probability Model (TPM)

The TMP used in this application is an Isla occupancy map (cells containing probabilities and diffusing to neighbors), but with two significant modifications. First, the positions and neighbors of the cells are defined by the navigation graph (Darken and Anderegg 2008) rather than being a precisely regular grid. Secondly, when cells are brought in view by a fireteam member, the probability is reduced consistent with his ability to detect a target of the appropriate visible size and contrast in the given amount of time, rather than reducing it to zero. The amount of reduction is controlled by an exponential decay model driven by the detection probability. This is a simplification of the standard AQUIRE detection model that has the advantage of not requiring storage of any data beyond the probability that a cell is occupied by a threat. For debugging and demonstration purposes, the TPM can be visualized as a heat map, with colors in a spectrum from "hot" colors (such as red) denoting high values to "cold" colors (such as dark blue) denoting low values being rendered at each waypoint in the grid (see figure 2). The TPM update cost is given in table 4.

Waypoints	Mean	Std Dev
1112	1.25ms	0.09ms
3785	4.07ms	0.20ms

Table 4: Threat Probability Model update cost *(Intel Q6600 @ 2.40GHz, 2.00 GB RAM)..*

Search Algorithm

In each search frame, a fireteam member independently looks in the direction that is most threatening. Threat is aggregated over a fixed number of angular sectors. Only threats that are potentially visible are taken into account; the cached visibility data is critical in making this summation efficient. After the sector is brought into view, the threat probabilities in each visible cell are continuously reduced as described above.

Movement

The BASE-IT application user interface allows the user to command an entire fireteam as a unit. We focus exclusively on the most typical command, which is to move to an area and watch or provide suppressive fire on an angular sector swept out by the user. Suppressive fire is fire at an *invisible* threat intended to limit his ability to move and shoot back. Shooting at *visible* threats is automatic.

Cost Functions

All movement choices are governed by cost functions that produce cost values for any node of the navigation graph under consideration.

Distance: Total distance moved.

Time: Since waiting is possible, total time to the destination must be minimized in addition to distance.

Cover: The exposed surface of an individual in this location (visible solid angle) is weighted and summed up for all other locations using the visibility data described above. The resulting value is inverted.

Too little dispersion: Fireteam members must not get too close to one another to avoid simultaneous attacks.

```
DispersionCost(FireTeam f, mover m, waypoint w, time t)
  shortestDistance = infinity
  For each mover n in f except m:
    distanceToOtherMover =
      length(posAtTime(n, t) - pos(w))
    shortestDistance =
      min(shortestDistance, distanceToOtherMover)
  cost =
```

max(*minDesiredDistance* - *shortestDistance*, 0)

Too much dispersion: If someone takes fire, his teammates should be close enough to respond.

ComeTogetherCost(FireTeam *f*, Mover *m*, waypoint *w*, time *t*)
longestDistance = 0
For each mover *n* in *f* except *m*:
distanceToOtherMover =
length(posAtTime(*n*, *t*) - pos(*w*))
longestDistance =
max(*longestDistance*, *distanceToOtherMover*)
cost =
max(*longestDistance* - *maximumDesiredDistance*, 0)

Distance to leader: Each fireteam member needs to be close enough to the team leader in order to see hand signals or at least hear a verbal command.

FTLDistCost(FireTeam *f*, Mover *m*, waypoint *w*, time *t*)
If *m* is not teamLeader(*f*):
cost = length(posAtTime(teamLeader(*f*), *t*) - pos(*w*))
Else:
cost = 0

Progress: Determining progress along the fireteam path is one of the trickier costs to specify.

StringPullCost(Mover *m*, FireTeam *f*, waypoint *w*, waypoint *goal*)
stringPullPoint = ComputeStringPullPoint(*w*, path(*f*), CurrentSPPIndex(*m*))
cost = length(pos(*stringPullPoint*) - pos(*w*))
cost = *cost* + length(path(*stringPullPoint*, *goal*))

ComputeStringPullPoint(waypoint *w*, path *p*, int *currSPPIndex*)
newSPPIndex = *currSPPIndex*
If CanSee(*w*, *p*[*newSPPIndex*]:
While *newSPPIndex* + 1 < length(*p*) and CanSee(*w*, *p*[*newSPPIndex* + 1]):
newSPPIndex = *newSPPIndex* + 1
Else:
While *newSPPIndex* + 1 < length(*p*) and not CanSee(*w*, *p*[*newSPPIndex* + 1]):
newSPPIndex = *newSPPIndex* + 1
Return *p*[*newSPPIndex*]

Target in view: Counts how many signficant waypoints (likely threat locations) in the user-assigned sector are visible to the individual.

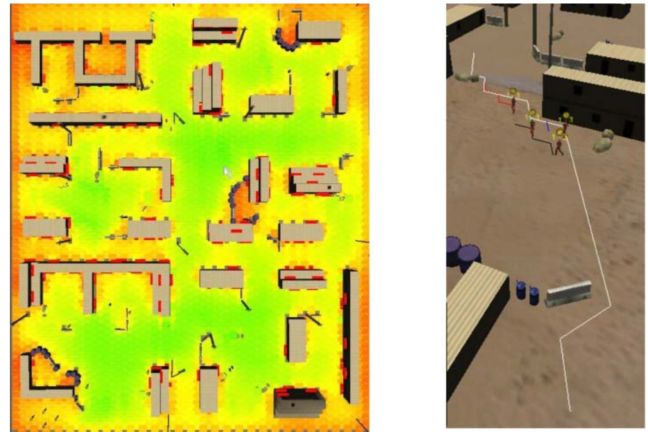

Figure 3: *Cover heatmap, redder is better cover (left), Path generation, white is fireteam path, short colored lines are paths for individuals (right).*

Path Selection

When a fireteam is commanded to move to an area, an A* algorithm is executed across the waypoints to plot a high-level path from the fireteam leader to the waypoint closest to the indicated goal. The cost function used for finding this path includes distance between waypoints plus cover, as described above. Once calculated, this path is then used for reference when subsequently plotting the paths of individual movers.

To avoid problems with getting stuck in local minima (due to a lack of look-ahead) that are associated with steering, as well as the extreme complexity of plotting complete coordinated paths for all team members before beginning movement, a compromise was reached between these two strategies. At regular intervals, each team member performs a greedy search to find a short path of fixed look-ahead to navigate closer towards the goal, and proceeds to navigate along this path until the next search. The times at which each mover finds their next path are staggered, so as time progresses they earch regularly take turns finding a new path, an approach inspired by Silver (2005). If a mover somehow drags significantly behind his teammates, an emergency catch-up fallback engages, and he plans a minimalist A* path to quickly catch up.

It is important that the paths found by individuals do not intersect, since one of the points of this design is to avoid collision checking and steering during movement execution. An Estimated Time of Arrival (ETA) Manager is used to track a schedule detailing the forecast location of each mover at any given future time, which is revised as individuals plan their paths, and which in turn is used for reference when planning a path so as to avoid collisions with other movers (and indeed to maintain dispersion and other criteria). Because the other individuals are moving, the search space explored is not merely *O(num waypoints)* but *O(num waypoints * available time)*, and so is technically unbounded.

Each of several factors are evaluated for each waypoint considered as a candidate for being a segment on a mover

path, each assessed as a cost rating between 0 and 1, then weighted and summed to determine the cost of that waypoint at that time. Factors weighted include dispersion, a punishment for movers grouping too close together; a complementary factor for punishing movers for getting too far apart; a “string pull” factor that rewards forward progress along the main fireteam path; and an incentive to maintain good cover by punishing paths that travel through regions of poor cover, distance to leader, distance moved and time consumed. Note that an individual may choose a "wait" action of fixed duration instead of moving.
The computation requirements in our larger test level are indicated below. For comparison, the longest possible fireteam (conventional A*) paths from corner to corner take about 4 msec to generate. Individual pathing is our most expensive movement-related computation. A look-ahead value of 5 or less is required to avoid "chugs".

Look-Ahead	Mean Time	Std Dev
3	2.6ms	0.6ms
5	13.4ms	2.8ms
10	123.6ms	25.0ms

Table 5: ***Look ahead vs. individual partial path generation time (Intel Q6600 @ 2.40GHz, 2.00 GB RAM).***

Destination Selection

With the first team member to come within range of the goal waypoint at the end of the fireteam path, a set of destinations (one navigation graph node per team member) is selected within a radius from the goal waypoint. This set is determined by evaluating each set of waypoints equal in number to the size of the fireteam within the radius according to various criteria similar to our movement criteria. The complete list of criteria considered includes q cover, dispersion (too much and too little), and maintaining that the target sector to be covered, if any, is easily visible.

For each mover in our fireteam, we find the length of the path (as computed via a conventional A* search with crow-flies distance as the only heuristic) from that mover to each of the waypoints in our selected set of best end waypoints, and store all these distances in a table. Then we consider each possible assignment of the members of the fireteam to the best end waypoints. We add up the total distance traveled by all the marines to get to their respective destinations in each assignment. Of all the possible total cumulative distances traveled by the fireteam in each configuration, we choose the shortest as the best.

Conclusions

The algorithms described have been implemented in a demonstrable prototype involving the simultaneous movement of three fireteams (a Marine squad). The fireteam members now clearly focus their attention on the most threatening areas first, rather than sweeping in an arbitrary pattern. The movement patterns are likewise sensitive to cover and dispersion, and much improved from fixed formations. Still, the simulated behavior is far from perfect. A recent study of Marines exposed to the prototype resulted in a list of a dozen or more items needing improvement. Clearly, at least for these potential users of the system to be pleased, the quest for ever more realistic infantry simulations must go on.

Acknowledgements

Support for this work was provided by the of the Office of Naval Research and the U.S. Army Training and Doctrine Command Research and Analysis Center, Monterey. The comments and suggestions of the anonymous reviewers were greatly appreciated.

References

Alt, J. and Darken, C. (2008). A Reference Model of Soldier Attention and Behavior. *Proceedings of Behavior Representation in Modeling and Simulation (BRIMS) 2008.*

Darken, C. and Anderegg, B. (2008). Particle Filters and Simulacra for More Realistic Opponent Tracking. In S. Rabin (Ed.), *Game AI Programming Wisdom 4*, Boston : Course Technology.

Darken, C. and Jones, B. (2007). Computer Graphics-Based Target Detection for Synthetic Soldiers. *Proceedings of Behavior Representation in Modeling and Simulation (BRIMS) 2007.*

Isla, D. and Blumberg, B. (2002). Object persistence for synthetic creatures. *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems.*

Jurney, C. (2008). Company of Heroes Squad Formations Explained. In S. Rabin (Ed.), *AI Game Programming Wisdom 4*, Boston:Course Technology.

Liden, L. (2002). Strategic and Tactical Reasoning with Waypoints. *AI Game Programming Wisdom.*

Reece, D. (2003). Movement Behavior for Soldier Agents on a Virtual Battlefield. *Presence* 12:4.

Silver, D. (2005). Cooperative Pathfinding. *Proceedings of AI and Interactive Digital Entertainment (AIIDE) 2005.*

Straatman, R., van der Sterren, W., and Beij, A. (2005). Killzone's AI: Dynamic Procedural Combat Tactics. *Proceedings of the Game Developers Conference 2005.*

vam der Leeuw, M. (2009) The PlayStation 3's SPU's in the Real World - Killzone 2 Case Study,Game Developers Conference 2009 presentation, accessible via http://www.gdcvault.com.