Killzone's AI: dynamic procedural combat tactics

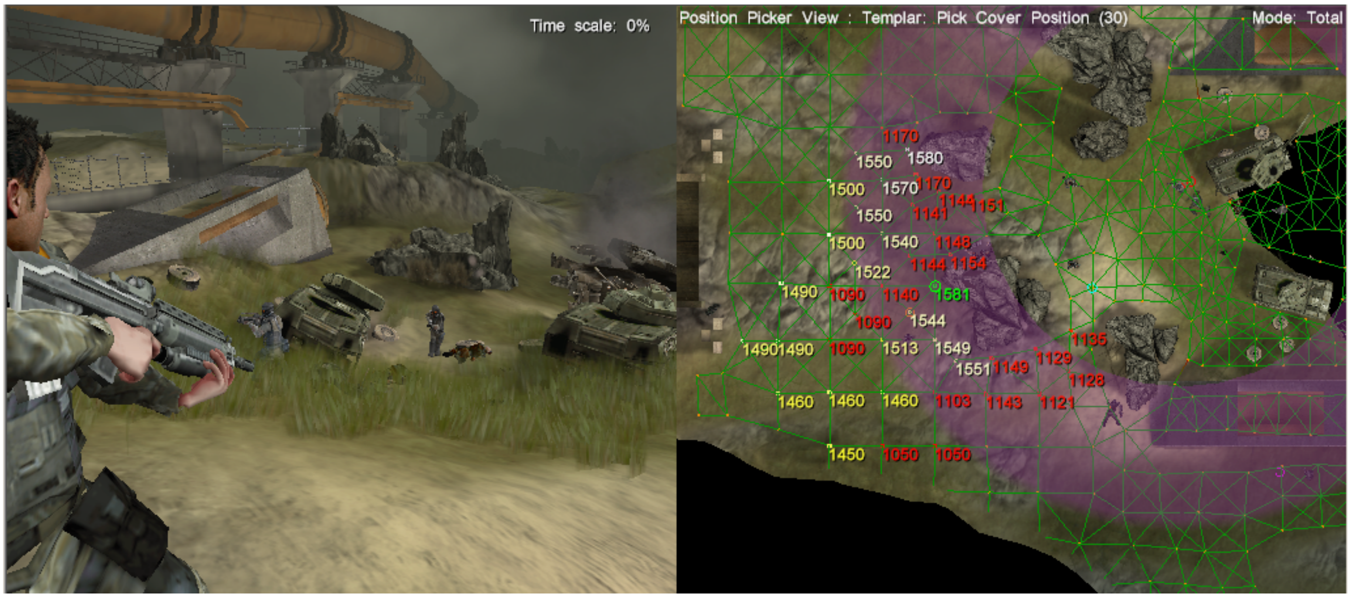
Remco Straatman
Guerrilla Games, Amsterdam, the Netherlands, remco.straatman@guerrilla-games.com
William van der Sterren
CGF-AI, Veldhoven, the Netherlands, william.van.der.sterren@cgf-ai.com
Arjen Beij
Guerrilla Games, Amsterdam, the Netherlands, arjen.beij@guerrilla-games.com

Introduction

One of the main challenges for game AI is to provide intelligent and responsive behavior. Responsiveness to different situations is often difficult to achieve using current techniques, such as level designer placed terrain hints.

In this document we describe the design of 'dynamic procedural combat tactics' AI for Killzone, Guerrilla's shooter for PlayStation2. In Killzone, the AI presents a wide variety of tactics tailored to the dynamic situation and terrain, anytime and anywhere.

Figure 1 Killzone's AI in full combat, as seen through the eyes of the player (left) and the AI developer (right)



We start by defining dynamic procedural combat tactics and a brief overview of Killzone's AI. Then, we describe the concepts and mechanics behind dynamic procedural tactics: position evaluation functions, their application in position picking and tactical path-finding, and the world representation supporting this. Two more examples the construction of dynamic tactics using position picking: indirect fire and suppression. Finally, we describe our experience with dynamic procedural tactics in Killzone, discuss related work and present conclusions.

Why dynamic procedural combat tactics?

Nowadays, more and more first - and third person combat games include a number of the following characteristics:

- freedom for the player to pick his (avenue of) approach to a fight;
- AI squads carrying out player issued commands at locations chosen by the player;
- AI populated multiplayer skirmishes, requiring the AI to fight anywhere on the map;
- changing paths and lines-of-fire due to destructible environment, and moving vehicles;
- authentic military and law enforcement tactics, even in complex and dynamic environments.

A common approach to create tactical AI behavior in a shooter game AI is to use a combination of scripts and level designer placed hints (to mark, for example, cover and ambush spots). The scripts define how the AI responds to specific situations, whereas the hints indicate where the AI could perform specific actions. Although simple, predictable and powerful, that approach has troubles to deal with the game AI requirements listed above.
The need to fight anywhere, against threats from any direction cannot be supported using this approach without placing loads of hints. Statically placed hints are less effective when the positions of the opposing forces are hard to predict, or when the environment is likely to change. Hints typically cannot express subtle properties, such as “this location offers a good ambush spot against distant threats from the northwest”.
Also, AI scripts and their ‘if-then’ constructs are not well suited to interpret and balance the dozens of inputs necessary to tailor a tactical maneuver to the situation and terrain at hand.

What are dynamic procedural combat tactics?

The AI has dynamic procedural tactics when it tailors the tactics to the situation and terrain at hand using of on-the-fly algorithms and dynamic inputs. For example, whenever the AI selects a new position to move to, that position is chosen by taking into account positions and lines-of-fire from threats and nearby friendly units. Whenever the AI selects a path, that path is chosen based on the travel costs and the required degree of cover or stealth from actual threats. Goals are abandoned as soon as the situation, in particular for the selected path and destination, changes for the worse.

The player will recognize this as AI being more responsive to subtle differences in his actions and his choice of position. He’ll see the friendly AI accompanying him act tactically sound even in (common) situations with multiple moving threats, a variety of weapons, partial cover, friendly lines of fire, and incoming projectiles to avoid. The player will experience robust execution of a wide set of tactics, anytime, anywhere on the map.

Level designers can concentrate on the overall flow of encounters, and still get generally good behavior by the AI. They can reduce the use of hints and scripted tactics to those exceptional situations where the default tactics won’t work or the default behavior does not provide the intended behavior.

To provide some context, we first present a brief overview of Killzone's AI before describing its dynamic procedural tactics in terms of concepts and examples.

A brief overview of Killzone's AI

Killzone is a first-person shooter offering primarily contemporary style infantry combat in a near-futuristic setting. In the single-player game, the player traverses thirty-odd single-player levels including trenches, warehouses, mountain passes, city streets and marshes, most of the time accompanied by several buddy AI characters. In addition, Killzone offers six on-line multi-player modes all of which are playable off-line with and against the AI.

To be able to understand where the Killzone AI has been made more dynamic and procedural, we describe the think cycle for an AI soldier. At any time, this soldier pursues the goal that is most desirable in that situation. To pursue that goal, the soldier executes the set of actions corresponding to it. He will continue to pursue that goal until the goal is achieved, the goal becomes unachievable, another goal becomes more desirable, or one of the goal's actions fails.
For combat goals, the corresponding set of actions often resembles a 'fire and maneuver' pattern: pick a destination, plan a path to that destination, move along the path (often in combination with searching, aiming and firing), arrive and adopt the final stance, and start performing a stationary action such as scanning, reloading or staying low.

In Killzone, the AI computes its destination and path completely based on (dynamic) information such as the amount of cover from multiple threats, lines-of-fire, danger zones, and area of operations assignments. Even when not moving, the actions such as suppression and hand grenade attacks also heavily use position evaluation.

The level designer primarily controls the AI by configuring each unit with a set of goals and an area-of-operations (the space to whichthe unit is restricted). From the level script, the level designer issues commands and may manipulate the area-of-operations.

Dynamic Procedural Tactics: Concepts & Examples

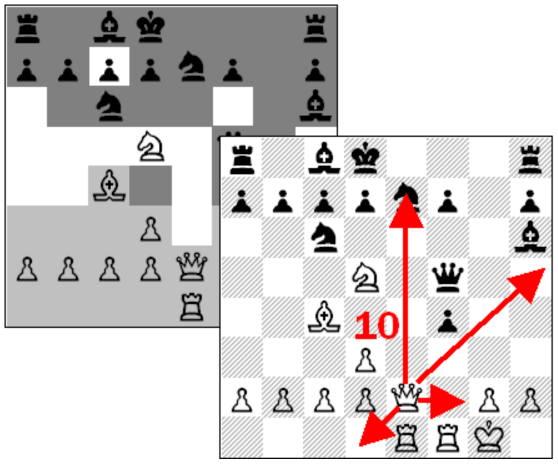
Position evaluation functions are one of the cornerstones of our approach. They enable us to gather a lot of static and dynamic information about the combat situation and combine that into a single number. This in turn allows us to make efficient and robust decisions about where to move, how to move there, where to look and where to fire.
Below we explain how position picking and path finding can use position evaluation functions, and how these algorithms can be configured to provide, tune and vary the behavior.

Position evaluation functions

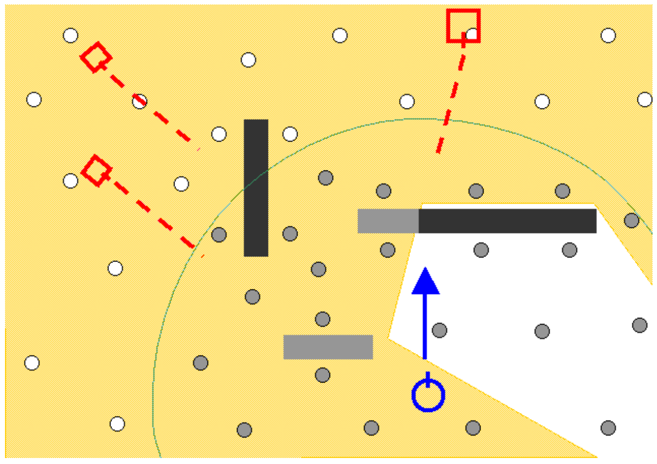
Position evaluation functions are a well-known AI technique, especially popular in computer chess. In computer chess, the AI generates possible board positions (as the result of individual moves), and evaluates these board positions to select the strongest series of moves.

To evaluate a chess board position, a single value is computed from the weighted sum of several easily computed properties such as: the number and weight of pieces, the mobility of each piece, the number of controlled board locations, pawn formations, and king safety [Laramée00].

Figure 2 Position evaluations shooter game AI and in chess AI.



a. Board control and piece mobility in chess.



b. Lines-of-fire and mobility in a tactical shooter.

Position evaluation functions can be valuable in shooter games as well. They can determine which of the positions near an AI soldier is the most attractive to move to in terms similar to chess' mobility, board control and king safety. Position evaluation functions also can express the safety or concealment at a location during path-finding, they can determine the relevance and priority of certain goals, and they can even assist in classifying threats.

An example of an evaluation function in a shooter game is computing the presence of cover on nearby locations from a given threat. Such a function can assist in picking those attack positions that are near cover to facilitate reloading. The next section provides a number of concrete inputs for a position evaluation function in a tactical shooter.
The position evaluation functions offer a modular and scaleable mechanism: new considerations are easily added.

The concept of position evaluation functions, although borrowed from computer chess, does not enable our AI to anticipate hostile actions. Chess AI anticipates worst-case opponent moves in its minimax game tree search, but such a mechanism is not efficiently applicable in 3D shooter games due to their greater complexity and real-time calculations. Instead, our AI compensates for small hostile movements in its representation of line-of-fires (discussed below) and otherwise makes the occasional mistake, not unlike humans.

Tactical position picking

During combat an AI character will frequently change position to achieve its goal. For example, the AI picks new positions to hide, obtain a better line of fire, close in with its target, or to obey a squad command. In general, the AI will attempt to find a suitable position in the direct surroundings of its current position, or in the area the squad is commanding it to.

To select or pick a position to move to, the AI considers all positions within a radius around its current position or area-of-operation's center (see Figure 3.b). From these, the AI eliminates positions already claimed by others, as well as the positions outside the area-of-operations.
For each of the remaining positions the position evaluation function is invoked, and the position is annotated with the resulting score (see Figure 3c-f). Higher scores suggest more attractive positions. The annotated positions are then sorted by score.

A post-processing step is performed to verify that the highest scoring position is suitable for the intended purpose. For example, when picking a cover in the center of a football field, even the highest scoring position will not actually provide cover. This will be obvious from the highest score, provided that the 'cover' input was given a stronger weight than all of the other inputs together. In cases like this the AI may decide to temporarily abort its search for cover.

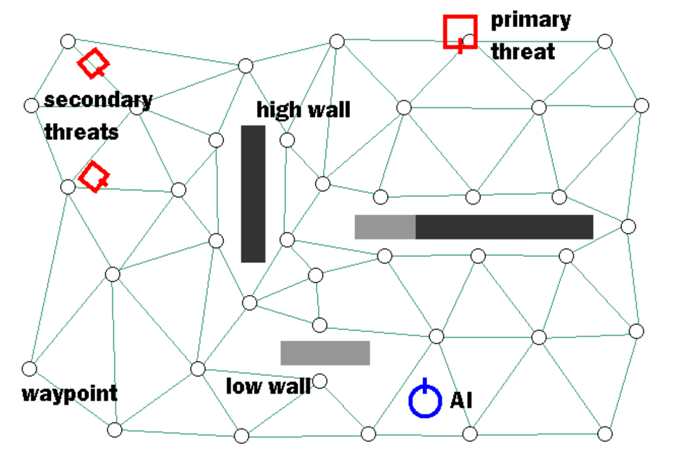
To prevent inputs from canceling out each other, we solely use non-negative valued weights and inputs. We also make sure all inputs reflect the attractiveness of a position. For example, we use safety rather than danger, proximity rather than distance, etc.

Table 1 below provides several examples of evaluation functions used to pick positions.

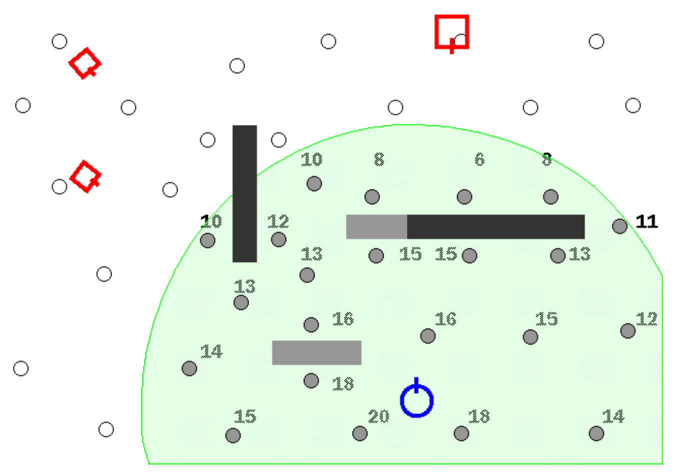
Table 1 Several examples of inputs for the position evaluation function used to pick positions.

Position evaluation input	**Description: (The evaluation function awards a higher score for …)**
Proximity to current position	being closer to or more quickly reached from the current position
Proximity from specific location	being closer to or more quickly reached from a specific location
Cover from primary threat	offering cover from the primary threat (given a stance)
Line-of-fire to primary threat	offering cover from the primary threat (given a stance)
Distance to primary threat	being preferred fighting distance from the primary threat
Outside danger zone	being outside the blast range of (expected) projectile
Cover from secondary threats	offering cover from one or more specified threats other than the primary threat
Outside friendly line-of-fire	being outside the line-of-fire of specified friendly units
Distance from friendly positions	being some distance away from friendly positions
Wall hugging	being close to a wall or obstacle in a specified direction
Nearby cover	being near positions offering cover from the primary threat
Player line-of-fire	not being a position across the player's line-of-fire from the current position
Preferred fighting range	being inside the preferred fighting range

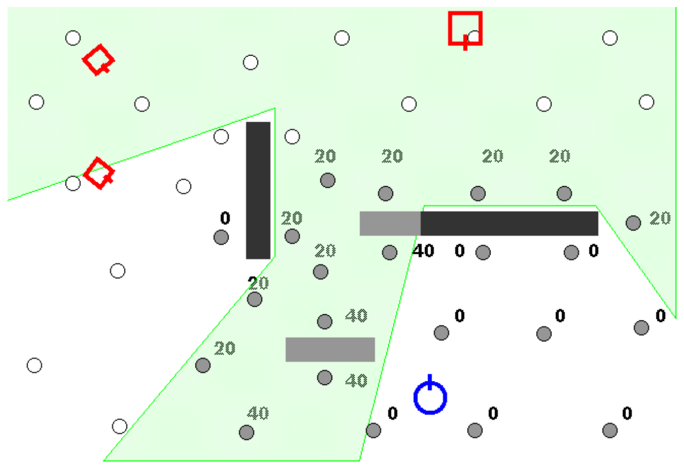
Figure 3 Tactical position picking to attack a threat. The AI wants to move to a position providing a line-of-fire to the primary threat, cover from the two secondary threats, preferably with partial cover from the primary threat and within the preferred fighting range of the threat. The best position to move to is computed in five steps.



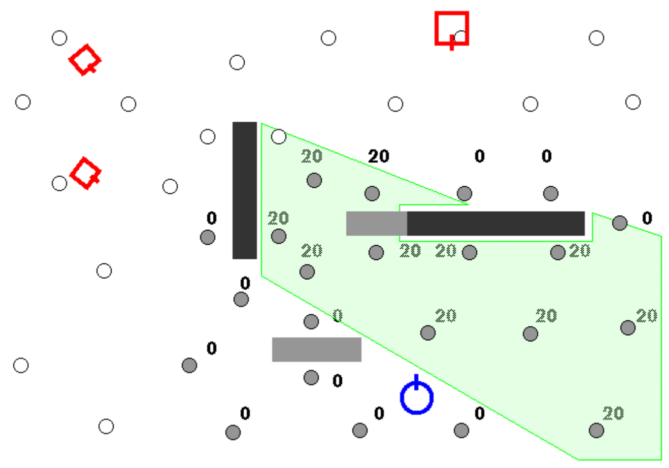
a. Initial situation with waypoint graph and legend.



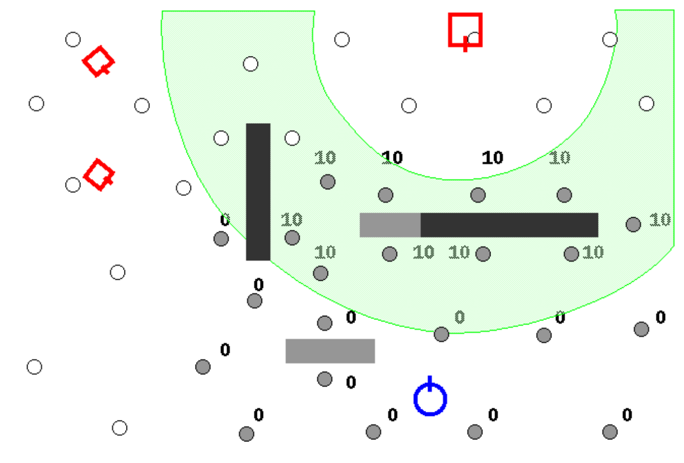
b. Selected nearby waypoints, annotated with proximity.



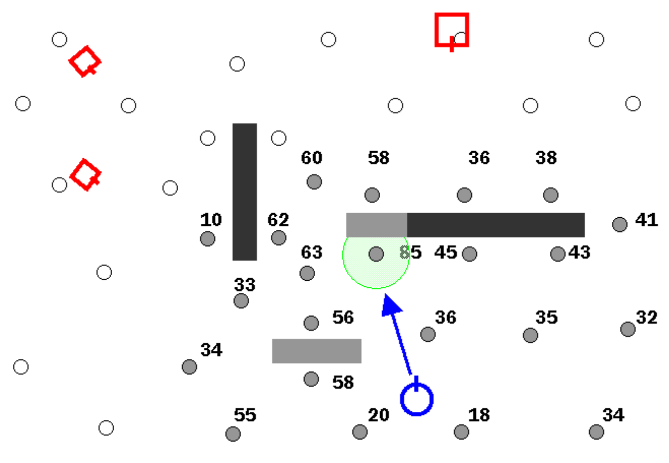
c. Annotations for positions with a line-of-fire to primary threat.



d. Annotations for positions with cover from the secondary threats.



e. Annotations for positions inside the preferred fighting range.



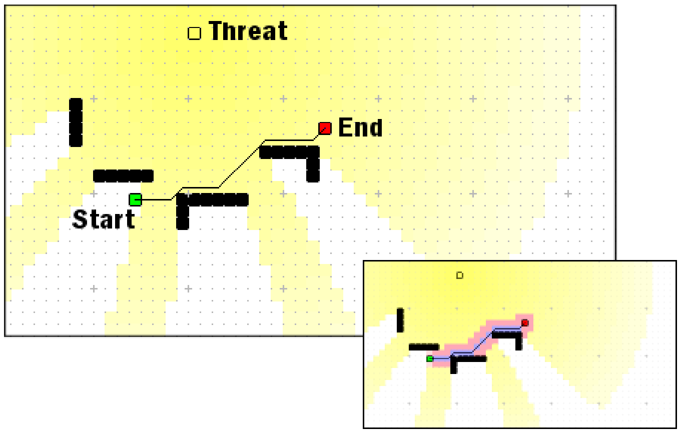
f. Adding up all the annotations yields the best attack position.

Tactical Path-finding

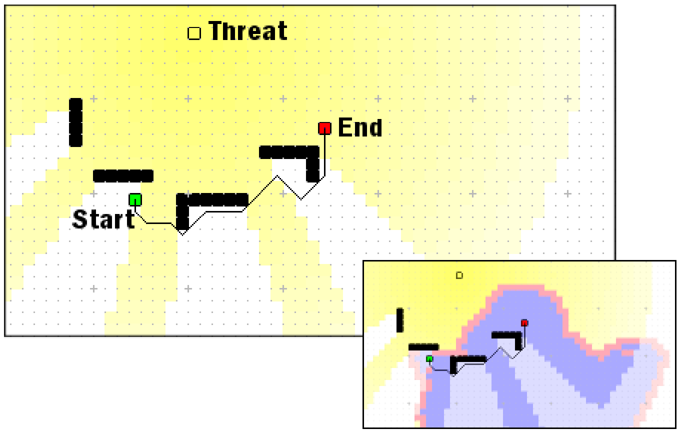
Position evaluation functions can also be used to make AI paths more responsive and tactical. In cost functions for conventional (shortest-path) path-finding, the costs for a move from one position to another solely consist of the corresponding travel costs.
Such a cost function can easily be extended to invoke a position evaluation function for each of the positions visited. This position evaluation function can, for example, add costs to make it more expensive (and less attractive) to cross the player's line-of-fire or to be visible to hostile guards. (In path-finding typically lower costs are better, unlike the position picking discussed above).

In Figure 4 we see the effects of adding costs for visiting a threat's line-of-fire to the path-finder's cost function: the path takes a small detour to avoid the line-of-fire where possible [Sterren02].

Figure 4 Tactical paths as a result from position evaluation functions in the path-finder's cost function [placeholder: turn this into two examples]



a. A shortest path ignoring the threat's line-of-fire. The inset shows how little terrain is explored by the shortest-path A* search.



b. A tactical path avoiding the threat's line-of-fire. The inset shows how much terrain the tactical A* search explores when it is not clamped to an area-of-operations

Although it is relatively straightforward to extend a path-finding algorithm with a position evaluation function, it then is more difficult to keep the run-time costs of that algorithm under control. The addition of position evaluation functions may cause the path-finder to explore a large part of the terrain (as shown in the insets in Figure 4).

In Killzone, we assign area-of-operations to each AI unit to control and limit where the units move, fight and hide. Consequently, we can have the A* path-finder restrict its search to the unit's area-of-operations. This keeps the path-finding efficient and the resulting paths predictable even in dynamic situations with several hostile and friendly lines-of-fire.

Creating unique behavior and tactics by parameterization

For game AI, configuration facilities are essential to create distinctive behavior the different character types.

Position evaluation functions offer plenty of opportunities for configuration and tuning because they combine several inputs in a weighted fashion.
These weights are easily manipulated to value one input over another, or to fully exclude some input. For example, different behavior is achieved by changing weights in the attack position evaluation function so it to prefers out in the open positions over positions near cover.

Aside from the weights, the inputs themselves can be parameterized as well. For example, the preferred fighting range can be used to make the AI fight close to a threat or keep a certain distance.

World Representation

In the complex game world of a 3D shooter, the AI's actions are only as good as their world representation. When the AI does not know how a position can be accessed or how the surroundings block line-of-fire, the AI will never move there for cover.

The minimal world presentation necessary to support the tactical behavior described in this paper consists of:

- navigation info supporting position enumeration and connectivity queries
- visibility info supporting line-of-fire (LoF) and line-of-sight (LoS) queries.

Navigation, position claims and danger zones, and areas-of-operations

The navigation system has to be capable of generating all accessible locations near a given position, and of listing all locations in direct connection to a given accessible location. It does not matter whether the navigation system is based on meshes, on cells or on waypoints, as long as its granularity matches the size of the individual cover and attack positions.

Killzone is waypoint based, with a waypoints placed about every 2 meters and closer to cover where applicable. For navigation purposes, we maintain a connectivity graph for the waypoints, where the links are annotated by estimated travel time. This is a lightweight representation, where it would be easy to adapt the structure on the fly in response of terrain changes.

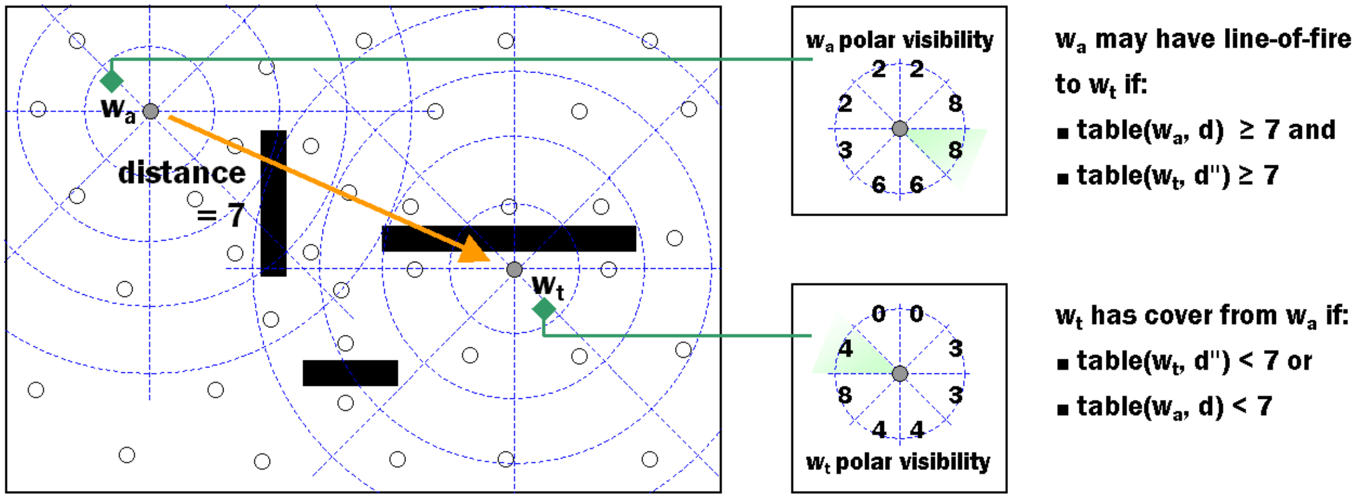
The Killzone AI uses some additional information in its position picking and path-finding. All AI units can claim as their destination positions. Other friendly units then attempt to not to pick these claimed positions or move through them.

An AI unit also keeps track of (observed) entities that present actual dangers such as an incoming grenade. Danger zones are used to represent the area where danger exists.

Lines-of-sight, lines-of-fire and a ‘worst-case’ visibility lookup table

Killzone, like many other 3D shooter games, features complex character models and world geometry. It also allows the player and AI characters to assume multiple postures (stand, crouch). As a result, it is not easy to answer whether one character is able to see (part of) another character. And it is certainly difficult to answer whether a character would be able to find cover at any of the nearby locations from multiple threats who are assuming postures offering them the best line-of-fire. However, the tactics we want to create in our AI heavily depend on these kinds of queries.

Figure 5 An example of the waypoint based polar visibility info and the line-of-fire and cover queries.



To efficiently answer the numerous line-of-sight and line-of-fire checks, Killzone uses a look-up table for waypoint visibility, for standing and crouched stances. To fit in a modest amount of memory, this table does not attempt to record accurate visibility for each pair of waypoints. Instead stores an approximation of that visibility in a polar representation:

> For each waypoint w_t, each stance *s*, and each direction *d* in the polar representation, the table entry for (w_t, *s*, *d*) contains the largest distance for which an AI actor in stance *s* at or near waypoint w_t does not have cover from an attacker in some stance *s*’ at or near some waypoint w_a positioned in direction *d* relative from w_t.

Using this table, an AI character positioned in stance *s* near a waypoint w_t is guaranteed to have cover from a ground based attacker at distance *z* in direction *d* if *z* is greater than the distance recorded for w_t, *s* and *d* . The reverse is not necessarily true: when a threat is closer than the recorded distance for w_t, *s* and *d* , this threat need not have a line-of-fire.

We assume symmetry for lines-of-fire. Solely when the table states that w_t does not have cover from w_a, and that w_a does not have cover from w_t, there might be a line of fire from w_a to w_t. Based on this assumption, w_t is also guaranteed cover from w_a if the table states that w_a has cover from w_t.

This table represents a “worst-case” assumption about cover for a given direction and distance: if there is a single position at a certain distance and in a certain direction from which an attacker can establish a line of fire in some stance, then it is assumed there is no cover from any position represented by the same distance and direction.

This “worst-case” cover assumption has two important benefits. First, although our table is inaccurate in its representation of line-of-fires, the AI can fully rely on a position offering cover when the table says so. Solely statements about a position not offering cover (thus having a line-of-fire) may be too optimistic and require verification with a ray cast. This is attractive, because in position picking and path-finding, the AI has a far greater need for reliable positive statements about cover than about lines-of-fire. For example, to pick an attack position the AI uses a line-of-fire query, but will also query for partial cover from the target and full cover from any secondary threats.

A second benefit of the “worst-case” cover assumption is a degree of robustness against threat movement and stance changes. Even when a threat effectively does not have a line-of-fire when the AI picks a position or plans a path, if that threat easily could establish a line-of-fire from a nearby position then our lookup table most likely states that the AI does not have guaranteed cover from the threat. The resulting robustness can be regarded as a limited ability to anticipate threat movement and stance changes.

The visibility table’s polar representation results in a memory consumption that is linear in the number of waypoints. For 2,000 waypoints, Killzone’s visibility look-up table uses a mere 32Kbyte.

Position evaluation range and the AI decision cycle

For the AI to be dynamic, tactically sound and CPU efficient, it is important to balance the position evaluation range and the decision cycle. The position evaluation range determines the size of the area in which the AI searches for the best position. The decision cycle determines how frequently the AI adopts a new goal and starts another ‘fire-and-maneuver’ sequence of actions.

The smaller the AI’s position evaluation range, the more likely the AI foregoes nearby good tactical positions. A smaller position evaluation range results in shorter movements, leaving the AI more time to fire its weapon. Shorter movements also increase the probability of ending up in the situation as computed by the position evaluation function, since threats have less opportunity to move.
A larger position evaluation range leads to a larger choice of positions and, on average, a better fighting position. However, longer movements increase the chances of threat movement, and also increase the ‘maneuver’ part at the expensive of ‘firing’. Furthermore, a larger position evaluation range results in a larger CPU load, since more positions are to be evaluated.

Because of this, position evaluation ranges should match the AI’s decision cycle. That is, the travel duration corresponding to position evaluation range should be in the order of the AI’s decision cycle. Much smaller ranges cause the AI to fight from bad positions, whereas much larger ranges lead to a series of movements to promising locations without getting there.

Killzone uses position evaluation ranges between 5m and 15m (corresponding to respectively 1.0s and 3.0s decision cycles).

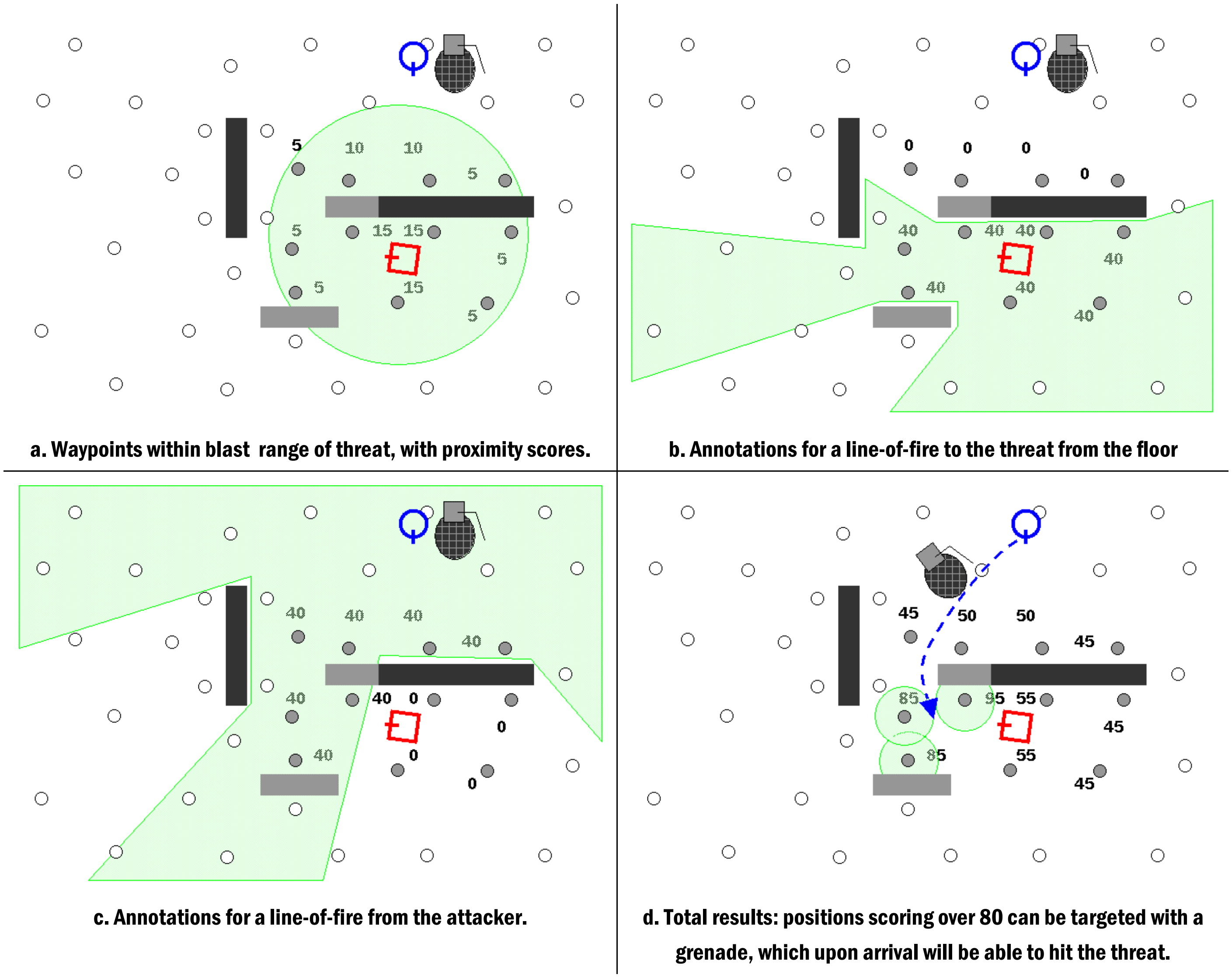
Dynamic Procedural Tactics: Advanced Examples

The position picking mechanism has applications beyond generating movement for an AI character. We can also use position picking to reason about a threat's position and how the threat may be vulnerable to indirect fire. We can reason about the moves available to the threat how to deny him these by suppression fire. This section illustrates both these tactics.

Indirect fire: Hand grenades

Indirect fire involves attacks without having a line-of-sight to the target. In Killzone, grenades, missiles and tank shells can be used for indirect fire, because of their blast damage.

Figure 6 Killzone's grenade attacks use position picking to select promising target locations against hidden threats.



a. Waypoints within blast range of threat, with proximity scores.

b. Annotations for a line-of-fire to the threat from the floor

c. Annotations for a line-of-fire from the attacker.

d. Total results: positions scoring over 80 can be targeted with a grenade, which upon arrival will be able to hit the threat.

In order to deliver a grenade at a position from where it can damage the target, we need a destination within blast range of the target, and two lines of fire: one LoF from the grenade's destination to the target, and a second LoF from the attacker to the grenades destination.
With the position picking mechanism, we can select the most suitable grenade destinations. Only for those destinations we have to check the corresponding trajectories with (expensive) ray casts.

To generate suitable grenade destinations, the AI performs the following steps. First it checks whether a hand grenade attack is safe, for example, based on the distance and proximity of friendly units. Then it uses the position picker to:

- select positions within blast range distance of the threat, and annotate their proximity with a small weight *x* (Figure 6.a);
- annotate positions with weight *y* that have a line-of-fire from a prone attacker to the target, assuming that the grenade will end up on the floor (Figure 6.b);
- annotate positions with weight *z* that have a line-of-fire to them from the AI's position (Figure 6.c).

If we have chosen weights *x*, *y* and*z* so *x* is smaller than both *y* and*z* , we now only need to select these positions whose annotation is greater or equal to *y* + *z* (Figure 6.d).

The result is an AI capability to surprise and attackout of sights threats by delivering grenades near them through windows, or into alleys or trenches, without consuming many ray casts.

Suppression Fire

The position evaluation system has more applications than simple position picking. We also implement the 'suppression fire' using the same approach.

The aim of suppression fire is to reduce or remove the opposing force's ability to fire and/or move. The AI can do so by firing at those positions where a (hidden) threat is likely to attack from or move through. Because lines-of-fire are typically symmetric, the positions that the AI should fire at to suppress the threat can be approximated by taking those positions near the threat that offer this threat a line-of-fire to the AI attacker.

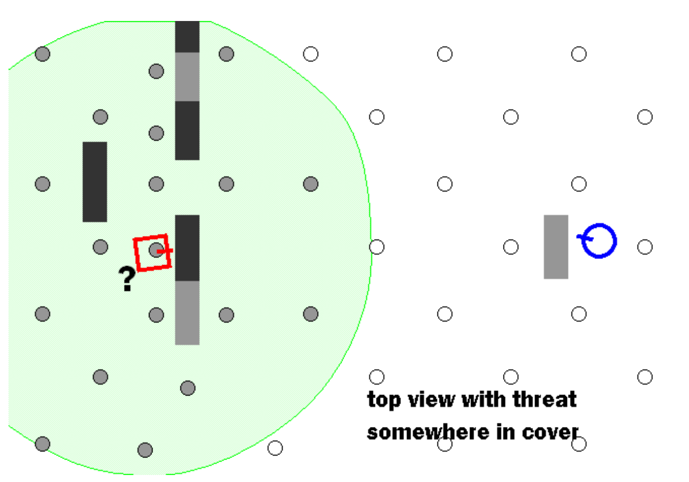
With that understanding, we can create an efficient and tactically correct pattern for suppression fire as follows. First we use position evaluation to determine those positions that are:

- easily reached from (the predicted) position of) the hidden threat (Figure 7.a); and
- offer the threat a line-of-fire to the attacker (Figure 7.b); and
- have cover positions from the attackers nearby (Figure 7.c).

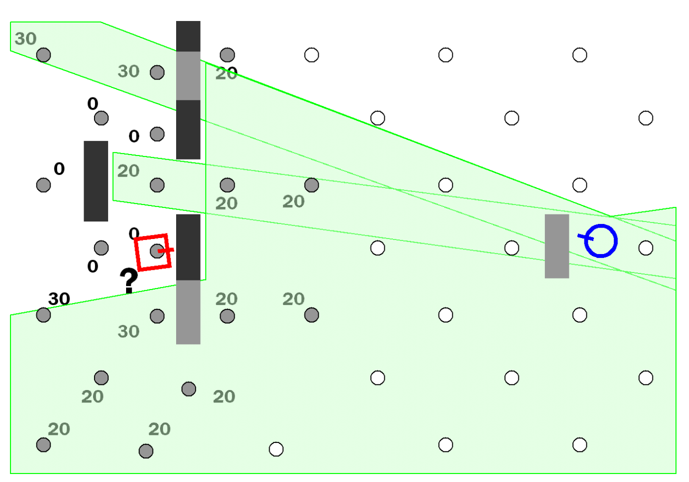
We then select only those positions scoring on all three of these properties (Figure 7.d). Finally, we create firing zones from the selected positions, merging positions that overlap in their pitch and yaw as seen by the suppressing AI (Figure 7.e).

The result is an AI capable of sensibly suppressing its threats (Figure 7.f), without silly mistakes such as firing into positions the threat cannot reach, or into walls between the AI and the threat.

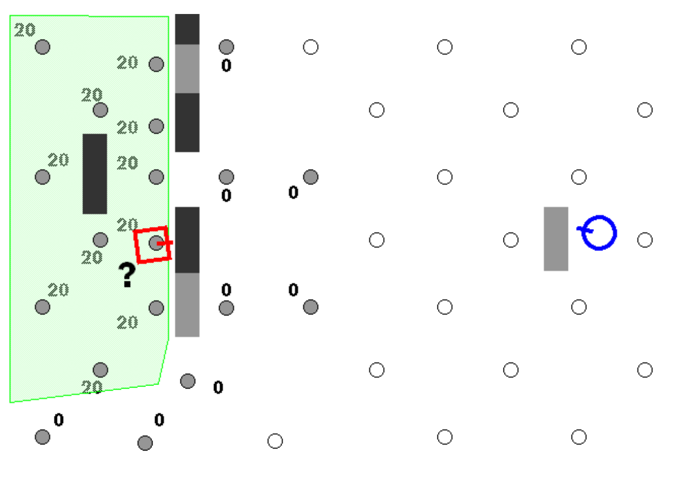
Figure 7 Suppression fire: computing the target positions to fire at in order to pin down the hidden threat.



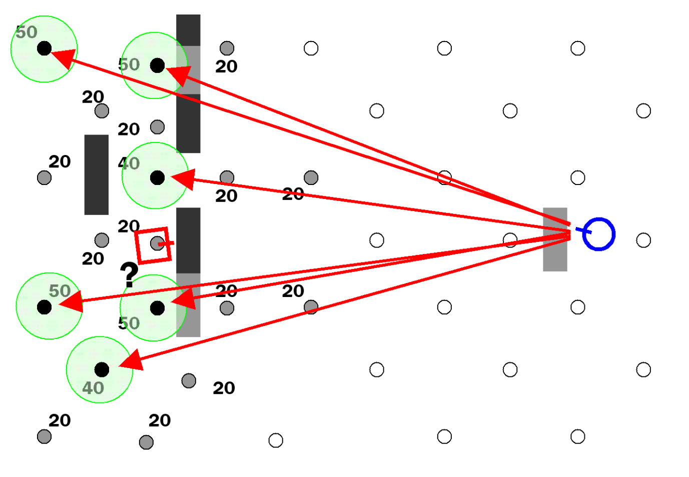
a. Selected waypoints near the hidden threat.



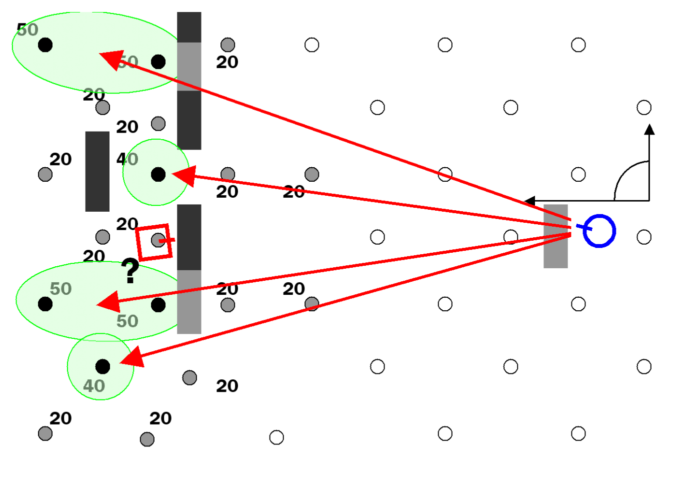
b. Annotations for offering the threat a LoF to the attacker



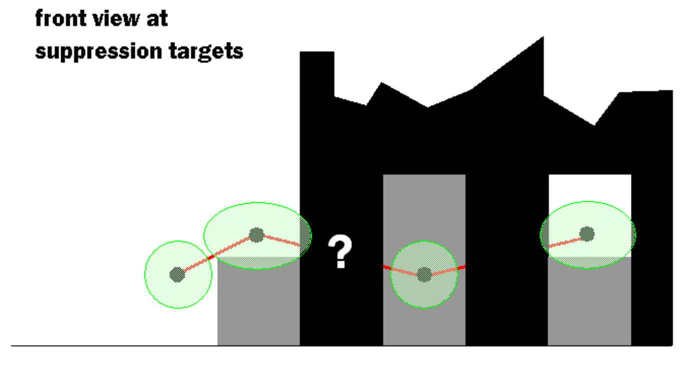
c. Annotations for offering nearby cover from the attacker



d. Selecting positions with score ≥ 40 yields the suppress targets.



e. Merging targets that overlap in yaw and pitch.



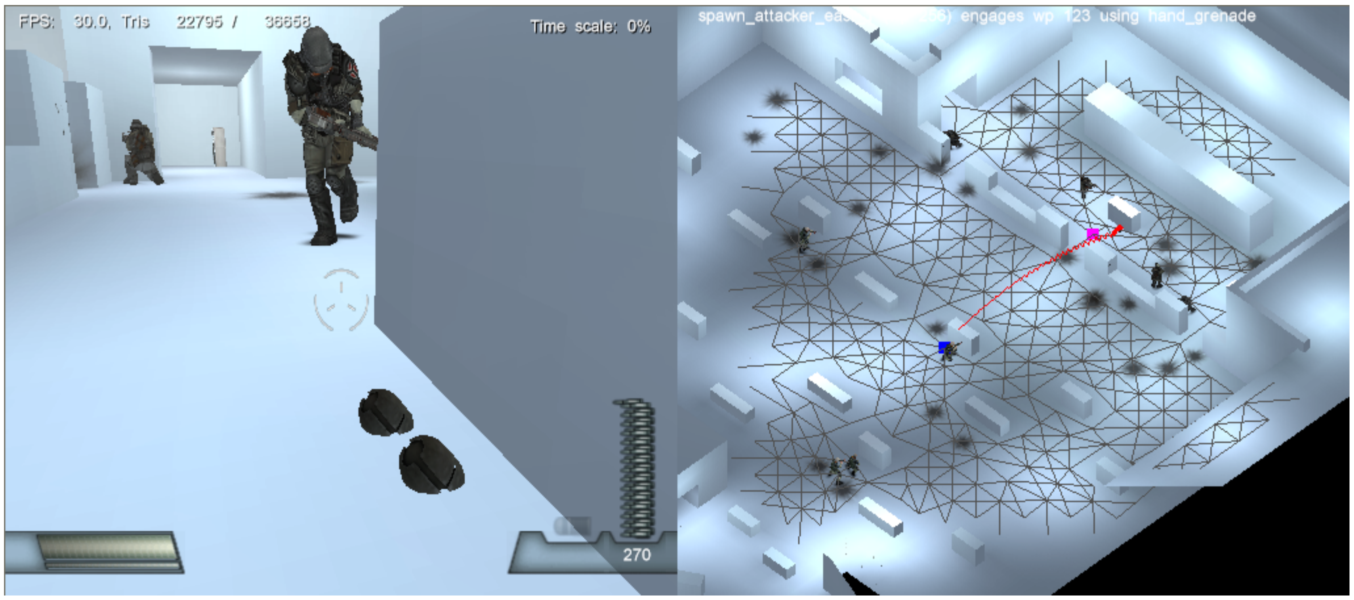
f. Finally, the suppression targets that surround the hidden threat.

Applying dynamic procedural tactics in Killzone

Late 2004, we completed Killzone, our first-person shooter for PlayStation2, published by Sony. Killzone's AI relies heavily on dynamic procedural tactics, both in its single player campaign, and in the off-line multi-player game modes. In the multi-player games modes, up to 14 AI characters may be autonomously fighting each other all over the map.

Tactical position picking is an essential part of just about every infantry action in the game, whether fighting other infantry, armored vehicles or aircraft. Because AI controlled vehicles and flying jet bikes travel along fixed paths, there was no need for them to use position picking.

Figure 8 To tune and test the Killzone AI, we often used a combined game and debug view



Tactical path-finding is used by ground based infantry for the majority of its path-finding. We also experimented with tactical path-finding for squad maneuvers, but did not have sufficient development time to include it in the game.
All squad maneuvers communicate the intended area-of-operations in each of their commands to the squad members. They also dynamically vary the size of areas, depending on the situation.

Suppression fire is used by all infantry when they carry automatic weapons or man a fixed gun. The suppression fire behavior successfully brings bullet whizzes and bullet impacts near the player, even when he is hiding.

Indirect fire based on position evaluation as described before was used both by infantry to deliver hand grenades. The infantry can easily do the same with rifle launched grenades and rockets, but that behavior was too challenging for the player and therefore disabled in the AI's configuration. The AI tanks also use the indirect fire mechanism to fire shells within blast range of hidden threats.

A strong demonstration of Killzone’s dynamic procedural tactics is that we were able to turn our single player AI into multi player AI with only a little effort. The designed-in AI ability to fight with proper tactics anywhere on the multi-player maps was appreciated by many gamers.

During the development, the 'procedural' nature of the AI often was an advantage. Because the AI tactics were defined in code, rather than in level specific scripts and hints, improvements could be rolled out game-wide with every new build. For the same reason, Killzone was able to benefit from behavior and improvements developed for Shellshock: Nam ’67 (which share a considerable part of the AI with Killzone) and vice versa.

The ‘procedural’ nature of the AI made it more complex to test and tune that AI: additional effort was necessary to create environments and configurations in which the AI would be forced to display the tactics to be tested.
To speed up development and debugging, we developed many in-game debugging views to inspect the details of the AI's position picking, path-finding and tactics (Figure 8).

Performance measurements late in the developed cycle showed that position evaluation and tactical path-finding consumed only a modest part of the AI's CPU budget. The use of position evaluation functions also has reduced the amount of ray casting. Because we first evaluate the nearby positions for their attractiveness, we can rule out bad candidates before we spend ray casts.

Level designers initially had troubles getting sufficient control over the AI. For example, when the game design called for AI movement along a specific path, the AI might choose 'smarter' path. We reduced this problem by giving level designers finer control over the AI paths through the scripting system.
Level designers do appreciate how little effort is required to create a representative AI encounters in a new (or modified) level: just place the waypoints and spawn a few squads.

Related Work

Our approach is similar to influence maps [Tozour01] in that both add up several influences and project them on the terrain. However, influence maps typically represent the whole map, are shared by multiple units, and represent slowly propagating influences. A single shared influence map cannot answer the needs of a specific tactic for an individual unit, such as ‘preferred fighting range’, ‘cover from secondary threats’ and ‘damage from a grenade if it is thrown there’. Having a global view also is overkill for most tactical decisions.
We do see influence maps as useful additional inputs for our position picking, for example to represent the terrain recently surveyed by all units of a single force.

The concept of spatial databases for runtime spatial analysis [Tozour03] is close our use of position evaluation functions. Except for the visibility information, we do not bother to keep data around in some kind of database and prefer to compute all required information on demand.

Tactical reasoning based navigation info is not new, but is often performed off-line [Sterren01], relying on level designer placed hints [Reed03], or using only a few inputs [Lidén02]. We are happy to have constructed an AI capable of a large amount of dynamic reasoning at run-time using a single mechanism, without relying on level designer placed hints.

Conclusions and Further Work

Using dynamic procedural tactics, the Killzone AI provides responsive, tactical opponents and allies anywhere on the map. Because of the linear nature of the single player campaign, we might have created similar AI behavior for the single player game using level specific hints and scripts. Catering for the Killzone's different character routes and playing styles would already have been a lot more difficult without our approach. However, we wouldn't have been able to create solid multi-player AI in that way, and subsequent projects would not have benefited as much from the development, tweaking and tuning done for Killzone.

We now have a system in place that is scalable: many new tactical demands can be dealt with by adding new or improved inputs to the position evaluation functions used for position picking and path-finding. Because of its procedural nature, this system also is ready to benefit from advances in CPU technology.

We expect this approach to dynamic procedural tactics to be beneficial to other games with a strong 'fire and maneuver' character, whether dealing with infantry combat, armored combat, or stealth and evasion.

References

[Laramée00] Laramée, Francois, *Chess Programming VI: Evaluation Functions*, 2000. http://www.gamedev.net/reference/programming/features/chess6/

[Lidén02] Lidén, Lars, *Strategic and Tactical Reasoning with Waypoints* in AI Game Programming Wisdom, Charles River Media, 2002.

[Reed03] Reed, Christopher and Geisler, Benjamin, *Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System* in AI Game Programming Wisdom 2, Charles River Media, 2003.

[Sterren01] van der Sterren, William, *Terrain Reasoning for 3D Action Games* in Game Programming Gems 2, Charles River Media, 2001

[Sterren02] van der Sterren, William, *Tactical Path-Finding with A** in Game Programming Gems 3, Charles River Media, 2002

[Tozour01] Tozour, Paul, *Influence Mapping* in Game Programming Gems 2, Charles River Media, 2001

[Tozour03] Tozour, Paul, *Using a Spatial Database for Runtime Spatial Analysis* in AI Game Programming Wisdom 2, Charles River Media, 2003

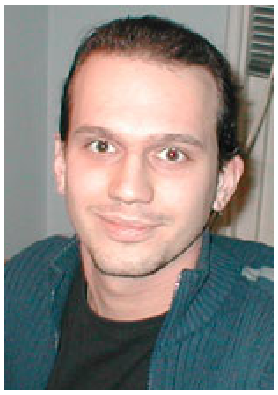
Acknowledgements

We specifically want to thank Robert Morcus, who also contributed a lot to the work described in this article. Furthermore we like to thank everyone at Guerrilla, SCEE and SCEA who contributed to Killzone and Shellshock, and gave us feedback on the AI we developed during these projects.

Biographies

Remco Straatman

William van der Sterren


Arjen Beij

Remco Straatman headed the AI development for the Killzone and Shellshock: Nam'67 titles at Guerrilla Games. Before joining Guerrilla, Remco worked as a researcher in the field of expert systems and machine learning and developed multi-media software. Remco Straatman currently is the lead programmer for one of Guerrilla’s new titles.
(remco.straatman@guerrilla-games.com, http://www.guerrilla-games.com).

William van der Sterren is an AI consultant for games and simulations at CGF-AI. He assisted Guerrilla Games in creating the AI system used in Killzone and Shellshock. William authored articles on squad AI, AI terrain analysis, and tactical path-finding for the Game Programming Gems and AI Game Programming Wisdom book series.
(william.van.der.sterren@cgf-ai.com, http://www.cgf-ai.com)

Arjen Beij is an AI programmer on Guerrilla Games' generic AI system and the tactical first person shooter Killzone specifically. Before joining Guerrilla three years ago he developed multi-media software and games at Lost Boys interactive for two years.
(arjen.beij@guerrilla-games.com, http://www.guerrilla-games.com)