Gearing the Tactics Genre: Simultaneous AI Actions in Gears Tactics

Matthias Siemonsmeier

Introduction

Turn-based tactical games had largely disappeared from the mass market by the late 1990s but celebrated a large revival in 2012 with the *X-COM* series reboot, *XCOM: Enemy Unknown*. Today there are countless games across all platforms for turn-based tactic lovers [1].

In comparison, the first *Gears of War* game was released in 2006 and is widely seen as the progenitor of third-person, cover-based, cooperative shooter games of the new console generation [2]. The core gameplay consists of tactical use of positioning and equipment in a group of players in real-time with the objective of killing an emerging force of monsters called the *Locust*. As a cooperative game, players need to coordinate their actions to be successful.

Gears Tactics merges these two very different yet tactical experiences into a fast-paced, turn-based, single-player tactics game in the *Gears of War* universe. This article describes the challenges faced from an AI point of view and the resulting system for simultaneous AI actions of independent planning units while preserving tactical clarity for the player.

Motivation

The core gameplay experience of *Gears of War* matches up with modern tactic games in terms of tactical positioning and cover usage. The coordination between the players in the main game matches the usage of multiple units by one player in a tactics game. In *Gears Tactics*, the player needs to combine specific abilities and equipment of their units to succeed against an emerging *Locust* threat.

Early in development of *Gears Tactics* we first had to figure out how to translate the enemies from a real-time shooter to a tactics game, while ensuring we were consistent with fans expectations of those units. While the actual behavior can be different between the different games in the different genres, the feeling that those enemies create in the player need to be the same and their identity needs to come across to the player. Early prototyping led to the implementation of the first two enemies in the game. The Drone would attack the player with its machine gun from cover. We call this a mirror unit, as it has similar capabilities as the player. The other unit was the Wretch, a small and agile melee unit.

Using a similar number of Drones and player units achieved satisfying results; the player was challenged and had to outsmart enemies that could perform similar actions as the player units. In contrast, Wretches were problematic; if used in similar quantities as the Drones, the player would have attacked and killed them from distance immediately, leading to a reduction of decision space for the player. The player did not have to think about the best action as the best action was always to shoot at them as soon as possible. Thus, Wretches barely got in range to perform their melee attack and never actually threatened the success of the player, making them uninteresting to play against. If used in higher quantities, however, the player suddenly had to make decisions in the face of an overwhelming force of enemies, which posed a real challenge to overcome. The player gained huge satisfaction from successfully deciding when to attack, use an ability or run away. The high kill count also matched the expectation of an action game from the *Gears* IP and felt more rewarding to the player.

Make it fast! Make it Gears!

While we had created a fun and engaging challenge for the player, the enemy turns were slow and dull. Each enemy was selected, decided on the best next action, and executed it. This process repeated itself until all action points were spent. Since Wretches usually spawned in large groups far away from the player, this meant on the enemy turn every individual Wretch would move, one at a time, to get closer to the player. In the main series, however, the Wretches are fast and agile and would surround the player in huge numbers if not taken care of swiftly. Clearly this implementation would not achieve the desired effect or match fans' expectations of the series. Worse, the player typically had all the information they needed after the first unit moved because Wretches tend to move in the same direction. This increased the sense of boredom in watching the units move one after the other.

Instead, we made some small changes to the prototype so that all enemies executed their actions at the same time. While the enemy turn was now exciting to watch and led to some unexpected situations, like enemies shooting a friendly unit that is running in front of them, the player often now missed crucial information. This helped us appreciate how important tactical clarity is for the player; the player always needs to understand changes to the situation that influence their decision making. Important information does not only need to be shown, it needs to be highlighted. The best enemy behavior can be perceived as unintelligent or random if the player does not see and understand its impact [3].

Finally, if groups of Wretches move at the same time, the enemy turn would be faster and give all the necessary information to the player. Traditional squad implementations, however, would mean that the units would always pursue the same goal and would not split up, which was a problem for the game. *Gears Tactics* has a free movement system without an underlying grid. The levels can have huge open areas, but also narrow indoor areas. If the groups cannot split up the threat of the units would be decreased, as there is simply not enough space around the player to attack in all circumstances. Also, we were planning on many different enemy types of different sizes and strengths. Do we want to only allow enemies of the same type to act together? Would this restrict us in level design and scripting?

Tactical Clarity In Gears Tactics

We considered variants of basic flocking algorithms, follow the leader improvements and dynamically splitting the groups. None of these solutions gave us the satisfying results we were looking for, so we went back to the drawing board. We started to think about the game as a whole and the role the AI would play in it. We went from the initial goal of *"make it fast!"* to an elaborate goal statement:

Fast-paced enemy turns that pose a challenge to the player, without losing tactical clarity in a game with a high enemy count, a free movement system, live bullets, and semi-procedural generated maps.

In *Gears Tactics*, each unit can spend their action points in any order they like. There are no distinct movement and action phases. This complicates things further, as it expands the possibility space. It should be noted that different enemy types use different restrictive rules in *Gears Tactics*, this is dependent on their role and characteristics.

We knew that we wanted each unit to decide for themselves what they needed to do to maintain their own character and desires, and that we wanted units to act simultaneously. Given that, we started to ask how we might predict the behavior of a unit to understand how to group them up. Grouping of actions needs to consider that relevant actions need to be reasonably observable by the player. The conclusion was to pre-plan the whole turn and then group any actions that happened in the same spatial area. By limiting the actions happening at any one time in this way the player would be better able to focus their attention and understand what was happening, while still feeling that the game captured the appropriate speed and feel.

Pulling this all together, we identified three rules of tactical clarity, the evolution of which will be described in more detail in the subsequent sections:

Rules of Tactical Clarity

1. Never attack more than one player unit with direct attacks simultaneously.
2. Try to split the actions of a single unit as little as possible.
3. Highlight actions that apply drastic changes to the game situation for the player (e.g. push back actions) by playing them exclusively.

The AI System

The AI System of *Gears Tactics* follows a layered approach (see Figure 1 for an overview). Each layer takes an existing plan as input and produces a new plan to pass on to the next layer.

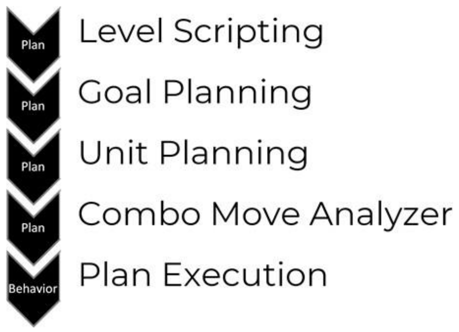

Figure 1: AI System overview.

The first layers are the plan creation layers, including level scripting, goal planning and unit planning. The created plan is passed to the Combo Move Analyzer and then executed in the Plan Execution layer. The following sections will go through the different layers of the system and focus on how decision making was tied into the idea of simultaneous action execution and cooperation between units.

The WorldState

To make pre-planning work, every AI unit needs to understand what changes other units are planning to make. Figure 2 shows two problems. The first is if a unit plans to take a position in the world and the unit planning afterwards is not aware of this, they may try to take the same spot. This will create a conflict when the abilities are executed (see Figure 2a).

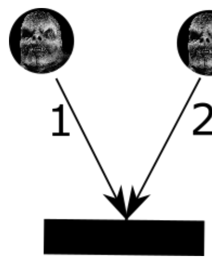

Figure 2a Two units are planning to move to the same spot.

The other problem is that a unit could think it had a clear line of sight to an enemy, but a friendly unit planned to move between the shooter and the target (see Figure 2b).

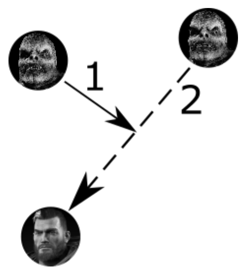

Figure 2b One unit moves in front of another unit that is trying to shoot.

The *WorldState* represents the state of the game after executing an ability. Each ability is simulated in the *WorldState* instead of the game world. The *WorldState* reflects all important information required for decision-making, including health values, locations, and status effects of all units on the battlefield. All planning layers use the *WorldState* as the basis for decision making and modify it according to the newly planned elements.

We knew from the outset that building a strong ability system would help us further down the line. We decided to go for a content-driven approach that was based on nodes. Each ability is a sequence of nodes and each node provides the functionality to simulate its outcome in the *WorldState* instead of modifying the game state. As this is a systemic solution to the problem, designers did not have to take simulation into account when creating new abilities.

All enemy units reason on the current *WorldState* to make their decisions, then after committing to an action the outcome is simulated into the *WorldState* for the next round of decision making. It is important to point out that the simulation happens after the unit commits to an action. Otherwise, it would lead to enemies never taking a shot they would miss, for example, which would be cheating and not very interesting to the player.

Plan Creation

We use three different layers of our AI system for decision making. Each of those layers queue abilities into the plan and modify the *WorldState* accordingly. After a unit has completed each planning layer, it can start executing its plan according to the rules of the Plan Execution layer. It is important to acknowledge that queued actions from all layers are considered in the Combo Move Analyzer layer.

Level Scripting

The scripting layer allows the level scripters to take over an AI unit for one or more turns to queue specific abilities. This is heavily used during the tutorial of the game where the scripters set up specific game situations to guide the player through the mechanics of the game. To make it easy to interact with the AI, specific scripting nodes were created to queue abilities and follow the same principles as in other layers. This gave the required control, like picking the target for a shooting action if necessary, but also provided adequate flexibility. There is no difference between the behavior of scripted and self-planning units visible to the player. This is especially important during the tutorial, as here the player expectation for how the game will work is set for the rest of the game.

Goal Planning: Mission Objectives and Global Tactical Decisions

There are major differences when approaching a tactics game versus a real-time game. In a real-time game, the AI can decide in time intervals if what it is doing is still the correct thing to do. In turn-based games however, the AI needs to commit to a series of actions within one turn and defend this sequence of decisions against the scrutiny of the player. This increases the complexity of the tactical decision-making process further. Not only do we want to select actions that make sense to the AI, but also that the player can understand, deem intelligent and predict over time.

In *Gears Tactics*, the player has several main and side objectives to complete during a mission. The objectives vary from reaching a specific location, interacting with objects in the world such as freeing prisoners from torture pods, to taking over and defending an area from escalating waves of enemies. The AI needs to take all of this into account to present a reasonable challenge to the player.

Mission objectives can be broken down in two different types of AI behavior: attack and defend. There are different unit types more suited for some objectives than others. For example, a sniper unit is better used to keep its distance to a capture circle instead of trying to rush into it, while the opposite is true for fast-paced melee units.

Next to mission objectives, there are also several other global decisions to be taken in a tactics game, such as buffing allies to improve their impact, which should happen before they shoot.

For our tactical decision making, we use a fuzzy logic assignment system. The system uses goal data to describe possible goals in the game the AI needs to reason about. Table 1 shows the setup of the data for such a goal.

Target	DBNO Enemy
Destruction	Target Invalid
Activation	Target CanBePerceived
Deactivation	¬(Target CanBePerceived)
Priority	60.0 - 69.0 (Distance ToFriendly)
Max Score / Max Units	1.0 / 1
Assignable Units → Wretch → Drone	 1.0 (Distance ToTarget) 0.9 (Distance ToTarget)
SubGoals	Empty

Table 1: Simplified overview of goal data for an execution goal. An instance of this goal is created for every down-but-not-out (DBNO) enemy on the battlefield (target) and is destroyed if the target is invalid. The goal is active if the target can be perceived. The priority of this goal is between 60.0 and 69.0, set by the distance of the target to any friendly (from the AIs perspective) unit. The max score and max units determine that only one unit can be assigned to this goal at a time and possible units are Wretches and Drones. Wretches get a higher base score, weighted by the distance of the specific unit to the target. This goal does not have any sub-goals.

The target determines for which objects in the game a goal will be instantiated. In Table 1, one instance of this goal is created for each enemy in the down-but-not-out (DBNO) state.

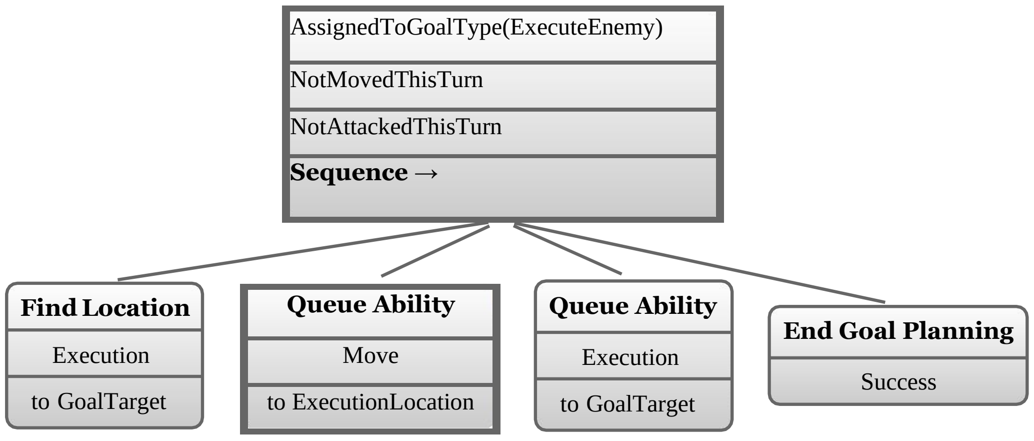

Figure 3: Sub-behavior tree for planning against specific goal assignment type of executing a DBNO enemy. The goal planning is considered successful if the last node is reached, failed otherwise. This tree queues a move to a valid execution location followed by the execution ability itself. Queuing the abilities simulates their outcome into the WorldState. If the sub-tree fails the planning result is returned to the goal planner as *failed* and no actions from this planning are considered. In this case the changes that any *Queue Ability* node has applied to the WorldState needs to be reverted.

The goal also provides rules for when it gets destroyed, in most cases this is when the referenced object is destroyed but can also have other rules based on the game state. The priority rule determines the priority of a goal instance, in this example we define the priority by the distance to enemy units. Other examples include defending the main objective, healing or reviving friendly units (which can be prioritized by distance), progress on the objective (such as how many resources have already been collected), number of friendly units in an area and their health values. It is worth noting that goals impact each other, for example healing a friendly unit increases the troop strength in the area to defend this can have an impact on the defend goal. A capture circle that is not held by any unit has a higher priority than one that is already held by multiple friendly units. The priority rules are fuzzy and interpolate between the minimum and maximum value.

The assignment rules determine how many enemies can be assigned to a specific goal instance and how much the different enemy types count towards this. For example, Wretches should be prioritized over Drones to execute DBNO enemy units. Each of those assignment rules comes with their own set of (fuzzy) conditions.

Once the system has decided which unit is a good candidate for a goal, it assigns a planning job to this unit providing the goal instance. The unit executes a behavior tree for this type of assignment and uses data, like its location, to plan towards it (see the behavior tree in Figure 3). This way the unit is in charge and can decide how to achieve a certain assignment. This can be done by overwatching from distance, for a sniper, or by moving in close range and attacking for a melee unit. The behavior tree reports the result of the assignment back to the goal system. The goal system updates the insistence, the run-time priority, of this goal accordingly. It re-sorts all goals by insistence again and continues the process until all units are assigned to a goal or all goals are fulfilled.

Unit Collaboration

In addition to mission objectives this layer also handles cooperation between units. A typical example is a player unit in cover generating a *push-out-of-cover* goal. Units that have a cover push ability will have priority to do their planning before other units. If a unit is pushed out of cover, the hit chance against it is increased drastically, so other units that are planning afterwards can profit.

Similar to working towards the same goal, the same system is used to decide which unit should take an exclusive goal, like reviving a downed teammate. This decision is more problematic if the downed unit cannot be reached within one turn. Without an overarching tactical decision-making layer, the decision to move towards the downed teammate could be taken by multiple AI units and this could contradict with the overall tactical decision that the player would perceive as natural.

Unit Planning

During goal planning, not all action points of the unit might be spent, or a unit might not be assigned a goal at all. In this case, units have a default behaviour tree to execute to plan their turn. Those behavior trees are built around the desires of the specific unit type and are executed when the unit is selected for unit planning.

After the unit planning layer is complete, we have figured out what to do this turn and pass the information to the combo move analyzer to then be executed.

The Plan: Combo Moves

A combo move is an action that includes more than one unit, such as multiple units moving into the same area or attacking the same player unit. Additionally, there are cooperative combo moves which include one unit applying a buff that other units can utilize afterwards. Combo moves lead to faster enemy turns, take the possible camera framing into account and give needed tactical clarity to the player. While every unit in the game makes its own decisions, the combo moves help give the impression of cooperating with each other. Further voice overs, animations and special cameras also add to this perception for the player.

The enemy turn can be expressed as a series of abilities that may be executed at the same time; this plan can consist of different element types:

- **Ability**: Triple of *(unit, ability, run-time data)* or with different words *(who, what, how)*. The run-time data includes, next to other data, the target of an ability like the target to shoot or a location to move to.
- **Sync**: Marker that is used during plan execution. All actions between two Sync elements are assigned to the units simultaneously. Each unit executes all assigned actions in sequence.
- **Combo**: Bundle of several other elements.

Looking at Figure 4, the plan can be written as follows:

> (A, move); (A, shoot, X); (Sync); (B, move); (B, shoot, Y); (Sync); (C, move); (C, shoot, X); (Sync);

(Note that this is a simplified notation for readability reasons.)

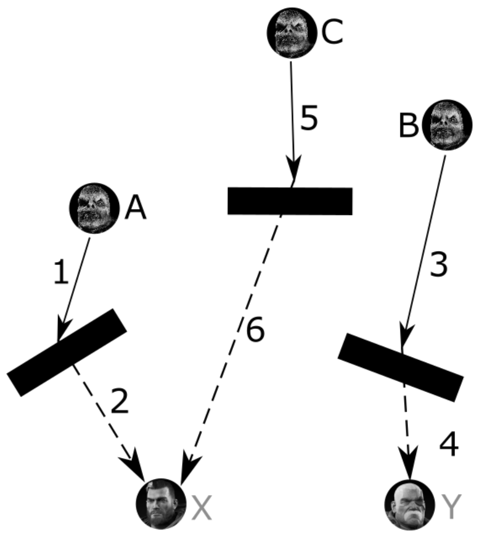

Figure 4: Example game situation with plan. AI units on top, player units at the bottom. Rectangles represent cover nodes. Arrows correspond to the planned action (dotted for shooting; straight for movement). The numbers correspond to the planning order. In this

example all units plan to move into a cover position and to shoot at the player afterwards. Unit A plans first followed by unit B and then C.

Assuming all units together are frameable by the camera, we could have all three AI units act simultaneously if we removed the Sync elements:

> (A, move); (A, shoot, X); (B, move); (B, shoot, Y); (C, move); (C, shoot, X); (Sync);

This plan would be hard to parse for the player, however, and conflicts with our thoughts on tactical clarity:

> First Rule of Tactical Clarity
>
> 1. Never attack more than one player unit with direct attacks simultaneously.

Note also that multiple player units could receive damage from stray bullets (missed shots) or splash damage (explosions) at the same time.

Taking this into account the goal for this situation is actually to achieve:

> **Combo[(A, move); (A, shoot, X); (C, move); (C, shoot, X)]**; (Sync); (B, move); (B, shoot, Y); (Sync)

The actions of unit B can be played before or after the combo move in this situation. This decision should be taken by the current camera location to minimize camera movement during the enemy turn.

Combo Move Analyzer

The Combo Move Analyzer reorders plan elements and merges them into combo moves. This is a post-planning process and there are two main combo moves to be considered: Simultaneous Attacking and Simultaneous Movement. Next to other combo moves, like showing the same status effect messages simultaneously, this layer also provides a chance to trigger flavor actions that tie into the immersion. This section will discuss the main combo moves and describe the process from the initial plan to a plan that executes actions simultaneously.

Simultaneous Attacking

This combo move involves all units that are attacking the same target. We iterate over the plan and create buckets for attacking abilities based on their target and possible camera framing. Each unit might have preconditions stipulating other abilities to execute before the attack, such as moving into a shoot location. These abilities might also be part of other combo moves. Furthermore, we need to consider the planning time for a specific action and what the world looked like when planned. For example, a push-back was applied to a player unit which changes its location. If a unit has planned before this event it should also run the attack before this event. This guarantees that line of sight (LoS) considerations are still valid for the attack.

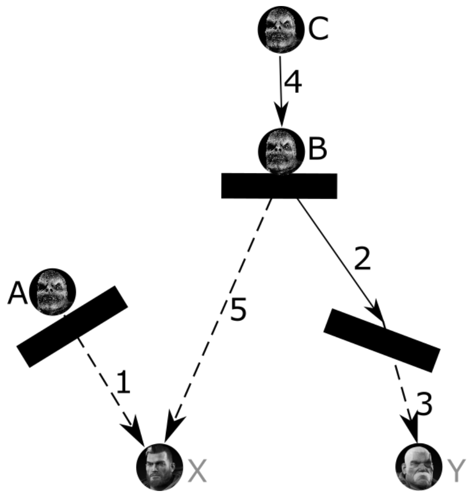

Figure 5: Example game situation. Unit C is planning to take the location that Unit B will leave during execution. Plan can be written as:(A, shoot, X); (Sync); (B, move); (B, shoot, Y); (Sync); (C, move); (C, shoot, X); (Sync)

Figure 5 shows an example of a situation that can be framed by the camera. The initial plan can be modified to let all units attack at the same time. We discovered that this is hard for the player to parse as the target of the actions is different. For reasons of tactical clarity, we decided to not play actions with different targets simultaneously. Unit A and C are attacking the same target and can do this at the same time, so we can combine their actions into one combo move. We need to take into account that C needs to move into position before shooting, so the movement becomes a pre-ability of the combo move. In this case, the movement action can be added to the combo move, as there are no other dependencies. In more complicated scenarios, the pre-ability could already be part of another combo move and this would lead to nested combo moves. One important constraint to maintain is to keep the order of actions for each unit. When to execute them can change, but never the order. The resulting plan for this situation can be written as follows:

Combo[(A, shoot, X); (C, move); (C, shoot, X); (Sync)]; (B, move); (B, shoot, Y); (Sync);

Unfortunately, executing this plan would lead to a conflict. Unit C would try to move into the location currently occupied by Unit B. When unit C planned its actions unit B had already moved away in the WorldState, but because of the reordering, this has not yet happened in the real world. Pre-abilities can also come from other units, in this case, the movement action of unit B to let unit C take the shooting location. The plan can be written as:

(B, move); Combo[(A, shoot, X); (C, move); (C, shoot, X); (Sync)]; (B, shoot, Y); (Sync);

While this plan solves the conflict, we discovered that it breaks the tactical clarity for the player. Splitting the actions of unit B made it harder to parse the situation. The aim should be to split the actions of a single unit as little as possible or keep them as close to each other as possible. This makes the second rule of tactical clarity:

Second Rule of Tactical Clarity

2. Try to split the actions of a single unit as little as possible.

Thus, the final plan for this game situation is as follows:

(B, move); (B, shoot, Y); (Sync); Combo[(A, shoot, X); (C, move); (C, shoot, X); (Sync)];

(Note that though rare, it is not always feasible because of potential cycling dependencies of units in different combo moves. In such instances, it is necessary to allow splitting of a unit's actions.)

Player perception is most important consideration when making the turn fast. We discovered that a player needs to know which unit applied a push back, for example. The impact of this finding is that we need to play some actions exclusively, including some specific buffs, push-back actions or the execution ability. Those actions can still be part of a combo move but will always be surrounded by Sync elements to ensure their exclusive execution. Thus, we arrived at our third rule:

Third Rule of Tactical Clarity

3. Highlight actions that apply drastic changes to the game situation for the player (e.g. push back actions) by playing them exclusively.

Simultaneous Movement

In the same way that we can bundle up the attacks on a specific player, we can also bundle up the movement into an area. All units move at their own pace into a location and the combo move is completed when the last unit moves into position.

Simultaneous movements do not necessarily need to show the starting locations of the units that are moving as the direction is often enough for the player to understand what is happening. As a result, only the final destinations need to be frameable, not the starting locations. The camera should zoom out, focusing for the player on the final destinations and the player will see enemies from all directions moving in. This intensifies the feeling for the player of being surrounded and increases the immersion.

Flavor Actions

After combo moves are created there is a chance to add what we call *flavor actions*. These actions aim mainly to give the player the perception of a living world and do not have any gameplay effect. Usually this is achieved by playing animations and voice overs. A leader unit can point at a location and shout "move" before a group of units move to this location, for example. This behavior is added to combo moves after their creation and is free to execute for the AI.

Plan Execution

The plan execution layer takes a plan as input and produces the behavior that the player can see on the screen. Therefore, the plan execution layer goes through the plan and collects all elements up until the next Sync point and assigns them to the units. A unit executes all assigned plan elements in sequence and reports back when done. The plan execution layer then repeats the process until all elements are executed. When executing actions for multiple units simultaneously, there are also small random delays added to ensure all units do not start at the exact same time.

Looking back at Figure 4 again, the plan execution layer would assign the move and shoot elements to unit A, then wait for it to finish before repeating the process for units B and C.

Interruptions

If something goes wrong during action execution, the system will trigger an interruption. An example of such a situation is shown in Figure 6. An interruption will cause the whole planning process to restart, beginning with the creation of a new *WorldState*. Interruptions can be triggered for multiple reasons, for example when a planned ability fails execution or for special player abilities that modify the world during the enemy turn.

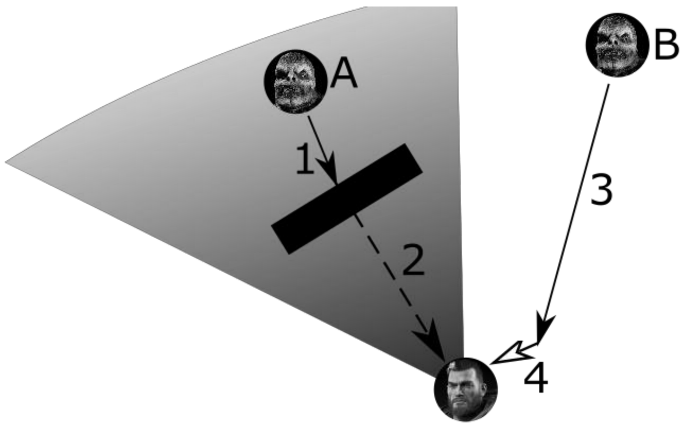

Figure 6: Interruption. Unit A plans to move into the cover location and shoot at the player. The expected outcome is that the target will fall DBNO and unit B plans to move next to it and execute the unit. The player has an active overwatch action running and will shoot at unit A as soon as any action is executed, this might end the action, DBNO or kill the unit or apply any form of buff. The outcome of the actions of unit A might not match the WorldState at the time of planning for unit B. In that case an interruption is fired.

Pre-planning those abilities in the planning phase would be too expensive in an environment with free movement and live bullets. Handling interruptions is run-time expensive and therefore should be kept to a minimum. On the other hand, the impact on planning time needs to be considered.

After an interruption, a unit needs to be aware of what it has done already in the world during the last rounds of plan execution. This is necessary as many units have specific rules like only attacking once per turn. This information is handled through the blackboard during planning. Instead of modifying the blackboard directly from a behavior tree, we use a structure called blackboard modifications that will be reflected in the plan elements. If a plan is interrupted the opposite effect is applied. Consider a unit that is allowed to attack twice during a turn. Each time the shoot-ability is queued in the behavior tree, the blackboard value for the number of shooting actions is increased by one. If an interruption occurs before the ability is executed, the value is decreased by one.

Divide (planning) and conquer (execution)

Free movement requires testing a lot of sample points in the world while the live bullet system requires ray casts from each of those locations since they cannot be precomputed. Combined with a high enemy count and planning in *Gears Tactics* can take some time.

When we showed the enemy turn videos in our team updates, people were excited about what they saw. When people played the game themselves, however, not surprisingly we got reports of unreasonably long enemy turns. Specifically, the problem lay in the planning time before the player saw the enemies act. Our goal was to make the enemy turn more exciting, but it seemed that, despite all the effort we put in, we achieved the opposite.

It's all about Perception

Our implementation of the *WorldState* meant we know what to expect from a plan execution before it happens, allowing us to execute sub-plans as soon as they were available and letting units plan in the background. . This minimized the player downtime when nothing outside of idle animations could happen, but also meant that the first unit will always act on its own. This did not seem like a viable solution, especially taking into account different playstyles of different players. We wanted to preserve the satisfying feeling of killing a whole group of Wretches with one well-placed overwatch.

We started to experiment with different initial planning times before already planned actions are executed. There were three major findings in those experiments:

1. More initial planning time did not always lead to faster overall turns, because planning can already happen while other units are executing their actions.
2. The player reacted positively to simultaneous actions.
3. The player reacted negatively to long downtimes but also appreciated that the AI typically needs some time to reason at the beginning of the turn in tactic games. Downtime during the enemy turn was perceived as slower overall and detrimental to the overall experience.

These results led to the implementation of the following rules:

1. Start executing as soon as there is a push back action in the plan. Elements from before and after this will not be executed at the same time.
2. Define a timeout for the player downtime for the beginning of the turn and a much smaller one for mid-turn after an interruption. We could also hide more planning time at the start of the turn as there is a banner and a sound played to the player which reduces the player downtime on its own.

A few seconds at the start were perceived as acceptable as long as it was a maximum of a couple of seconds during the turn.

Debugging Enemy Turns: Enemy turn stuck!

As discussed, the actions of the enemies are intentionally framed in tactic games for tactical clarity and to allow other processes to finish before enemies can reason about the world. During development, we nonetheless faced endless complaints about “the enemy turn stuck issue is still not fixed”. The reality is that a stuck enemy turn is a symptom, not the disease itself. The reasons for legitimately stuck enemy turns varied from abilities that did not terminate to wrong content that triggered navigation mesh rebuilds every frame.

Such issues can have multiple causes and are expected during the development of a complex game with a big team. Nevertheless, when you encounter a bug your game is over, so the importance of fixing these issues promptly is immense. In all cases, the bugs went first to the AI team to investigate. We spent hours debugging and fixing those issues, but very often we also assigned them to other teams to fix specific (content) problems. Ultimately, we knew that if we wanted to hit our goals for the game, we needed to spend less time investigating these symptoms.

Along the way we learned a valuable lesson. Not all of the issues were easily reproducible, but even if you see the issue only once out of a hundred times you have to take care of it. Scaling up player numbers meant that even a one in hundred chance would translate to a lot of players stuck in enemy turns and whose experience’s will be ruined. Dealing with these bugs at those reproduction rates is tough as a programmer and was complicated even further when we had issues we could only reproduce on specific machines.

In addition to standard tools like a remote debugger, one tool that helped was the Visual Logger implementation that comes with Unreal Engine 4. We made extensive use of this feature and expected the visual log file attached to every bug ticket we received. This file was written automatically in non-shipping builds and allowed us to understand a whole game session after it was played. The timeline allows to move back and forth and inspect each event. Logged events included started abilities, goal assignments, planning results for each unit, but also random seeds, environment query results and behavior tree executions.

Performance Tracking

Performance tracking for a system like ours was not straightforward. The AI module in UE4 is already heavily time-sliced for things like environment queries. On top of that, we had different planning layers triggering several different environment queries at different times. Bad rules in the goal assignment layer would increase the number of failed attempts to plan against them and would lead to even more planning attempts.

To track all of this we used telemetry and an offline analysis tool over multiple games. The main variables we tracked within a turn were planning time for each unit towards a specific goal, planning time for unit planning, failed and successful goal assignments, the number of interruptions, and the player downtime. Those variables were tracked against a specific version of the game, the map and the platform the game was played on. This allowed us to identify bottlenecks or repeating errors. Instead of fixing what we think might go wrong, we had a tool that drove our decisions about what would have the most impact on the player.

As described in the previous paragraph we did an extensive amount of logging and this comes with a cost, too. We compared absolute numbers only in shipping builds and focused on the relationship of numbers in other configurations. For example, planning times in development builds can be very high overall, but the important information is if a unit takes three times as long as another unit or if a specific goal assignment always fails for a unit on a specific map.

Conclusion

Ultimately, *Gears Tactics* was perceived as very action intense. We like to think that this is, next to a lot of other features, down to the choreographed enemy turn.

The previous sections presented a system focused on player perception and entertainment to solve the problem of the high number of enemies while engaging the player in the action. Our solution proved successful in complex environments and delivered the quality needed for a AAA game. In extreme situations the enemy turn time with combo moves was up to 4.5 times faster compared to the same version of the game with combo moves disabled. In other situations, the turn would slow down and show the necessary information to enable the player to make tactical decisions. The system itself is simple to work with and after jumping over the initial hurdles it is easy to create new behaviors and enemies while giving huge flexibility to the designers. The systemic handling of simultaneous actions is utilized automatically with minimal impact during the design of new abilities. Layers can be added or replaced easily which enables multiple ways for decision making to feed into the system.

Many of our learnings can be applied to a variety of turn-based games to make the enemy turns faster and more exciting rather than slow and dull. This approach requires the simulation of abilities and their effects in the world. The effort to achieve this should not be underestimated. Work on the basics first and make sure that the *WorldState* is an integral part of all systems, not just AI. This will help further down the line, I promise!

Acknowledgments

Making the enemy turns fast and exciting required the combined efforts of the whole team and I would like to thank everyone involved, but also their families and loved ones. None of this could have been made a reality without all your passion and dedicated work!

References

[1] F. Brown, “Turn-based tactics won the decade,” 12 2019. https://www.pcgamer.com/uk/ turn-based-tactics-won-the-decade.

[2] B. Ashcraft, “How cover shaped gaming’s last decade,” 01 2010. https://kotaku.com/ how-cover-shaped-gamings-last-decade-5452654.

[3] B. Wetzel and K. Anderson, “What you see is not what you get: Player perception of ai oppo- nents,” *Game AI Pro 3: Collected Wisdom of Game AI Professionals*, pp. 31–47, 2017.