33

Using Your Combat AI Accuracy to Balance Difficulty

Sergio Ocio Barriales

33.1 Introduction

In a video game, tweaking combat difficulty can be a daunting task. This is particularly true when we talk about scenarios with multiple AI agents shooting at the player at the same time. In such situations, unexpected damage spikes can occur, which can make difficulty balancing harder. This chapter will show how to avoid them without compromising the player experience and while still giving designers lots of affordances for balancing.

There are a few different ways to deal with this problem. We could adjust the damage AI weapons do; we could add some heuristics that dynamically modify damage values based on things such as the time elapsed since the player was last hit or the number of AIs that are simultaneously targeting the player; or we could have AIs be less accurate and only really hit the player once every few shots.

The latter will be our focus for this chapter: Playing with AIs' accuracy to achieve better control over the amount of damage the player can receive each frame. This is a complicated and interesting topic, with two main parts:

1. What algorithm is used to decide when it is time to hit the player? How many agents can hit the player simultaneously?
2. How can we make our AIs not look ridiculous or unrealistic when they are purposely missing or holding off their shots?

33.2 Damage Dynamics

In an action game, difficulty level is usually related to how likely the player is to die during a game, or to the amount of experience required to progress through the experience (Suddaby 2013). Depending on the design of the game the way we control this can change. For example, in a first-person shooter we can decide to make enemies more aggressive, carry more powerful weapons, or simply spawn a higher enemy count; or, in a survival game, we could reduce the ammunition a player will find scattered through the world.

Adjusting how challenging a video game experience should be is normally a long and costly process in which different teams collaborate to improve the final player experience (Aponte et al. 2009). Programmers will expose new parameters for designers to tweak, enemy placement will change, levels will continue evolving and things will be tested over and over until the game is released … and beyond!

Let us focus on the average cover shooter game. Normally, in these games, a few enemies will take covered positions that give them line-of-sight on the player. AIs will move around the level to try and find better firing locations if their positions are compromised or if they lose line-of-sight on their target, and they will unload their clips at the player, taking breaks for weapon reloading to add some variety to their behaviors. Combat will usually continue until (a) every AI is killed, (b) the player is killed, or (c) the target is not detected anymore.

In a scenario like this, one of the easiest ways to control difficulty, providing we keep enemy count constant, would be playing with damage dynamics, that is, adjusting the amount of damage inflicted by the AI when the player is hit (Boutros 2008). This is not a complex method in terms of the programming involved, but it requires lots of tweaking by designers.

Although this is a good strategy, it also has its problems. For instance, we could still face the problem of multiple AIs potentially shooting and hitting the player at the same time. This is solvable, for example, by tracking the total damage the target has received each frame and adjust incoming damage accordingly (e.g., not damaging the target anymore after a certain threshold has been hit), but this could lead to other believability problems; though solvable, the focus of this chapter is on a different type of solution, that we will talk about in the subsequent sections.

33.3 Token Systems

One way to address the one-hit kill syndrome is to have AIs take turns when attacking the player. This is commonly achieved by using a logical token that gets passed around the different agents involved in combat. Tokens can control different behaviors, but they basically fulfill the same purpose no matter what the action is: Only the AIs that hold a token will be able to execute the behavior. For example, we could decide that only AIs with a token can fire their weapons. If we have, let us say, just one token for the whole game,

this means one, and only one, AI will be able to shoot at any given point in time. When the shooting is done, the token holder will release it, and another agent will take a turn.

The problem is that since only one AI is allowed to use its weapon at a time this could yield potentially unbelievable behaviors, such as AIs being at perfect spots to shoot and hit the player but not even trying to do so because they are clearly waiting for their turn.

33.4 Dynamic Accuracy Control

Our goal is to control the difficulty of our game and offer players a fairer, less chaotic game. Token systems will help us achieve that, but we can still make some improvements to offer a more polished experience.

Let us change the rules we used in the previous section. Tokens will not gate the shooting behavior anymore; instead, AIs can freely enter any of the shooting behaviors, but shots can only actually hit the player if the shooter has a token; any other shot will deliberately miss the player and hit some location around him or her.

Token distribution is controlled by a global timer that tracks how long has passed since the last hit. But, how long should the delay between hits be? Players will understand agents are not so accurate if, for example, they are far from their targets. In that case, having a long delay between hits is not a problem. But missing shots can affect the believability of our AI if the target is clearly visible and in range. To try and have the best of both worlds—predictability over the damage the target is going to take and a believable behavior, we need to use a variable, dynamic delay between hits.

33.4.1 Calculating the Final Delay

To define this delay, we will start from a base value, and define a few rules that will generate multipliers for our base value, increasing or decreasing the final time between shots. The final delay will be calculated as:

$$\text{delay} = \text{delay}_{\text{base}} * \prod_{i=0}^{n} \text{rule}_i$$

Where rule_i is a floating-point value resulting of running one of our rules.

In this section we will simplify things and say that we are fighting a single AI agent—we will show how to deal with multiple agents in the next section. With this in mind, let us define a few rules and show what the delay would look like in a couple of different scenarios.

33.4.2 Rules and Multipliers

Our first rule is distance. We want players to feel more pressure if they are closer to the AI, so we will allow our agents to hit players more frequently the closer they are to their targets. For our example, we will say that any distance greater than 15 m will not affect the delay, and that anything closer than 5 m will halve the time. Figure 33.1 shows the function we will use to generate our multiplier based on distance.

For our second multiplier, we will check the current stance of the player. In this case, we will keep the base delay if the player is standing (i.e., the multiplier is 1), but double it

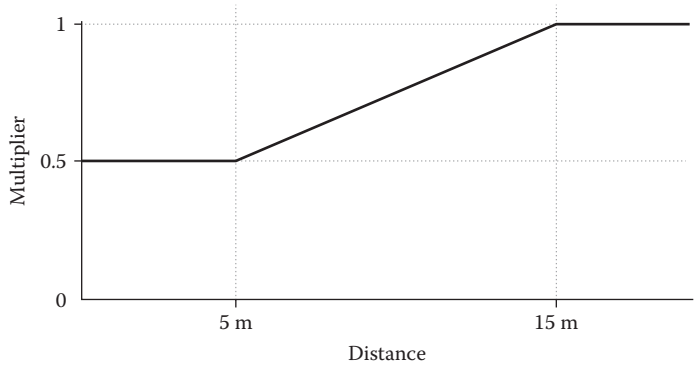

Figure 33.1

Our distance rule will generate a multiplier that will make the delay smaller the closer the player is to the AI.

if they are crouching, making the player feel safer. Likewise, we will use another rule that will double the delay if the player is in a valid cover position.

The current facing direction of the player is also an interesting factor. In this case, we can use it to be fairer and more forgiving with players if they are looking away from the AI, doubling the delay if the angle difference between the facing vector and the vector that goes from the player to the AI is greater than 170 degrees, and leaving the delay unchanged otherwise. Similarly, we can use the velocity of the player to check if they are trying to run away from the AI or toward our agents. Based on the angle between the velocity vector and, again, the vector that goes from the player to the AI, we will calculate a multiplier such as shown in Figure 33.2.

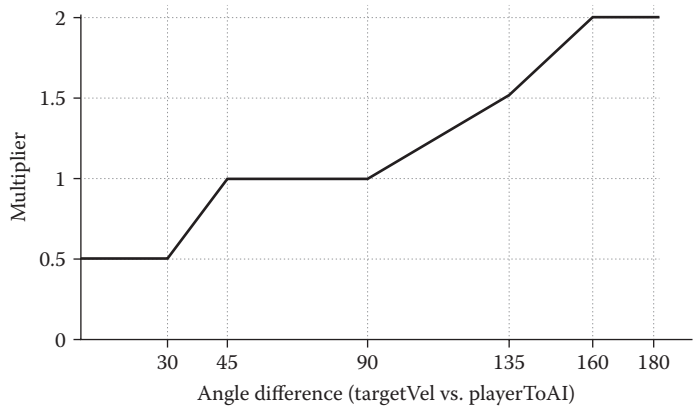

Figure 33.2

For our rule that takes into account target velocity, we have used a more complex curve to define how the multiplier is calculated.

For our example, this rule will halve the delay if the angle is lower than 30°, maintain it unchanged between 45° and 90° and make if longer the closer we get to 180° (we are actually capping it at 160°).

Finally, we could use other rules and multipliers, such as a rule based on the weapon the AI is wielding, but, for simplicity, we will not study these in this chapter.

33.4.3 Dynamic Delay Calculation Examples

With our rules defined, let us say our base delay is 0.5 s and define a first scenario in which the player is hiding, crouching behind cover 30 m away from the AI. With these constrains, we have:

- The distance multiplier is 1, since we are well over the 20 m cap.
- The stance of the player generates a multiplier of 2.
- Because the player is behind cover, we have another multiplier of 2. Also, since we are in cover, we will ignore the rest of the rules, as they are not applicable in this case (e.g. there is no player velocity, and we can consider the player facing is not relevant).

The final delay is $0.5 * (1 * 2 * 2) = 2\ s$.

We will define a second scenario in which the player is out in the open, running toward the AI, 10 m away. Our rules would be yielding the following results:

- Based on distance, we have a multiplier of 0.75.
- The player is standing, which results in a multiplier of 1, and not behind cover, so that is another 1 for that rule.
- The player is looking at the AI and running toward it. Let us say that, in this case, both facing and velocity vectors are the same—they could be different if, for instance, the player was strafing—and the angle difference is very close to 0 degrees. This will generate a multiplier of 1 based on facing and the target velocity rule will calculate a 0.5 multiplier.

The delay for the second scenario is $0.5 * (0.75 * 1 * 1 * 1 * 0.5) = 0.1875\ s$.

The dynamic delay will handle what the player is doing and increase—by allowing the AI to hit the player more frequently—or decrease difficulty accordingly. This system avoids multiple hits occurring on the same frame and can be tweaked by designers to balance the game in a way that is reactive to players' actions.

33.5 Dealing with Multiple AIs and Different Archetypes

In the previous section, we studied how our solution would work if we are dealing with a single enemy. However, this is normally not the case. Enemies will most likely be spawned in groups, and we can, as well, face different enemy types (archetypes). In this case, the algorithm remains very similar: We still use a single timer and we need to update its duration properly. The main difference is that, in the case of a single AI, all the calculations were made by applying our rules to that agent; when multiple AIs are present, we need to choose which should receive the token, and calculate the duration of the delay for that particular agent.

33.5.1 Selecting the Most Relevant Agent

Calculating which AI should get the token involves looking at a few different variables. Some of these are as follows:

- *Distance to the player:* The closer an agent is to the target the more options it has to receive a token.
- *Target exposure:* Depending on the position of the AI agent, the player can be more or less exposed to the attacks of the AI. For example, a cover position can be valid against a particular agent, but the same position can be completely exposed to an AI agent that is standing directly behind the player. The latter agent would have a better chance of obtaining the token.
- *Archetype:* The tougher the type of enemy, the easier it is for it to get the token.
- If an agent is currently under attack, it is more likely that it will receive the token.
- *Token assignment history:* Agents that have not received a token in a long time may have a higher chance of receiving the token soon.

A weighted sum will be used to combine these factors, generating a score for each actor. The actor with the highest score will be chosen as the most relevant agent. The weights we use can vary depending on the type of experience we want to offer. For example, if we had an enemy with a special weapon that generates some effect on the player that is really important for our game, we could use a very high weight for the archetype factor while using lower values for other ones. That way we would almost be guaranteeing that the special enemy will be the one hitting the player.

33.5.2 Dealing with Changes

Rechecking who is the most relevant agent every frame, we solve problems that are inherently common in action games, such as what happens if an AI reaches a better position, if the player keeps moving, or if an AI gets killed. Let us analyze a couple of scenarios.

In the first one, we have two agents, *A* and *B*. *A* is selected as the most relevant agent and the delay is 1s. The player runs toward *B* and starts shooting at it. The algorithm determines *B* is more relevant now since it is under attack and changes the selection. Also, because the player is running directly toward it, the facing and target velocity rules decide the delay should be shorter, so it is recalculated as 0.75 s. Time passes and the timer expires based on the new delay, so *B* hits the player, who turns around trying to retreat after being hit. *A* becomes the new selected AI and the delay goes back to 1 s. A second later, *A* hits the player.

For the second scenario, we have three agents—two regular soldiers (*A* and *B*) and a heavy soldier (*C*). *C* has just hit the player and now *A* is the most relevant AI. The delay for *A* is 2 s. After 1.5 s, *A* is killed. The algorithm determines *C* is, again, the most relevant AI, since it is in a better position than *B* and it is wielding a more powerful weapon. *C* is also a heavy, and the system calculates the delay is now 0.75 s. Since it has already been 1.5 s since the last hit, *C* gets a token immediately and the player is hit.

33.6 Improving Believability

If we want players to enjoy our game and our AI, we need to prevent breaking the suspension of disbelief (Woelfer 2016). In our case, this means hiding the processes that are happening behind the scenes—our damage control system—from players, so they are not distracted by the fact that the AI is being generous and missing shots on purpose to make the game more fun.

With this in mind, we have to take two points into consideration:

1. Although we normally communicate to the player he or she is being hit by using some HUD or VFX on the screen, we would completely lose that information if the player is not actually being hit.
2. Since most of the shots are going to miss the target, we need to make players not notice this fact.

33.6.1 Conveying Urgency

An important part of every game is that players understand the different situations they face and know how to identify and read the messaging the game provides about its systems and/or the state of the world.

If we stop hitting the player, we still need to produce a feeling of "being under pressure" when he or she is being targeted by an AI, in order to maintain engagement. We will mainly do this through sound effects and visual effects. For the former, 3D positioned sounds can help players pinpoint where shots are coming from, and also that some of them are missing the target; for visuals, we will use tracers and some VFX, like sparks or other particle effects.

33.6.2 Choosing Interesting Random Targets

If we want to make things interesting and good looking—so we can divert attention from the fact that the AI's accuracy is not very good—we need to ensure bullet hits will be seen by players and that they will clearly convey the player is under attack (Lidén 2003). This applies both to single shot weapons and automatic or semi-automatic ones; for the latter types, we could either calculate an interesting target for each bullet in the burst or, potentially, generate a single target and add some randomization around it or some special patterns, such as straight lines that the recoil of the weapon may be generating, to create better effects. The objective to keep in mind is that we are trying to polish the looks of things!

Let us refine the idea of picking random locations around the target, and ask ourselves: Are all these missed shots going to be noticed by the player? The answer is probably not. If we are just aiming at a random position in a circle around the target, as we show in Figure 33.3, the only thing players are going to notice for most of the shots is their tracers.

Instead, what we want is to make our "accuracy problems" an interesting situation from the visual standpoint, so what we should try to do is hit things surrounding the player to generate sparks, dust… in a nutshell, destruction. We still want to randomize our shots and fire the occasional one that does not hit anything, but we should try and minimize them.

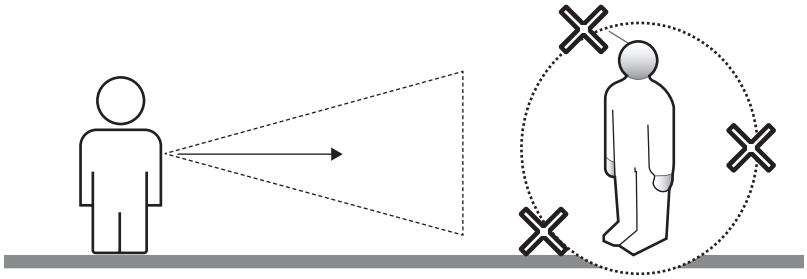

Figure 33.3

An AI that is consciously missing shots could target random positions around the enemy.

So, how do we choose better targets? In the average cover shooter, we will find that almost all scenarios fall in one of these three categories: Target is behind full cover, target is behind half cover, and target is out in the open. Let us analyze each of these individually.

33.6.2.1 Target Behind Full Cover

We call "full cover" anything that occludes the full height of the target. These are normally door sides, columns, pillars, and so on. Peeking out can only be done left and/or right, depending on which sides of the cover gives the user of the cover line-of-sight on its target. Figure 33.4 depicts this type of location.

If the target is behind full cover, we have two options. First, we can aim at positions near the ground next to the peek-out side of the cover; this are most likely going to be noticed by the player, since they are hitting the area he or she would have to use to leave the cover. Alternatively, we could also hit the cover directly, especially when the player is completely peeking out, as these hits are going to remind the player he or she can be hit at any point.

33.6.2.2 Target Behind Half Cover

A "half cover" is that which only occludes half of the height of the user, requiring him or her to crouch behind it to be fully protected. In this case, peeking out can also be done standing up and looking above the cover. Figure 33.5 shows this scenario.

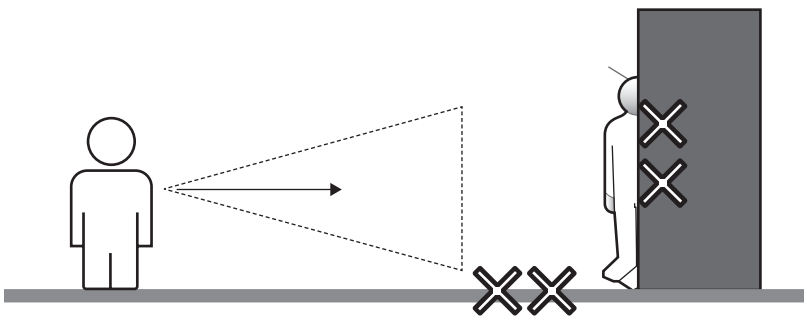

Figure 33.4

The AI's target is behind full cover, so we can target both the ground next to the cover and the cover itself.

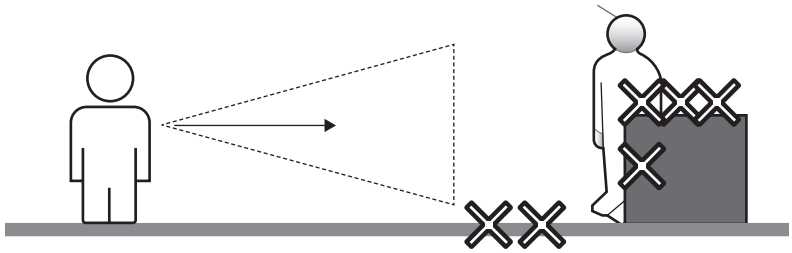

Figure 33.5

The AI's target is behind half cover, so we can target the ground next to the cover, the top of the cover and its side.

If the target is behind half cover, our options are similar to the ones we had in a full-cover situation. However, now we have an extra targetable area: the top of the cover. Most of our missed shots should aim at this zone, since they are closer to the eyes of the target, and thus, more likely to be seen.

33.6.2.3 Target Out in the Open

If our target is out in the open, the position we should choose is not as clear, and it depends on what the player is trying to do. For example, if the player is stationary, we could potentially shoot at the ground in front of the player, if this area is within camera view. But normally players will be moving around, so our best bet will be trying to whiz tracers right past the player's face at eye level.

33.6.3 Targeting Destructible Objects

Although we have presented some tricks to help choose the best random targets we can try to hit when we do not have the shooting token, we can always tag certain objects in the scene that are destructible and use them to create more spectacular and cinematic moments.

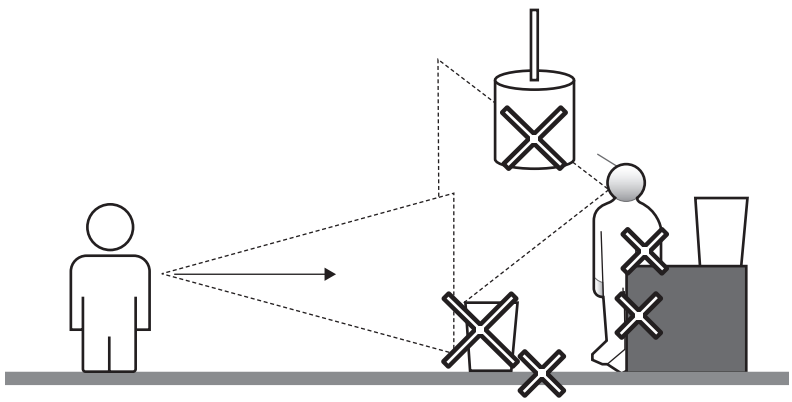

Figure 33.6

Destructible objects that are within the player's FOV should be targeted when our AI agents miss shots.

It is important to note that the destructibles we choose to target should also be in generally the same area as the player, if we want things to remain credible. Moreover, we should only target these special items if the player is going to see it, since these objects are limited in number and we should not waste the opportunity to use them. Figure 33.6 shows an area with destructible objects.

33.7 Conclusion

In this chapter, we have talked about different options to help make damage throughput more predictable, making it easier for designers to balance the game. But we also showed that, in order for our AI to look smart and believable, we need to control how the AI is going to deal with its targeting decisions.

Game AI is an art of smoke and mirrors—everything is permitted as long as we create a spectacular and fun experience. Tricks like playing with our AIs' accuracy can be used to control difficulty, and can help designers create better and more enjoyable games. It is our hope that readers can make use of these, or similar, techniques to keep improving what AI has to offer.

References

Aponte, M., Levieux, G., and Natkin, S. 2009. Scaling the level of difficulty in single player video games. In *Entertainment Computing–ICEC 2009*, eds. S. Natkin and J. Dupire. Berlin, Germany: Springer, 2009, pp. 24–35.

Boutros, D. Difficulty is difficult: Designing for hard modes in games. http://www.gamasutra.com/view/feature/3787/difficulty_is_difficult_designing_.php (accessed April 29, 2016).

Lidén, L. 2003. Artificial stupidity: The art of intentional mistakes. In *AI Game Programming Wisdom 2*, ed. S. Rabin. Rockland, MA: Charles River Media Inc.

Suddaby, P. Hard mode: Good difficulty versus bad difficulty. http://gamedevelopment.tutsplus.com/articles/hard-mode-good-difficulty-versus-bad-difficulty--gamedev-3596 (accessed May 9, 2016).

Woelfer, A. Suspension of disbelief|game studies. Video game close up. https://www.youtube.com/watch?v=Y2v3cOFNmLI (accessed May 9, 2016).