Open-world Enemy AI in *Mafia III*

Jiri Holba and Gael Huber

1 Introduction

Mafia III is a third person action-adventure game set in Vietnam War Era America. The protagonist Lincoln Clay, an orphan and war veteran, fights to take power away from the Italian mob in the city of New Bordeaux, a fictional version of 1968 New Orleans. But this chapter is not about Lincoln, it is about the hundreds of nameless enemies he fights to get to the mob controlling the city.

The open world has plenty to offer; encounters and player interaction abound. Lincoln can be subtle and lure gangsters from safety, or more direct by getting into fire-fights or police chases. These situations occur systemically throughout New Bordeaux while leaving designers the freedom they crave to work both within and outside of the bounds of these systems.

The whole system is made of numerous individual pieces which create the illusion of deadly gangsters and fearless police officers. This chapter is a broad survey of the most important systems that went into creating *Mafia III*'s AI such as perception, position selection and the cover system. While we use a behavior tree as the main decision-making structure, it is these supporting systems that provide the essential information used by the behavior tree to make sensible decisions. For more detail on behavior trees, see (Simpson 14) or (Colledanchise 18).

2 Enemy design

When the player dives into the world of *Mafia III*, they find themselves in a city rich in detail and era-accuracy. The player can hear the great music of the '60s, ladies walk down the street wearing dresses or slim fit pants, low-heeled shoes, and carrying small rectangular bags; men wear suits or just shirts with a tie and suspenders – all to create an immersive recreation of 1968 New Orleans. The enemies in such a realistic setting have to be believable as well. They have to look like, behave, and be as mortal as humans.

Realistic enemies means realistic perception, abilities and properties. They have limited lifespans because when a human enemy gets shot in the head, everybody expects him to die immediately.

We knew from the beginning that creating such enemies would require a lot of iteration, which demands flexible, data-driven systems. The data-driven approach largely eliminates engineering and game compilation time from the iteration cycle through techniques such as in-game configuration reload.

We decided to break the enemy design down into modular parts to improve data sharing and overall system maintainability. NPCs all have an archetype which provides data

on the appearance and behavior of the NPC, in particular the tactics it can use. A tactic defines decision trees and scoring functions used in battle decisions such as position selection, target selection, in-cover behavior, etc.

2.1 Archetype

An archetype is a description of an NPC type, and contains data such as the list of usable tactics, configuration for senses and voice, weapons, reactions to the player, animation set, etc. Each archetype can have multiple variations and supports inheritance of the settings. At the root of the inheritance hierarchy lies a basic configuration for all archetype variants. This base configuration would have some (or all) of its members overridden in each variant.

The triggerman is an example of one such archetype. In the base configuration the triggerman is equipped to throw Molotov cocktails, but in a variation the triggerman might throw grenades instead.

2.2 Tactics

Tactics define how the NPC behaves in combat. The tactic itself is broken down into these basic blocks:

- Position selection – where the enemy should stand or take cover
- Target selection – who to shoot
- Movement style – how to move into selected position
- Cover action – what to do while staying in a cover position
- Open space action – what to do while staying in open space
- Reload behavior – when to reload

Each of these blocks is another data description which lives in a separate file and the tactic references it. One description of the target selection can be used in many tactics. The position selection and target selection are lists of scoring functions for the utility system described later. The movement style, cover action, open space action and reload behavior are authored as decision trees (Colledanchise 18).

3 AI systems

This section describes systems used either as inputs for the behavior layer - perception, archetype definition, position selection or decision trees – or consumers of the output such as shooting. The general overview of the system’s connections is displayed on Figure 1.

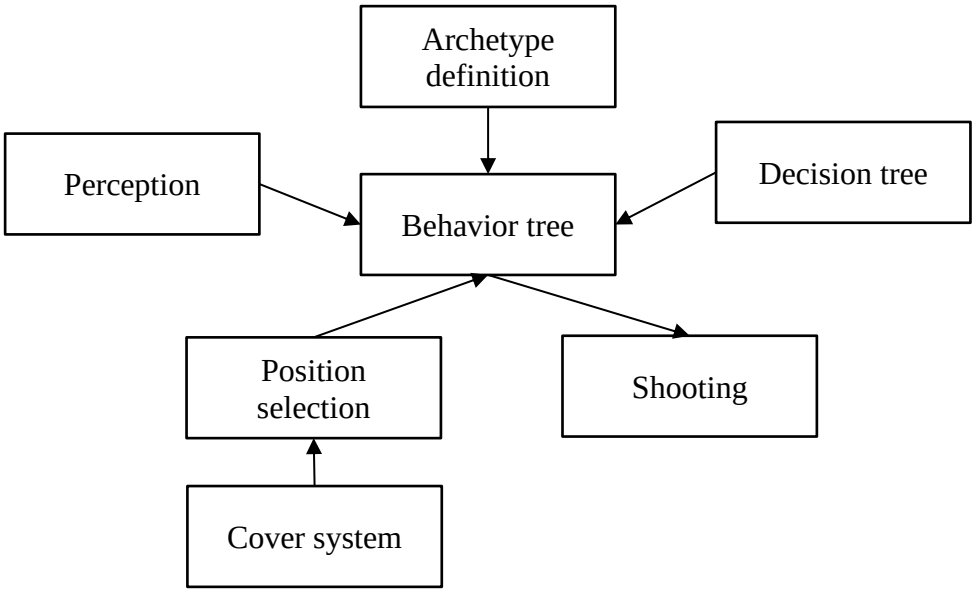

Figure 1 The AI systems are glued together by the behavior tree.

3.1 Perception

What would an NPC be without its senses? Pedestrians, policemen and gangsters all need to be able to see and hear the player when he decides to wreak havoc. The expectation is that we see civilians fleeing in panic, police officers trying to eliminate the threat to public safety and gangsters protecting their turf.

All sight and hearing properties are data driven. They form yet another block of configuration data stored in a separate file referenced by the archetype definition. Designers can change everything from how far away the NPC is able to see, to how far the footsteps sounds travel. Every variable is exposed for editing. If a property is defined as a function, it is authored as a piecewise linear graph.

3.1.1 Sight

Sight in *Mafia III* is used for detection of any event that the AI should react to when there is a clear line of sight between the NPC and the event. Every NPC in the game has a sight detector attached to its head and can detect any sight event within range. The player is constantly emitting the "Player" event which can be detected by enemies. The system is not limited to player detection. It is used, for example, to detect a melee fight in progress by emitting the "melee" event on both participants of the fight. NPCs detect dead bodies, pedestrians react when seeing a weapon in the player's hand, etc.

The core of the sight engine in *Mafia III* uses a vision cone check combined with raycasts to multiple points on the player, similar to systems in other third person action games, namely in the *Tom Clancy's Splinter Cell Blacklist* (Walsh 15).

The *Mafia III* sight engine uses five detection points which can be either data driven, or attached to specific bones. The points are attached to the head, elbows and knees by default, but points are defined by designers for most animation states. Figure 2 shows the

designer authored detection points for the player hiding in a cover position. Using predefined positions instead of animation bones plays better when the player is in stealth mode and it provides better protection during battle.

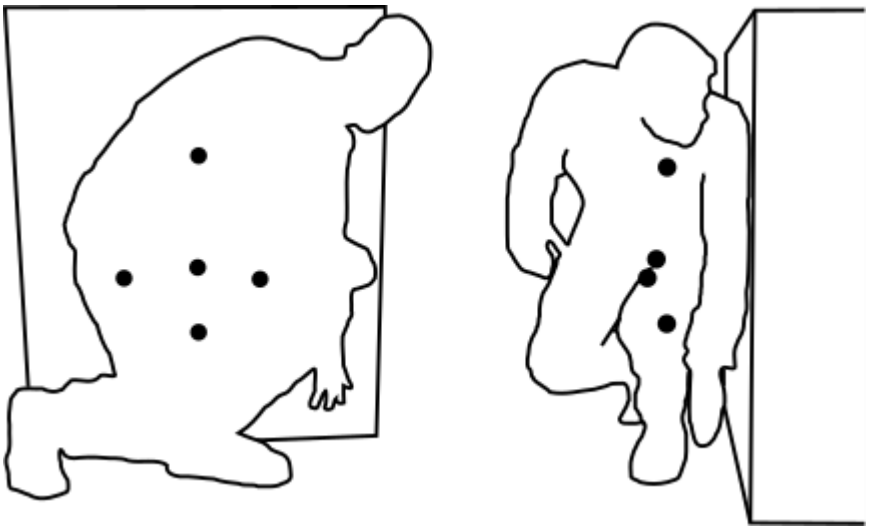

Figure 2 Detection points are moved closer to the chest and collisions to give the player better protection.

When the enemy detects the player, he does not attack immediately. He uses the event as an impulse that something is in his field of view and he starts recognizing it. The speed of recognition is influenced by many factors. We measure the following properties:

- Distance to the event
- Angle between direction to the event and the direction the enemy is looking
- How well the object and NPC are lit
- The object speed
- Number of visible points

The influence of each property is defined in a configuration file as a piecewise linear function and the time needed for a full recognition is given by the Equation 1.

$$t = \frac{1}{\prod_{i=0}^{n} function_i} \quad (1)$$

3.1.2 Hearing

Enemies sometimes need to be able to react to events that they cannot see as well. For that purpose, any sound which is critical for the AI is registered with the AI hearing engine and propagated to all NPCs within range. In *Mafia III* the AI hearing engine is completely independent of the audio engine. Any NPC can have a detector registered with the system.

When the game emits any AI relevant sound, we notify all detectors within range of the event. The propagation distance is different for each sound. For some events, such as the shooting, we react immediately and we switch the NPC into battle behavior. For others, such as footsteps, we react only when we get a certain number of events during a short time span.

3.1.3 Awareness

Once the player is recognized, NPCs can be in one of the five states shown in Figure 3. These states are tied to the enemy behavior. Most situations go through all the states as displayed, but there are exceptions.

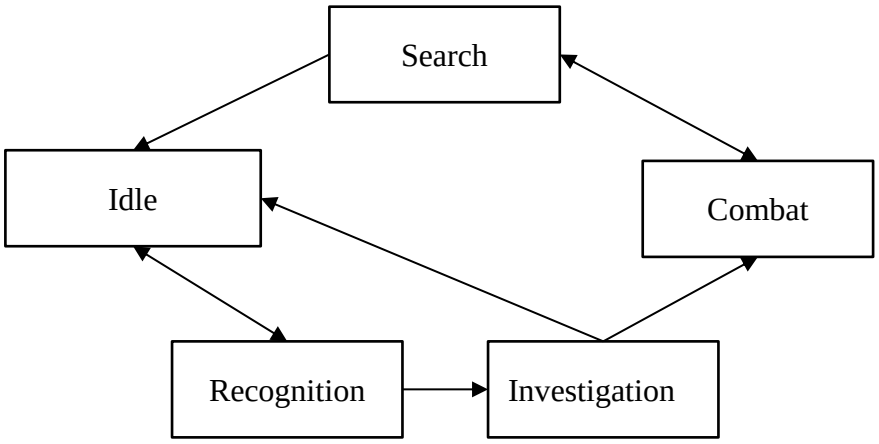

Figure 3 AI uses a state machine for transitioning between recognition states.

Each of these states can use different sight and hearing settings. For example, an unaware enemy will not notice the player as far away as an enemy in active combat.

3.2 Stealth tools

The player has several options when it comes to stealth.

We adopt the common convention that when the player crouches and moves slowly, they are being stealthy and are harder for enemies to detect. The sight detection points on the player are moved lower in the crouch animation and his steps make no sound while he moves slowly. On top of that, when the player hides in a cover position, the speed of recognition is even slower.

During the development of *Mafia III*, the speed of recognition was also slower when the player was sneaking, but we removed this later on because it didn't feel right. A big guy crouching in the middle of the street is not harder to notice than a standing one. The change of detection points position still gives the player better chances to hide behind objects.

To give the player better chances to sneak through the enemy hideout, the game offers different ways to distract the enemies.

- Whistle: Whistle is a powerful ability to distract one of the enemies and lure him out. The crucial part of this ability is the enemy targeting for which we used player driven, screen space targeting. We used a scoring function for choosing the NPC close to the player and close to the center of the screen. Equally important is to have an appropriate reaction for the remaining NPCs in the group we are luring enemies from (Ocio 18).
- Voodoo doll: This consumable item starts emitting sound events when thrown to the ground. Enemies react to this event by approaching to investigate where the sounds are coming from. When the enemy is nearby, the voodoo doll explodes.
- Hit squad: It is a group of four allies helping the player in battle. It is not a stealth distraction per se, but it also does not break the stealth for the player. While the group of friendly AI starts killing the enemies, the player can still sneak around. The enemies are aware of the hit squad, but still unaware of the player. The hit squad members use their senses to detect enemies and they share their knowledge of enemies with each other. Since they need to be able to help the player, they are also automatically aware of all enemies detected by the player who is running a simplified version of the recognition system using a 360 degree field of view. The squad coordination is achieved through the position selection system. When one member of the squad searches for a good position, the system is aware of current positions of all squad members as well as their target positions. This way, we can control how close they stay to each other.

3.3 Cover system

Mafia III is a cover shooter game, so having a good cover system is very important. All cover positions in the game are semi-automatically generated in the game editor using the static world collision geometry. Each cover position can be manually modified to fix situations where the automatic detection fails or the cover position needs to be adjusted for gameplay purposes. Individual cover positions are one meter wide, and they are used in game as positions the AI can choose. Each cover position holds this information:

- Transformation information – Position, rotation and width
- Leaning positions – where the AI can lean and shoot from (left, right, up)
- Height function – this piecewise linear function describes the height profile of the obstacle. It has four control points stored in a 64-bit value. The distance from cover position edge on one axis and height of the obstacle on the other axis. Four points were good enough for describing obstacles which are not flat, such as car hood and roof.
- Protection directions – the full circle around the cover position is split into 32 directions. If there is an obstacle in a given direction, the corresponding bit in a 32-bit value is set to one. We explore just the nearby area around the cover using half a meter long raycasts.
- Left and right neighbor ID – set to non-zero value if the cover position is connected to another one.

Figure 4 shows cover positions generated around a static obstacle, including the directions

the cover position gives protection from and the connections between neighbors. Connections between adjacent cover positions are omitted from the figure for simplicity.

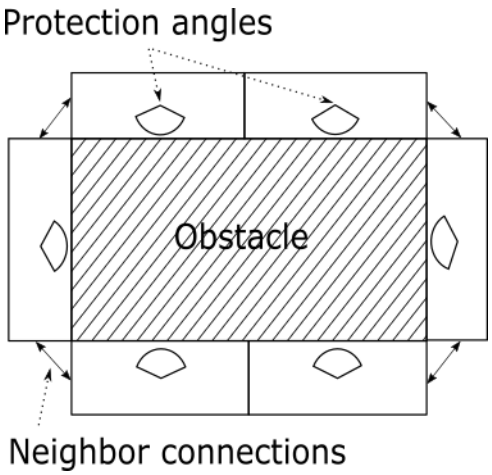

Figure 4 Obstacles have generated cover positions around them.

Dynamic objects, such as cars, crates or garbage containers, also have cover positions attached to them when the object is not moving. As soon as the dynamic object starts moving, or is destroyed, the cover positions are removed from the AI world. Since the dynamic object can move to any location in the game, we need a system which makes sure all cover positions are in a valid state.

When a static or dynamic cover position gets blocked by a dynamic obstacle, we need to disable the blocked part of it. If the remaining part is too small to be a standalone cover position, it is attached to its neighbor if it has any.

When the dynamic object stops moving, the associated cover positions are added into the AI world. All of them need to be snapped to the ground, because using the dynamic object position alone is insufficient. For example, in a situation when a car stops next to a sidewalk, the associated cover position has stored the height of the car. Since the car is on the road and the cover position is on the sidewalk, there is an elevation difference between them. If we do not detect where the ground is, the character would take cover there and his head would be sticking above the car.

When a new dynamic cover position is added into the world, the system will connect it to surrounding static cover positions which then become its neighbors. These connections are used by the AI and the player for free movement between cover positions without leaving them. An example of this is shown in Figure 5. A car stops next to a static obstacle. One of the static cover positions shrinks to fit into the space, while another one is disabled and its neighbor is stretched to fill in the remaining space. Dynamic cover positions around the car are enabled and adjusted as well. All the cover positions are connected to allow NPCs and the player to move along both obstacles seamlessly.

Cover adjustment uses a series of shape casts to determine the unobstructed region of the cover. From the contiguous unobstructed region of up to two neighboring covers, a new

cover slot will be created spanning this region (Figure 6) and the neighbor connections will be fixed up.

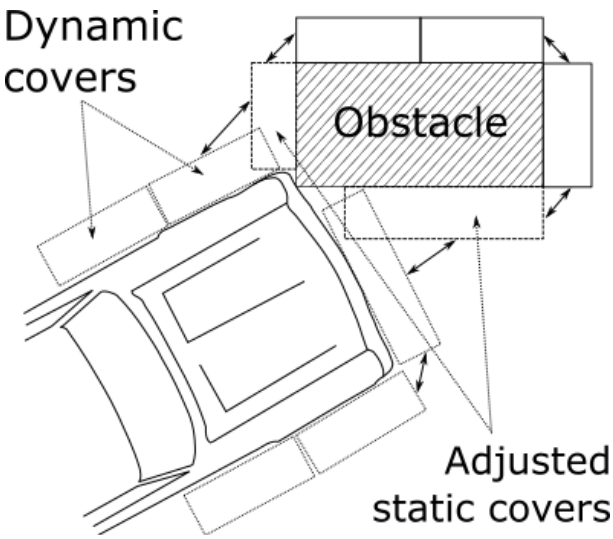

Figure 5 Cover positions blocked by car are disabled and surrounding covers are stretched to cover the missing space.

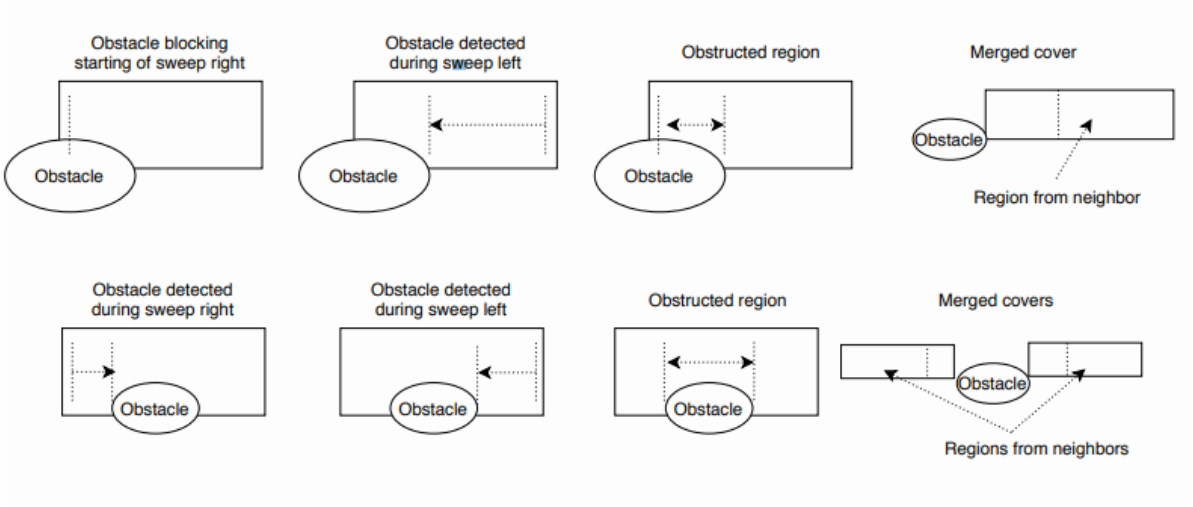

Figure 6 Unobstructed regions of covers are used to merge neighboring cover slots.

3.4 Position selection

Finding the best position in which to hide or stand is a very hard problem to solve. When battle breaks out, the most sensible choices are for the character to either to stand its ground at its current position or to jump behind the closest obstacle. After that, we need to search a wider space for better positions. Position evaluation functions are a common technique for choosing from a flat list of possible positions.

Position evaluation functions use a heuristic function to calculate a score for each of the possible positions. The chosen position is either the highest scoring one, or it is randomly chosen from a subset of high-scoring positions. This technique has been popular in action games for some time (Straatman 05).

We used this technique in our generic scoring system for position selection, target selection and searching positions. Inputs are position/target candidates and a list of scoring functions to apply on each of them. Each scoring function takes some fact about the game and converts that into a single number. For example, distance to current position, line of fire to the player, protection from the player's fire etc. The output is X number of positions with the best score, where the score is calculated using Equation 2.

$$score = baseValue * \prod_{i=0}^{n} \quad function_i \tag{2}$$

We used the product because we wanted to be able to drag the overall value down to zero with any of the functions and this worked the best for us (Lewis 15). However weighted average, mean average, sum etc., are all valid options we considered for combining the function scores.

3.4.1 Limiting the search

In a large world, we first need to know where to search. It would take an unacceptably long time to search the whole map, even though the whole process is time sliced. We limit the search space to a radius around enemies, a radius around the player or a radius around the NPC itself. The search space is given by the currently selected tactic. An enemy in a "support" tactic will be searching for positions around itself, while an enemy in an "assault" tactic which is limited to a search space around his targets as shown in Figure 6.

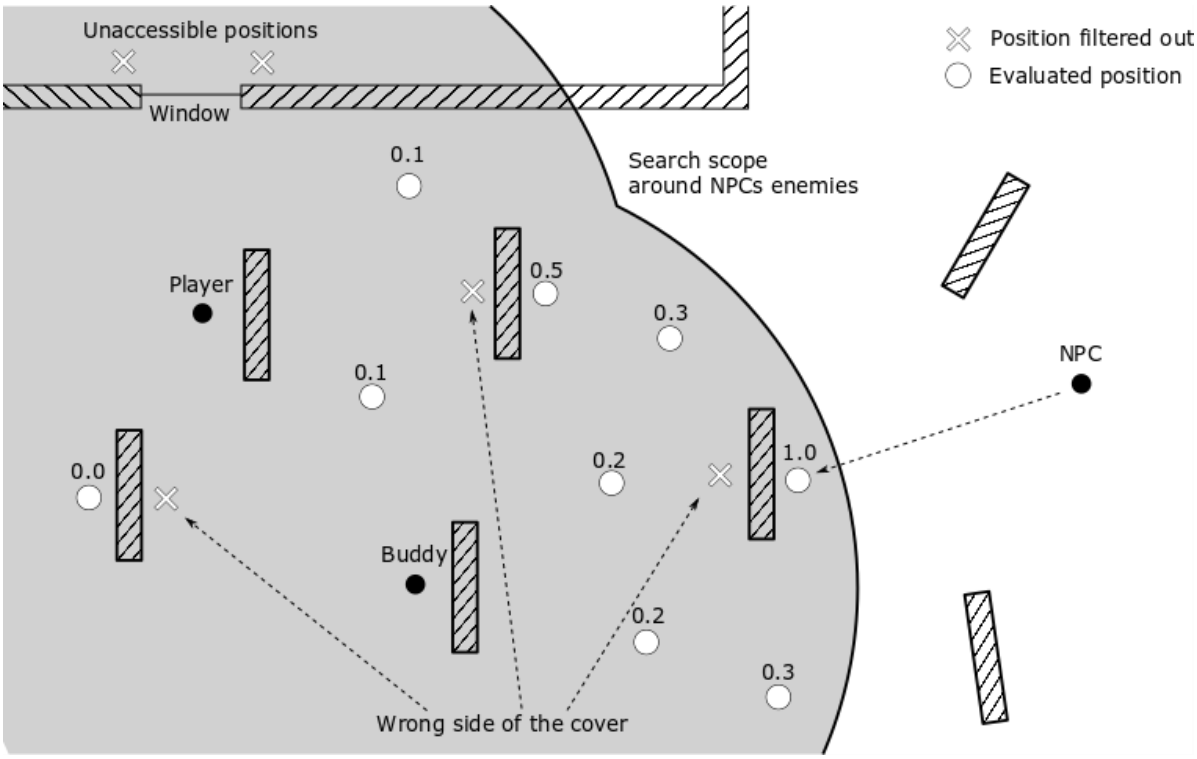

Figure 6 An NPC searches for positions around enemies while assaulting or flanking them. Valid positions are scored and one of the best is selected.

Mafia III uses a Havok navigation mesh for generating open space position candidates in battle and automatically generated cover positions in build time, around both static and dynamic obstacles. There are different ways positions can be generated on the navmesh (Lewis 17). Generating positions in a grid and limiting the search space by approximate path distance worked the best for us.

3.4.2 Scoring functions

Scoring functions in *Mafia III* are usually piecewise linear functions used to normalize the input values. Collecting input values for the scoring functions vary in complexity. It might be as simple as getting the type of the evaluated position, or as complex as the pathing distance from the NPC's current position or a set of raycast checks. Even in the limited search space, there might still be hundreds of position candidates. For example, one of the melee archetypes in the Sign of the Times DLC searches for approximately 120 positions around the player in a six meter radius. A more common case is our base triggerman archetype which searches for all covers plus 120 open space positions in a fifteen meter radius around itself while using the assault tactic. That yields up to 300 positions in complex levels.

We also want to avoid the expensive checks if possible, so at some point in the evaluation process, we compare the scores and choose X of the best positions so far and run the expensive tests, such as raycasts and path searches, only on those. The threshold and

how many positions will proceed to the next phase is all defined in the tactic configuration. In the triggerman example mentioned above, only the fifty top scoring positions are used for raycast checks and only the top twenty positions out of that run the full path search.

Filtering based on partial scores can potentially lead to a situation where we discard the best position just because the score is not good enough at the beginning. While it is not possible to avoid this situation, unless we run all scoring functions on all candidates, it is possible to minimize the issue by careful ordering of the scoring functions and number of positions that we keep in the evaluation.

The development version of the game also had a detection mechanism which, instead of filtering the positions out, would just mark them as "would be removed" and leave them in the selection process. At the end, all positions are scored by all functions and the subset of the best positions is created. If any of the best positions is marked as "would be removed" the game asserts and an engineer or designer can debug why that happened.

3.4.3 Dealing with game dynamics

The state of the world in an action game is changing all the time and characters in the game should react to changes sooner rather than later. When an NPC selects a position from which it can shoot at the player, it needs to get there first. But the player can be long gone before the NPC reaches it.

The simplest solution is to keep running the position evaluation while the NPC is moving into the position while giving a bonus to the currently selected position. It is also a very expensive solution, so we use it very rarely. Another solution is to keep calculating the score only for the selected position and discard the position if the score drops too much. The trick is to use a different set of scoring functions for this purpose. Table 1 shows an example set of scoring functions used in the initial search for a position and set of functions used for consecutive score updates of the selected position. In this example, once we select a position and we start moving into it, the path distance becomes irrelevant.

Table 1 Difference between sets of scoring functions used in full search and selected position update.

Scoring function	Full search	Update
Distance to enemies	✓	✓
Line of fire to enemies	✓	✓
Cover from enemies	✓	✓
Path distance from the current position	✓	
Time spent in the selected position		✓

Using this technique allows us to search for positions less often while making sure that the currently selected position is still valid. Some of the NPCs were searching for new positions only once per twelve seconds while updating the currently selected position score five times per second.

Implementation of such a system is relatively simple. The harder problem is to come up with a list of scoring functions and tuning them. Here are a few tips and tricks we found

useful during the development of *Mafia III*.

- Invest in your debugging tools. The arguably most practical way to debug different systems is the in-game debug draw (Gregory 14]). We drew a sphere for each considered position and used color coding for fast high level orientation in the results. Red sphere means the position is inaccessible, gray means it didn't make it to the final selection and green means that the position is one of the best. Brightness of the green color signifies the final value where the brightest position is the best. We would also draw a text with all inputs and outputs for individual scoring functions and how the final score was calculated for any position within 5 meters from the game camera.
- Use simple to understand response curves for the scoring functions. We used the piecewise linear functions because they are not just simple to implement but they are also very simple to author. Another option is to use a set of different mathematical functions and provide the designers with some tool, such as Curvature (Lewis 18).
- Less is more. Start with a simple short list of scoring functions and add more only when you really need to solve some edge cases. We often fixed a wrong position selection by going through the list of scoring functions and removing everything that was unnecessary, leaving just the bare minimum. Game development goes through many iterations and as the game changes you might need to change some previously tuned system. Don't be afraid to cut something.

When the position is discarded due to a sudden drop in its score, usually 50%, during movement, the NPC stops and the behavior tree switches into the default behavior in open space. If the NPC would not switch into the default behavior it could potentially make a poor decision that breaks immersion. For example, the NPC might enter a cover position without any protection from threats. We also experimented with an option where the NPC would find a new valid position while still moving into the invalid one. The result was that the movement looked erratic. Stopping and shooting from open space makes the AI look more decisive.

3.5 Target selection

We know how enemies choose where to move, but they also need to know who to shoot, or who should be their primary target. As mentioned above, the position selection system is a generic scoring algorithm. Target selection uses the same system for calculating the score of each potential target and choosing the best one. The difference is the set of scoring functions we use.

NPCs will get a list of all their potential targets, which can include other NPCs, the player or even virtual targets such as the position NPCs expect the player to come from. We could mark up the windows and doors as potential targets in scripted scenarios as well, to give enemies a hint as to where to expect the player. They pass this along with the list of scoring functions into the selection system and they will get the best-scoring target as an output. This process is much faster than position selection because relatively few target candidates need to be evaluated, and usually also with fewer scoring functions.

3.6 Shooting and aiming

Creating an AI which can kill the player within seconds in a game using ray casts for shots is an easy task. The AI just needs to aim at the player's head and shoot. The hard part is making the AI randomly hit in a controlled and believable way.

The behavior tree itself is not deciding when to hit or miss. It decides when to start or stop shooting and whom to shoot. The actual shooting is done in game code which is updated every frame since the behavior tree is updated at too low a frequency. The behavior tree provides a target and a list of additional validation functions to the shooting code. When the game code aims at the target, before it pulls the trigger it will go through the validation functions and run them. If any of them returns false, the NPC cannot shoot in this frame. The most frequently used functions are:

- Shooting style – This function defines how fast the NPC can shoot. The definition says for how long it needs to aim before the first shot and what is the delay between subsequent shots. For example, the sniper could shoot at the player within half a second because that is how long it takes the animation to match the desired direction, but the shooting style can say that he cannot shoot earlier than two seconds after he starts aiming to give the player some opportunity to dodge.
- Ally protection – The NPC is not allowed to shoot if there are any of its allies in the line of fire.
- Player protection – There are gameplay situations when we do not want the NPCs to shoot at the player. For example, during the final stage of the hideout boss interrogation.

4 Combining the systems

All the AI systems have strictly defined roles and they do their job well, but there must be a layer which glues all the systems together to create the final illusion of a human enemy. That is where decision making comes into play, and in our case decision making is done by the behavior tree.

The behavior tree is fed information it can reason about from position and target selection. Let us assume a scenario where the NPC is inside a car and it needs to take cover next to that car to be better protected from the player. The behavior tree starts a sequence of getting the NPC out of the car, moving into a new position and taking cover.

What the NPC does while in cover position could be determined by the behavior tree as well, but we used decision trees which simplified the behavior tree a lot. New behavior trees or branches are not needed for different tactics. The behavior tree runs a decision tree and uses the result as a hint. It also simplifies debugging and authoring, because the decision of what to do is separated from how to do it. Adding a new tactic or changing the possible outcomes of one tactic does not influence the behavior tree at all. An example of a decision tree used by the machine gunner archetype in its "assault" tactic is shown in Figure 7.

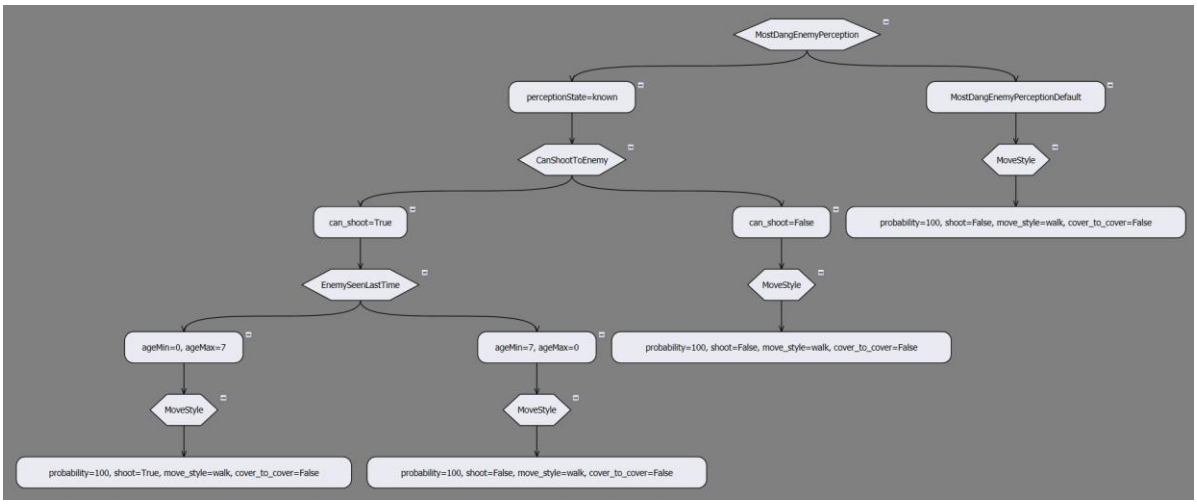

Figure 7 Decision trees were edited with an in-house visual editor.

Let us assume our NPC reached the cover position and is now hiding there. If the tactic is "reserve", it means that the NPC is not putting pressure on the player and is waiting for the right moment. The two actions this NPC can perform are to keep hiding or to stick its head out to look for the player. We run the decision tree and the result is to keep hiding, so the behavior tree plays another loop of a hiding animation. After that, we run the decision tree again and the result this time might be to look for the player, because we have not done that for some time.

The same approach is used in the case of open space behavior. We use a decision tree to select what we should do and let the behavior tree play the necessary sequence of actions. The difference is the set of actions we have available in open space. Enemies can move to either side to avoid the player's bullets, but that is about it.

5 Conclusion

No matter what system is used for decision making, the better information about the world we feed into the system, the better decisions it can make. That is why the supporting systems play such an important role. It is good practice to keep the individual systems separated as much as possible and give them strictly defined roles. It will pay off to follow these rules in the long term - especially right before shipping the game when time is precious - because it will be easier to debug the systems, maintain them and replace them if needed.

Using data-driven systems proved to be worth the trouble of creating them in the first place many times during the development of *Mafia III*. Designers had the creative freedom to experiment with enemy battle behavior without relying on engineering support.

Adding the inheritance support for the data was equally important. The final game has eighteen base archetypes and one hundred twenty variations of those. That would be a lot of variables to define and it would be a nightmare to maintain without the inheritance. Most of the variations overrode just a couple of values. We shipped the game with a couple

systems hard-coded, but we hope to make them data driven in future projects.

6 References

[Colledanchise 18] Michele Colledanchise, Petter Ögren. 2018. *Behavior Trees in Robotics and Al: An Introduction.* Boca Raton, FL: CRC Press.

[Gregory 14] Jason Gregory. 2014. *Game Engine Architecture*. Boca Raton, FL: CRC Press.

[Lewis 15] Mike Lewis, Dave Mark. 2015. Building a Better Centaur: AI at Massive Scale. GDC 2015. http://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI (accessed March 24, 2018).

[Lewis 17] Mike Lewis. 2017. Guide to Effective Auto-Generated Spatial Queries. In *Game AI Pro*^{3}*: Collected Wisdom of Game AI Professionals*, ed. S. Rabin, 309-325. Boca Raton, FL: A K Peters/CRC Press.

[Lewis 18] Mike Lewis. 2018. Winding Road Ahead: Designing Utility AI with Curvature. GDC 2018. http://gdcvault.com/play/1025310/Winding-Road-Ahead-Designing-Utility (accessed April 29, 2018).

[Ocio 18] Sergio Ocio Barriales, Kate Johnson. 2018. Triage on the Front Line: Improving 'Mafia III' AI in a Live Product. GDC 2018. http://gdcvault.com/play/1025291/Triage-on-the-Front-Line (accessed April 29, 2018).

[Simpson 14] Chris Simpson. 2014. Behavior trees for AI: How they work. Gamasutra. https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php (accessed March 8, 2018).

[Sloan 15] Robin J. S. Sloan. 2015. *Virtual Character Design for Games and Interactive Media.* Boca Raton, FL: A K Peters/CRC Press.

[Straatman 05] Remco Straatman, William van der Sterren, and Arjen Beij. 2005. Killzone's AI: Dynamic Procedural Combat Tactics. GDC 2005. http://www.cgf-ai.com/docs/straatman_remco_killzone_ai.pdf (accessed March 8, 2018).

[Walsh 15] Martin Walsh. 2015. Modeling Perception and Awareness in Tom Clancy's Splinter Cell Blacklist. In *Game AI Pro*^{2}*: Collected Wisdom of Game AI Professionals*, ed. S. Rabin, 313-326. Boca Raton, FL: A K Peters/CRC Press.