29

Hierarchical AI for Multiplayer Bots in Killzone 3

Remco Straatman, Tim Verweij, Alex Champandard, Robert Morcus, and Hylke Kleve

29.1 Introduction

First-person shooter (FPS) games often consist of a single-player campaign and a large competitive multiplayer component. In multiplayer games, some players' slots can be taken by *bots*, AI controlled players that mimic human players for training purposes. This section describes the AI techniques used to create the bots for Killzone®3, a tactical FPS released on the Playstation®3.

Killzone bots have been used both in an offline training mode with only one or two human players in the game and in multiplayer games with any number of human and bot players. Killzone's main multiplayer mode, *Warzone*, has a number of features, such as multiple team-based game modes being played on the same map in succession and class-based player abilities, which lead to specific requirements for the AI. The chosen approach is inspired by AI techniques from strategy games (hierarchical chain-of-command AI, use of influence maps) and leans heavily on planning and position picking techniques developed for our single-player campaigns.

In this article we describe the scope of the system, the chosen architecture, and the three layers of the architecture. We provide details on the techniques that make up the various layers and describe how we model behaviors. We conclude by describing our experiences, experiments, and future directions.

29.2 Scope

Warzone pits two teams (called *ISA* and *Helghast*) of up to 12 players against each other. During one round on a single map, seven different game modes play out in random order and the team that wins the most game modes wins the round. The game modes are:

- *Capture and hold* (teams gain points by controlling objects on the map)
- *Body count* (teams compete for the most kills)
- *Search and retrieve* (both teams try to return an object to their respective return point)
- *Assassination* (attacking team tries to eliminate one specific player on the other team)
- *Search and destroy* (attacking team tries to destroy an object of the defending team)

The last two game modes are played twice, with each team playing the defending role once.

Players and bots can pick one of five classes which give them access to specific weapons and abilities. For example, the engineer class can place automated turrets, while the medic class can heal teammates. Other abilities include calling in flying drones, disguising as an enemy, cloaking, placing mines, etc.

Each map contains a number of tactical spawn points (TSPs) that can be captured by the tactician class. Team members can spawn at captured TSPs. Mounted guns, ammo boxes, and vehicles (exoskeletons called Exos) are placed at fixed spots on the maps; these can be destroyed or repaired by engineers so that they are usable by the team. For a new player joining *Warzone*, there are many things to learn. To assist in the players' learning process, the bots should mimic experienced human players, master the maps, use the abilities, and work together to win each specific game mode. They should show high level strategies, tactics, and appropriate use of abilities.

29.3 Architecture

The multiplayer AI system is set up as a three-layered hierarchy, where each layer controls the layer below it and information flows back from the lower layers to the higher one. Figure 29.1 illustrates the architecture. Each team has a hierarchy responsible for its bot players. The *strategy layer* is responsible for playing the current game mode, and contains the *commander* AI. The commander monitors the state of the game mode, assigns bots to squads, and issues objectives. The *squad layer* contains the AI for each of the squads. The *Squad AI* is responsible for translating objectives into orders for its members, group movement, and monitoring objective progress. At the lowest level, the *individual layer* consists of the individual AI of the bots. The *bot AI* follows squad orders, but has freedom in how to execute those orders. Bot AI mainly deals with combat and using its class abilities.

The communication between the layers consists of orders moving downward and information moving up. The commander can order squads to *Attack area*, *Defend area*, *Escort player*, or *Advance to regroup point*. Squads will report to the commander on successful completion of orders or imminent failure—for instance, when the squad has been decimated during an attack. Squads will order their members to move to an area, attack a target, use a specific object, or restrict bots to specific areas. Bots will report back completion of orders and send information on observed threats.

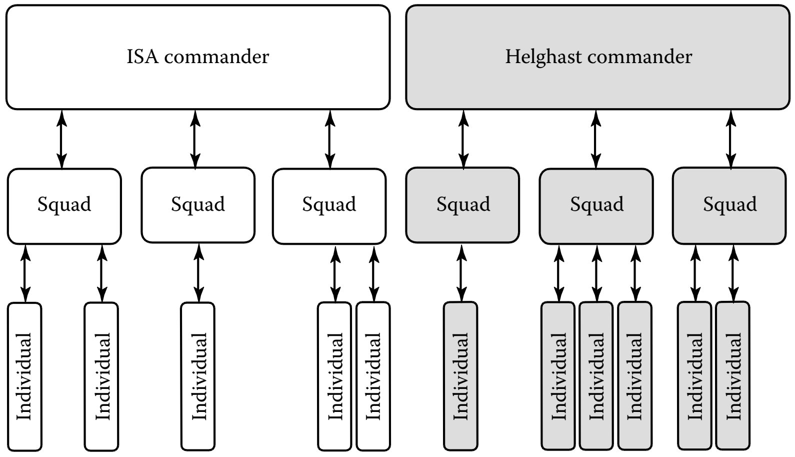

Figure 29.1

Hierarchical layered AI for both teams.

The previous sections have shown the scope and overall architecture. The following sections describe the individual, squad, and commander AI layers in more detail.

29.4 Individual Bot AI

At the lowest level in the hierarchy we find the individual AI for each bot. Even though the bots get orders from their squad, they are still highly autonomous agents. The squad does not micromanage each bot; bots gather information and decide how to fulfill their role. For example, bots can select their own attack target, the position to attack from, the weapon to use, and so forth. They can also decide to temporarily ignore orders in order to survive. Bots have the same weapons and abilities as human players.

29.4.1 Individual Update Loop

The bot AI takes a typical agent-based approach, as shown in Figure 29.2. First, the NPC gathers information about the current world state, either through *perception* or by messages from other team members. Agents perceive *stimuli* in the game world through various *sensors*. There are different sensors for seeing, hearing, and feeling, and different types of stimuli (visual, sounds, and contact). The sensors have limitations such as view cones and maximum distance. See [Puig08] for general information on perception systems. Abilities such as cloaking and disguise further influence perception and make the AI react believably. The information on threats derived from perception is placed in each individual's *world database*. Because of the limitations in perception, the bot's idea of the world may be believably wrong. Besides perception and messages from other agents, a number of *daemons* fill the agent's database with other data. A daemon is a component that adds facts about a specific type of information such as the state of the bot's weapons and health.

In the next step, the *planner* either generates a *plan* or continues execution of the current plan. A plan consists of a sequence of *tasks*. The set of tasks available to the bots is typical for a shooter (such as move to destination, select weapon, reload weapon, etc.).

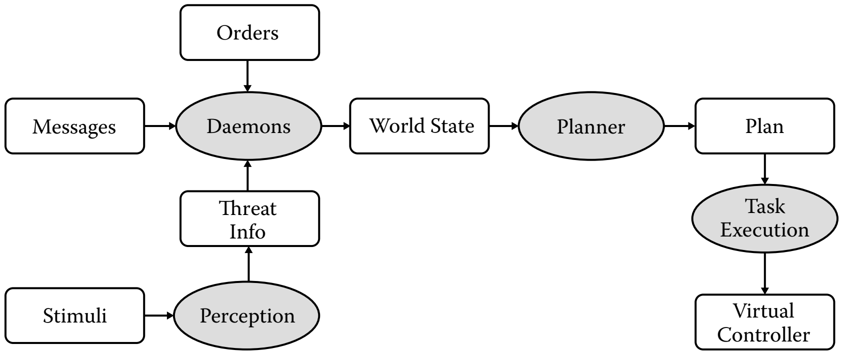

Figure 29.2

Individual update loop.

Each update, the current task of the plan gets executed. Completion of the task advances the plan to the next task, while failure will invalidate the whole plan. Executing a task updates the lower level *skill planner* that takes care of coordinating lower granularity actions such as looking, aiming, walking, crouching, etc. The skills finally update the bots' *virtual controller*, the interface both bots and players share to control their humanoid game avatar. The virtual controller enforces a strict separation between AI and the game engine, so that AI code does not directly control animations or guns, and thus ensures that the bots' capabilities are identical to those of the player.

29.4.2 Individual Planner

We use a custom planner that is based on the Hierarchical Task Network (HTN) planner architecture [Nau 00]. This planner was chosen because it provides good control over what plans can be generated and a clear definition of priorities between plans.

The individuals' behavior is defined in HTN by a *domain*, which consists of *constants* and *methods*. A method defines one or more ways to achieve a *task* by defining a list of *branches*. Each branch has a *precondition* and a *decomposition* consisting of a list of tasks. A task can be *primitive*, meaning it can be executed by the agent, or *compound*, meaning it needs to be further decomposed using a method.

Planning begins by instantiating a plan for a top-level task. The branches of the method for this task are attempted in listed order. When a branch's precondition matches, the planner recursively tries to create plans for all compound tasks in the branch's plan. The planner will backtrack over branches, variable bindings of preconditions, and plan instantiations for compound tasks. The resulting plan is the first one fully consisting of primitive tasks.

The primitive tasks modify their agent's memory, make the bot do things in the world, and add information for debugging. Table 29.1 shows some of the primitive tasks.

Listing 29.1 shows the method that turrets (which use the same planner as the bots) use for selecting a weapon to fire at some threat (represented by the variable `?threat`). Variables are prefixed by a question mark, constants by an @ sign, and the keyword `call` precedes C++ function calls. Each of the weapons of the turret has its own branch, with preconditions matching against facts in the agent's world database (e.g., `distance_to_threat`), resulting in the instantiation of local variables (e.g., `?dist`). The plan of both branches

Table 29.1 Example of primitive tasks (prefixed with !) for individual bots

Primitive task	Description
`!remember`	Add a (temporary) fact to agent database.
`!forget`	Remove fact(s) from agent database.
`!fire_weapon_at_entity`	Fire current weapon at specified entity.
`!reload_weapon`	Reload current weapon.
`!use_item_on_entity`	Use current inventory item on other entity.
`!broadcast`	Send a message to nearby bots.
`!log_color`	Add text to the debug log of an agent.

Listing 29.1. Example methods for selecting a weapon to fire. The syntax is based on the SHOP HTN planner, which was written in Lisp. Our HTN compiler converts this domain description into C++ code.

```
(:method (attack ?threat)
    (:branch "use bullets"
        (and (distance_to_threat ?threat ?dist)
            (call le ?dist @bullet_rng)
            (call request_line_of_attack ?threat bullets)
            (line_of_attack ?threat bullets) )
        (   (!begin_plan attack_using_bullets)
            (select_weapon bullets)
            (!fire_weapon ?threat)
            (!end_plan) )
    )
    (:branch "use missiles"
        (and (distance_to_threat ?threat ?dist)
            (call ge ?dist @bullet_rng)
            (call le ?dist @missile_rng)
            (call request_line_of_attack ?threat missiles)
            (line_of_attack ?threat missiles) )
        (   (!begin_plan attack_using_missiles)
            (select_weapon missiles)
            (!fire_weapon ?threat)
            (!end_plan) )
    )
)
```

contains a number of primitive tasks and one compound task (`select_weapon`). To form an instantiated plan that compound method needs to be decomposed further by another method.

The example in Listing 29.1 also shows how a request for information (`request_line_of_attack`) can be part of the precondition; the result will be placed in the database and is tested in the next condition (`line_of_attack`).

For bots, plan generation always starts with a root task (called `behave`). The methods in the domain determine the plans that can be generated and in which order they are tried. Listing 29.2 shows the final decomposition that led to a medic bot's plan to use the revive

Listing 29.2. An example decomposition tree for the HTN domain. "+" denotes a successful branch choice, "-" a branch that was considered but failed. Variable bindings are in italics. Not all successful decompositions are shown expanded.

```
behave
+ branch_mp_behave
    - (do_behave_in_vehicle_mp)
    + (do_behave_on_foot_mp)
        - branch_self_preservation
        + branch_medic_revive
            + (do_medic_revive)
                - branch_medic_revive_abort
                - branch_medic_revive_continue
                + branch_medic_revive
                    (!begin_plan medic_revive [Soldier:TimV])
                    (!log_color magenta "Medic reviving nearby entity.")
                    (!broadcast 30 10 medic_revives [Soldier:TimV])
                    (!select_target [Soldier:TimV])
                    + (walk_to_attack 5416 crouching auto)
                    + (wield_weapon wp_online_mp_bot_revive_gun)
                        - branch_dont_switch_weapon
                        + branch_switch_weapon
                            (?wp = wp_online_mp_bot_revive_gun)
                            + (wield_weapon_internal wp_mp_bot_revive_gun)
                        (!use_item_on_entity [Soldier:TimV] crouching)
                        (!end_plan)
```

tool on a teammate. The decomposition illustrates the way the HTN planner solves a plan, but also illustrates the way the domain is structured.

The final plan consists of all the primitive tasks in the decomposition. Starting at `behave`, the first branches decide between behavior in a vehicle or on foot, followed by a choice between self-preservation behavior (such as fleeing a grenade) and doing one of its class abilities, `revive`.

As described above, planning starts at the root node, generating a plan for `behave`. It does this by going through `behave`'s branches in the order specified, so that branches pertaining to self-preservation are considered before those for healing friendlies for instance. If it selects a branch that contains composite tasks, the planner recursively generates a plan for that branch.

Once the planner has finished, the agent can execute the plan. Each update the current task in the plan is executed. The current task can continue running for multiple updates before succeeding and moving to the next task in the plan, or fail. If the plan has reached the end or a task has failed the agent needs to generate a new plan.

In practice, making sure the agents' plan is still the best one given the ever-changing situation is more complicated. Because of this, the agent reruns the planner at a fixed rate and replaces the current plan if it finds one which is better (that is, one that traverses branches that are farther up the list than those in the current plan). For example, if the AI is in the process of healing a buddy when it replans, and it discovers that there is a grenade nearby, it will abort the heal in favor of a self-preservation branch (such as fleeing from the grenade).

Plans will also be interrupted if they are no longer relevant. For example, if the AI is running toward a buddy that it wants to heal, but that buddy has since died, then it is time to select a different plan. This is implemented by adding extra branches to the domain that contains the "continuation conditions." These branches are selected when a plan is active and the plan's continuation conditions are met. They contain a single `continue` task, which lets the planner know that further planning is unnecessary, and the current plan is still the best.

Within this general planning and reasoning framework the domains need specific information to make good tactical decisions. The next section briefly describes the types of information that are available, and how they are used.

29.4.2 Individual Tactical Reasoning—Waypoint Graph and Cover Data

The basis for the combat reasoning of our single and multiplayer bots is a combination of *waypoints* and *cover data*. The *waypoint graph* defines waypoints, positions for which tactical data is available, and links between them that allow the AI to plan paths and navigate. This waypoint graph is generated by our automated tools (described in [Mononen 11]). For each waypoint, cover data is stored that describes the cover available in each direction. The cover data is automatically created in an offline process.

This data, in combination with dynamic information on threats, is used for various tactical decisions. For instance, selecting a suitable position to attack a threat is done by *position picking*, where the waypoint graph is used to generate nearby potential waypoints and score these candidates based on a number of properties. Properties can include cover from known threats, distance from threats or friendlies, line of fire to threats, travel distance to reach the position, etc. The properties that are used and how they influence the score can be specified differently for different behaviors. Another use of this data is *threat prediction*, where the most likely hiding waypoints for a hidden threat are calculated. Our agents' tactical reasoning has been discussed in more details in our previous work [Straatman 06, Van der Leeuw 09].

These tactical services are available to the domains in a number of ways. Position picking or path planning queries can be used in preconditions of branches and will instantiate variables with the best position or path (if any). Predicted positions of threats can be queried in preconditions of branches and plans can use the resulting lists of hiding waypoints in a variety of ways—for example, to search or scan for the threat. In this way we combine a more specific, optimized tactical problem solver with the general HTN planner to generate tactical behavior.

29.5 Squad AI

The previous section described how our autonomous bots make decisions on combat and ability use. We now turn to squads, which will make these bots work together as a group to achieve a goal. A *squad* is an agent that controls a collection of bots. The squad AI structure is similar to an individual's: it collects information into a world state, generates and monitors a plan using the HTN planner, and executes the tasks in that plan. The difference lies in the data collected, domains, and primitive actions.

29.5.1 The Squad Update Loop and Planning Domain

During data collection, the squad AI gathers information on the state of its members. Instead of using perception to collect this information, however the squad bases its world

Table 29.2 Primitive tasks (prefixed with !) for squads

Primitive task	Description
`!start_command_sequence`	Start a sequence of commands to send to an agent
`!order`	Send command to agent's queue
`!end_command_sequence`	End a command sequence
`!clear_order`	Pop current command from own command queue

state on messages received from its members. Based on this state and the orders given to it by the commander AI, the squad planner generates a plan. The squad does not act directly, but through its members. As a result, most of its primitive tasks simply send an order to a member's *command queue*. Table 29.2 shows some of these primitive tasks.

Each squad and individual has a command queue (similar to RTS units). Newly arriving orders overwrite existing orders, unless they are part of a sequence, in which case they are queued. The individual's domain will try to handle the current command and removes it from the queue when the order is completed.

Listing 29.3 shows a part of the squad method for making one member defend an area. The branch `advance` takes care of the case where the squad needs to move to the area that must be defended. The plan starts by resetting the squad's bookkeeping on what the member is doing, and then commands the member to stay within the areas the squad pathfinder defines (discussed below), orders the member to move to defend, orders the member to send a message to the squad (so the squad knows it arrived), and on arrival orders the member to stay within the defending area.

Listing 29.3. Part of a squad method showing one branch for defending an area.

```
(:method (order_member_defend ?inp_mbr ?inp_id ?inp_level ?inp_marker
?inp_context_hint)
    …
    (:branch "advance"
        ()//no preconditions
        (   (!forget member_status ?mbr **)
            (!remember - member_status ?inp_mbr go_defend ?inp_id)
            (!start_command_sequence ?inp_mbr ?inp_level 1)
            (do_announce_destination_waypoint_to_member ?inp_mbr)
            (!order ?inp_mbr clear_area_filter)
            (!order ?inp_mbr set_area_restrictions
                (call find_areas_to_wp ?inp_mbr
                (call get_entity_wp ?inp_marker)))
            (!order ?inp_mbr move_to_defend ?inp_marker)
            (!order ?inp_mbr send_message completed_advance ?inp_id)
            (set_defend_area_restriction ?inp_mbr
                (call get_entity_area ?inp_marker))
            (!order ?inp_mbr defend_marker ?inp_marker)
            (!end_command_sequence ?inp_mbr)
        )
    )
)
```

As stated before, when the individual bot AI has `DefendMarker` as its current order its planner can make different plans to achieve this. The branches of the method that deals with this order specify both character class specific plans and generally applicable plans. The engineer specific branch specifies: "move there, place a turret nearby," the tactician specific branch specifies "move there, call in a sentry drone," and the generic branch just specifies "move there, scan around." The generic branch is always available, so after an engineer placed his turret he can generate a plan to scan around.

Similar to the individual bots, the squad planner needs information about static and dynamic aspects of the world to make tactical plans. The next section will describe the tactical reasoning available to squads.

29.5.2 Squad Tactical Reasoning—Strategic Graph and Influence Map

The squad AI reasons about the terrain using the *strategic graph*. This is a hierarchical summary of the waypoint graph and consists of *areas* (groups of waypoints) and connections between areas. A connection between two areas exists when there is a link between the waypoints of the two areas in the waypoint graph. This ensures that when a path exists between areas in the strategic graph, it also exists in the waypoint graph, and vice versa. The abstraction of detail the strategic graph provides makes squad reasoning more efficient. This is necessary because the squad's plans typically cover larger areas, and sometimes even the entire map, which would be prohibitively expensive if done on the waypoint graph. Since we do not want to micromanage the individual bots, reasoning at the area level also leaves choices for individual bots, when doing their tactical reasoning at the waypoint level.

The strategic graph is automatically generated from the waypoint graph at export time. The clustering algorithm incrementally groups areas together, starting at one area per waypoint. Clustering is done based on connection properties, number of waypoints in an area, and area surface. This process leads to logical areas that have good pathfinding properties. Similar clustering algorithms have been described elsewhere [van der Sterren 08].

An *influence map* provides dynamic information to complement the strategic graph. Each of the factions updates its own influence map. The map assigns an influence float value to each area expressing whether the area is under enemy or friendly control. Influence maps are a standard technique in strategy games and have been documented in detail [Tozour 01]. The influence map is calculated by counting the number of friendly and enemy bots and players in an area as well as automated turrets and drones. Recent enemy and friendly deaths also change the influence values. Next the values are smoothed based on distance and combined with the previous values in the influence map for temporal smoothing.

Each squad has a strategic pathfinder. Using the influence values in strategic pathfinding allows the squads to avoid enemy controlled areas. Another use is the selection of *regroup markers*, which are safe locations near the objective where squads will gather an attack. Choosing between regroup markers is done by taking the influence map values of the area containing each marker into account. Additionally a penalty is given to areas chosen by any friendly squad as part of their paths, which provides variation for repeated attacks and spreads out simultaneous attacks.

The pathfinder is implemented as a single source pathfinder which calculates the cost towards all areas in the level using the position of the squad as source. Figure 29.3 illustrates the results of this pathfinder.

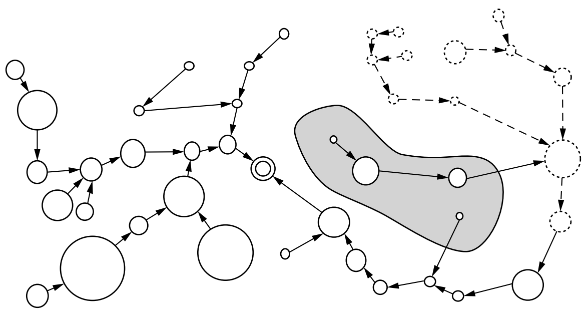

Figure 29.3

The spanning tree of the squad pathfinder in a section of the Salamun Market map. The ultimate destination of the squad is the double circle. The left section of the figure shows a spanning tree that follows the shortest Euclidean path. The shaded dotted zone in the middle right is marked as high cost for the squad, so the dotted nodes on the right avoid that area due to the pathfinder taking into account the costs from the influence map.

The approach allows the squad HTN planner to do many position and path queries as it plans. The squad pathfinder returns a *strategic path* (a sequence of areas) which is then used as a corridor for the squad members by restricting them to these areas. These restrictions constrain the portion of the waypoint graph available for individual path planning and position picking, which allows the individual bots to make their own decisions while still taking the more global tactical decisions of the squad into account. Furthermore the corridor restricts the search space of individual path planning, thus improving performance.

The strategic path planners are updated incrementally using an anytime algorithm to minimize performance impact [Champandard 02]. The incremental nature is less of an issue at the squad level as updates are less frequently needed.

29.6 Commander AI

The highest layer in our AI system is the commander AI for each of the two factions (Helghast and ISA). The commander understands how to play the game modes, and will assign objectives and bots to squads as appropriate for the particular mode being played. Because of the varying number of bots in a faction, the multiple objectives in the game modes, and aspects such as tactical spawn points (TSPs) and vehicles, the commander must be able to create new variants of strategies; a fixed policy for a game mode will not do.

29.6.1 Commander Objectives and Assignment

The commander consists of three parts:

1. A game mode specific part that generates objectives
2. A system for assigning objectives and bots to squads
3. A system for monitoring the objectives assigned to squads

The commander can create squads, delete squads, reuse previous squads, assign a bot to a squad, reassign a bot, and assign objectives to squads and to individual bots. There are four types of objectives: `AdvanceToWaypoint`, `DefendMarker`, `AttackEntity`, and `EscortEntity`. Each objective has a weight, which expresses its importance relative to the other objectives, and an optimal number of bots.

The mission specific AI consists of C++ classes, one for each mission type. These classes use the current information from the game state to add and remove objectives. For example, the class for the Capture and Hold mode needs objectives for attacking or defending the three conquerable areas. Initial objectives will be created to capture all three areas, but if a faction is ahead in score and owns two of the three areas, it can decide to only generate defend objectives for the areas it has already captured.

For Search and Destroy, objectives are added for each of the two destructible targets. The type of objective depends on whether the commander's faction is the defending or attacking side.

In Search and Retrieve there is one object that both factions want to return to their base. When the object has not been claimed, an objective will be generated to advance on the location of the object. If it is claimed, the holding faction will create an escort objective for the carrier, and the other faction will create an objective to attack the carrier to reclaim the object. Both factions may decide to add objectives to defend the enemy's base to prevent returning the object, or to attack their own base to make it easier for the (future) carrier to return.

Some objectives are relevant for all game modes. The tactical spawn points are important because of the shorter travel distances to objectives for re-spawning team members, so capturing or defending them is also a lower priority objective. Furthermore, one level contains vehicles (called Exo's), so there is a *Harass with vehicle* objective that will make a bot in a vehicle attack enemies near the most important objective.

The commanders are custom coded based on input from the multiplayer designers and QA testers. By specifying a list of current objectives, one can make the commanders for a game mode as simple or subtle as needed while the code for the commander stays small.

The algorithm for assigning bots and squads to objectives is as follows:

1. Calculate the ideal distribution of bots, then squads.
2. Create new squads if necessary.
3. If the previous assignments had too many squads assigned to any one objective then remove the excess squads.
4. Assign the best objective to each squad.
5. If too many bots are assigned to a particular squad or objective then unassign them.
6. Assign each free bot to the best squad.

In step 1, all active objectives' weights and desired number of bots are used to calculate the desired number of squads and their sizes. Step 2 and 3 ensure that the actual number of squads matches the desired number as closely as possible. Step 4 prefers to keep squads assigned to previously assigned objectives and otherwise assigns them based on the distance between the squad and the objective. Squads reassigned to new objectives may have too many bots, so step 5 makes these extra bots leave their squad. Step 6 takes care of assigning all bots without a squad to the closest squad in need of more bots. Bots can be assigned to a squad when they are about to re-spawn and then will select the spawn point with the best travel cost (including influence map cost) to their squad. Class selection on spawning by bots is done by a fixed heuristic. This ensures that there are some tacticians and engineers because of their importance for achieving objectives, and randomly assigns a class to the rest of the bots.

29.6.2 Commander Strategic Data

The Commander AI uses a number of *level annotations* to make better map-specific strategic decisions. We chose to manually place these, because they would require complex terrain reasoning to generate automatically but can be identified easily for each map by observing play tests. Some of the annotations are also used by the squad and individual AI at the lower layers when formulating their plans.

Generic level annotations include:

- *Regroup locations*, which are strategic locations that are good to control. Squads are sent here to form up before attacks.
- *Sniping locations*, which specify areas that have good visibility over key locations. Squads are sent here to help defend or attack objectives.

Mission specific annotations include:

- *Assassination hiding locations*. These are markers that specify good locations for the target bot to hide during Assassination missions. Good locations are typically inside defensible buildings. The commander for the defending team initially picks one of these locations, and may decide to move the target to another one based on distance and the influence map later.
- *Defend locations*. Missions with static objectives, such as Capture and Hold, use these locations to define where the defenders should stand or patrol.

Our previous work includes more details on the use of annotations [Straatman 09], as well as on the architecture itself [Verweij 06].

29.7 The AI in Action

As an example, in a Capture and Hold mission the ISA commander might send a squad to defend a capture point, another squad to take the nearby Tactical Spawn point (TSP), and a third squad (which consists of a bot manning an Exo vehicle) to harass any enemies in the areas near the capture point. The squad attacking the TSP plans a path from the base, avoiding another capture point that is owned by the Helghast. Once it has arrived, the tactician captures the TSP and the engineer places a turret on a defend marker near the TSP. After capturing the TSP, the tactician and engineer will patrol and scan the area to defend it. Meanwhile, the squad defending the capture location has come under attack. In response, the tactician calls in drones to help with the defense. Individual bots use the cover data to find cover and attack locations. ISA medics use their revive tool to revive wounded comrades. Engineers repair the supporting Exo when needed. ISA defenders that did not survive will spawn at the nearby TSP to get back to defending the capture point. The overall result is that, by using the architecture described in the sections above, the bots master all gameplay aspects of the *Warzone* mode.

29.8 Future Work

The process of shipping a title always brings new lessons and ideas. Based on what we learned creating *Killzone 3* we are considering the following improvements:

- The squad assignment component of the Commander AI was much harder to get working than it was for *Killzone 2*. This was probably because in *Killzone 3* there are more objectives to consider, but fewer bots to accomplish them with. Therefore we created an agent-based Commander AI that uses the same HTN planner. We expressed our current capture and hold commander and squad assignment in an HTN-based commander and would consider using this approach in future titles.
- It is relatively easy to express different strategies for a game mode using either the objective system or the HTN commander described above. Interviews with our internal testers, game designers, and watching games in public beta will lead to many possible strategies. However, deciding between strategies is harder. We have used reinforcement learning to adapt branch ordering and preconditions in the commanders' HTN domain based on the outcomes of bot versus bot games.

Both these changes are described in more detail in a paper by Folkert Huizinga [Huizinga 11].

29.9 Conclusion

This article described the AI systems we use for our multiplayer bots. We have shown the architecture, algorithms, the way we structure the planning domains, and the extra data (both static and dynamic) involved in making decisions. This approach divides the responsibility for decisions in a hierarchical manner that reduces complexity and maximizes opportunities for adding interesting dynamic behavior.

The use of hierarchical layers is one way we reduce complexity and maximize reuse. Within one agent we often combine static terrain data, a generic problem solver, and dynamic game state data to achieve a nice combination of predictability and reactiveness. Between hierarchical layers the translation from commands to behavior introduces choice and character-specific ways of achieving a goal.

We use various existing techniques in a complementary way. In some places we automatically generate tactical data whereas in others we rely on designer-provided input. We use a generic domain-independent HTN planner to define most of our behavior, but use specific problem solvers for position picking, squad path planning, and lower level humanoid skill planning.

We believe this section shows a way in which various established techniques can be used for combined effect, and that the architecture described here can be applied to many AI engines.

Acknowledgments

The work described here is the result of development over a number of projects and collaboration with numerous other developers. We want to thank the developers at Guerrilla, especially Arjen Beij and the multiplayer design team, for their contributions, feedback, and support. Furthermore, we thank Guerrilla and Sony Computer Entertainment for providing us with the environment that led to the development of the techniques described above.

References

[Champandard 02] A. J. Champandard. "Realistic Autonomous Navigation in Dynamic Environments." Masters Research Thesis, University of Edinburgh, 2002.

[Huizinga 11] F. Huizinga. "Machine Learning Strategic Game Play for a First-Person Shooter Video Game." Masters Research Thesis, Universiteit van Amsterdam, 2011. Available online: (http://www.guerrilla-games.com/publications/index.html#huizinga1106).

[Mononen 11] M. Mononen. "Automatic Annotations in Killzone 3 and Beyond." Paris Game/AI Conference 2011, Paris, June 2011. Available online: (http://www.guerrilla-games.com/publications/index.html#mononen1106).

[Nau 00] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. "SHOP and M-SHOP: Planning with Ordered Task Decomposition." Tech. Report CS TR 4157, University of Maryland, College Park, MD, June, 2000.

[Puig 08] F. Puig Placeres. "Generic perception system." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2008, pp. 285–294.

[Straatman 06] R. Straatman, A. Beij, and W. van der Sterren. "Dynamic tactical position evaluation." In *AI Game Programming Wisdom 3,* edited by Steve Rabin. Hingham, MA: Charles River Media, 2006, pp. 389–403.

[Straatman 09] R. Straatman, T. J. Verweij, and A. Champandard. "Killzone 2 multiplayer bots." *Paris Game/AI Conference 2011*, Paris, June 2011. Available online (http://www.guerrilla-games.com/publications/index.html#straatman0906).

[Tozour 01] P. Tozour. "Influence mapping." In *Game Programming Gems 2*, edited by Mark Deloura. Hingham, MA: Charles River Media, 2001, pp. 287–297.

[Van der Leeuw 09] M. van der Leeuw. "The PlayStation®3's SPUs in the Real World: A KILLZONE 2 Case Study" Presentation GDC 2009, San Francisco, March 2009. Available online (http://www.guerrilla-games.com/publications/index.html#vanderleeuw0903).

[van der Sterren 08] W. van der Sterren. "Automated Terrain Analysis and Area Generation Algorithms."http://aigamedev.com/premium/masterclass/automated-terrain-analysis/

[Verweij 06] T. J. Verweij. "A Hierarchically-Layered Multiplayer Bot System for a First-Person Shooter." Masters Research Thesis, Vrije Universiteit Amsterdam, 2006. Available online: (http://www.guerrilla-games.com/publications/index.html#verweij0708).