Reversing Anti-Cheat's Detection-Generation Cycle With Configurable Hallucinations

David Durst, Carly Taylor – March 30, 2023

Abstract

Anti-cheat and cheat developers are locked in a cat-and-mouse cycle of detection and generation. Anti-cheat developers struggle to detect cheating players' behavior. Cheat developers easily evade the detections by generating different behavior.

Anti-cheat developers may reverse this cycle with hallucinations. Hallucinations are in-game entities only visible to cheating players. Cheat developers must detect them, or cheating players will react and self-identify. Hallucinations' key property is configurable in-game behavior. Anti-cheat developers can respond to the detections by generating different behavior. Hallucinations' configurability reduces the anti-cheat problem to the bot design problem. As game developers create bots that better imitate humans, they can reuse these behaviors to make hallucinations harder to detect.

In this paper, we define hallucinations; discuss prior, non-configurable implementations; explain our implementation; and provide evidence that popular cheating techniques treat our hallucinations as real players.

Motivation

The competition between anti-cheat and cheat developers is an asymmetric, repeated cat-and-mouse cycle. Anti-cheat developers detect new behaviors indicative of cheating. Cheat developers respond by generating different behavior. Anti-cheat developers detect the new behaviors, and the cycle continues. Anti-cheat developers' traditional goal is to detect a wider range of inhuman behaviors each cycle iteration, forcing cheat developers to produce increasingly human-like behavior over time. Cheating players' unfair advantages should decrease as their behavior becomes more human. The problem with this approach is the task asymmetry: anti-cheat developers solve a harder problem, behavior detection, and cheat developers solve an easier one, behavior generation.

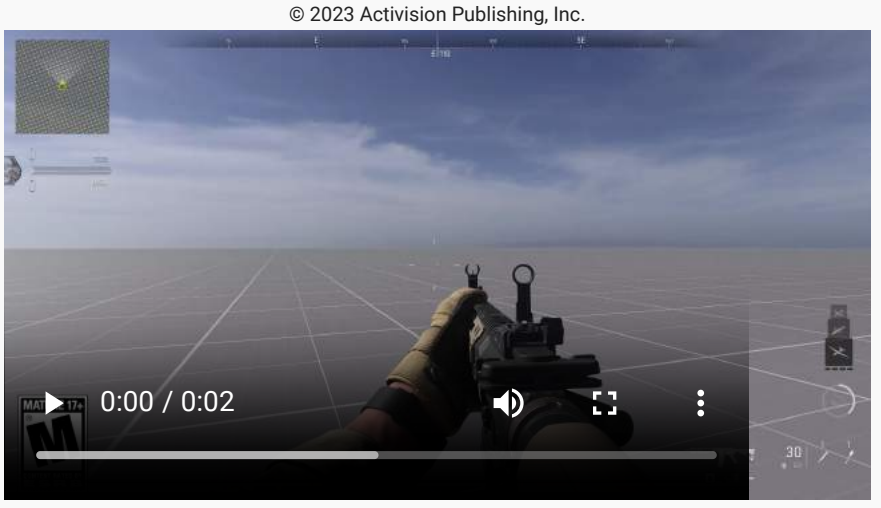
We want to reverse this cycle by generating behavior that cheat developers must detect. Anti-cheat developers can create in-game entities that are perceptible only by cheating players. Any player who reacts to the entities will self-identify as a cheater. Cheat developers invariably will detect these entities' inhuman behavior and remove them. If anti-cheat developers can easily configure the entities' behavior, they can respond by generating more human-like behavior, breaking the cheat developers' detectors, and restarting the cycle. The reversed detection-generation cycle is advantageous for anti-cheat developers because they solve the easier problem.

Example - Why the Current Cycle is Problematic

To better understand the detection-generation asymmetry, let's consider how anti-cheat developers might catch cheating players performing 180 degree flickshots. A 180 degree flickshot occurs when a player turns around 180 degrees and kills an enemy with a single flick of the mouse or joystick.

Activision anti-cheat developers might hypothesize that players generating perfect 180 degree flickshots are cheating. A perfect flickshot requires a player to quickly move their crosshair and stop precisely on the enemy's head. A skilled player can only approximate such a behavior. They'll take longer to move their crosshair and won't stop perfectly on the enemy's head.

Anti-cheat developers must spend significant time to detect perfect flickshots. They'll track new features of player behavior, re-train their models, and update the deployed detection pipeline. Once the update is



The player looks into the distance, seemingly unaware of the danger behind them. Then, the player perfectly flicks 180 degrees onto the enemy's head.

deployed, cheat developers can circumvent the detector by modifying their aimbots. A YouTube tutorial and fifty lines of code can add enough mistakes to make the flicks imperfect. We want to reverse this cycle so that anti-cheat developers can easily generate new behavior; cheat developers must update their feature logging tools, retrain their models, and deploy their detection pipeline updates.

Approach - Configurability is the Key

Hallucinations may reverse the cycle. Hallucinations are in-game entities that seem like human enemies to cheating players but are imperceptible to non-cheating players. For example, only a cheating player will react to a hallucination hidden inside a wall.

A hallucination hidden inside a wall. Only cheating players with wallhacks can perceive it.

We aren't the first anti-cheat developers to utilize hallucinations. Two Counter-Strike 1.6 (CS 1.6) server plugins utilized hallucinations over a decade ago: Lucia Hallucination and AimBot Detection. These systems tricked cheating players into aiming at entities that seemed like real players, but were hidden inside the map or directly above a cheating player.

The CS 1.6 plugins demonstrated hallucinations' potential and key challenge: how should they behave in-game? They must render if visible. If the hallucinations are marked invisible, cheat developers will have a clear signal that a hallucination isn't a real player and can easily filter it out. However, if the hallucinations are visible to non-cheating players, then non-cheating players will react to them. Hallucinations are useless for detecting cheating players if non-cheating players react to them. We must ensure that the hallucinations act like real players without becoming visible to non-cheating players.

The solution is configurability. Cheat developers will detect any static behavior, like hiding inside of a wall. The CS 1.6 plugins used static hallucination behaviors and were rendered obsolete once cheat developers identified them. Anti-cheat developers can use configurability to respond with new behaviors that break the detectors. Configurability reverses the detection-generation cycle because cheat developers must repeatedly detect new hallucination behaviors generated by anti-cheat developers.

Configurability is not a panacea. Configurability merely reduces the anti-cheat problem to the bot design problem: how do you design a bot to imitate human behavior? If you can generate bot behavior controllers that imitate humans, then you can reuse the controllers to drive hallucinations. If the hallucinations imitate humans, then they'll be hard to detect. If cheat developers can't detect the hallucinations, then cheating players will react to them and self-identify. The core assumption of configurability is that generating human-like hallucination behavior is easier than detecting it. We believe generation is easier than detection because we have experience detecting cheaters and designing bots.

Implementation - Adding Fake Players to the Snapshot

Any hallucination implementation must satisfy four constraints. The first three ensure that the hallucinations can be deployed at scale and produce in-game entities to trick cheating players. The CS 1.6 plugins satisfied these constraints. The last one is necessary to reverse the detection-generation cycle.

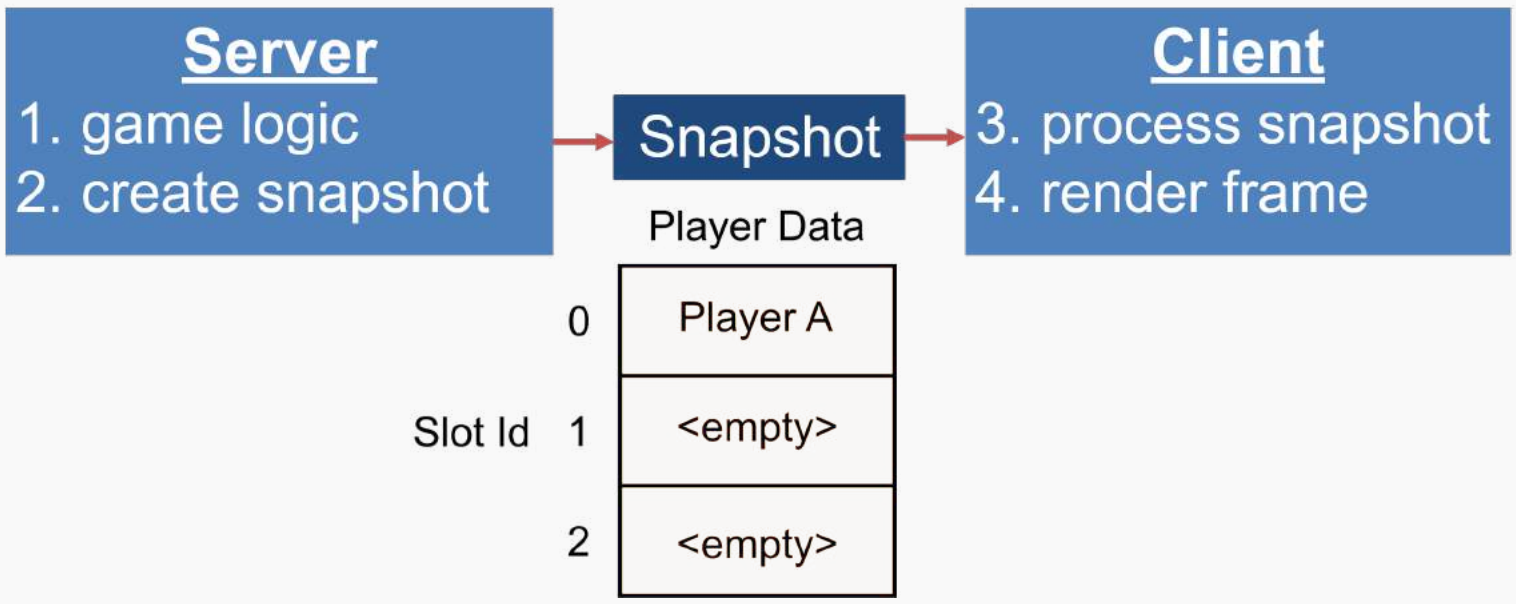
1. Hallucinations look like real players in memory.
2. Non-cheating players are not able to observe the hallucinations.
3. Implementation is maintainable and performant.
4. Hallucination mechanism supports configurable in-game behavior.

We made two key design choices to satisfy these constraints:

1. At what time in the game loop should we insert the hallucinations?
2. How do we modify the game state to insert them?

Design Choice 1: When to Create Hallucinations?

The below image shows the four steps in the game loop that are relevant for hallucinations. We must create the hallucinations during the second step in order to satisfy our constraints.



First, the server runs the game logic to update its state. The state includes where players are moving, aiming, and shooting. We can't change the server state when we create the hallucinations. Otherwise, we may introduce bugs in the game logic. Non-cheating players will notice if an indestructible, invisible hallucination prevents them from eliminating an enemy team and winning in a battle royale game.

The server serializes the game state into the snapshot during the second step. For this paper, we simplify the snapshot to an array of player data. We can create the hallucination by modifying the snapshot without violating our constraints. We won't introduce game logic bugs observable by non-cheating players since this frame's server state is finalized. Our hallucinations will look real in client memory since all changes are made to the snapshot before it's sent to the clients.

The clients process the snapshot during the third step. The processing updates their local state to match the server's state. The hallucinations must exist before this step. Otherwise, cheat developers can watch this client-side code to identify newly created entities as hallucinations.

The clients render the image during the last step. The hallucinations cannot appear in the image. Non-cheating players may react to the hallucinations if they are visible, invalidating hallucinations a cheating player detection mechanism.

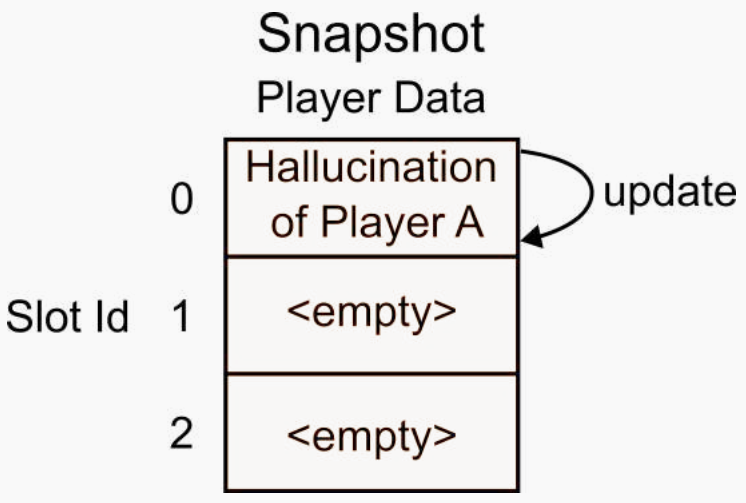
Hallucinations must be created during the second step. It's too early to create them during the game logic step, as the server state hasn't been finalized. It's too late to create them on the client side, as cheat developers can watch client-side code. The snapshot's timing enables us to create hallucinations without violating our constraints.

Design Choice 2: How to Insert Hallucinations?

There are three options for how to modify the snapshot:

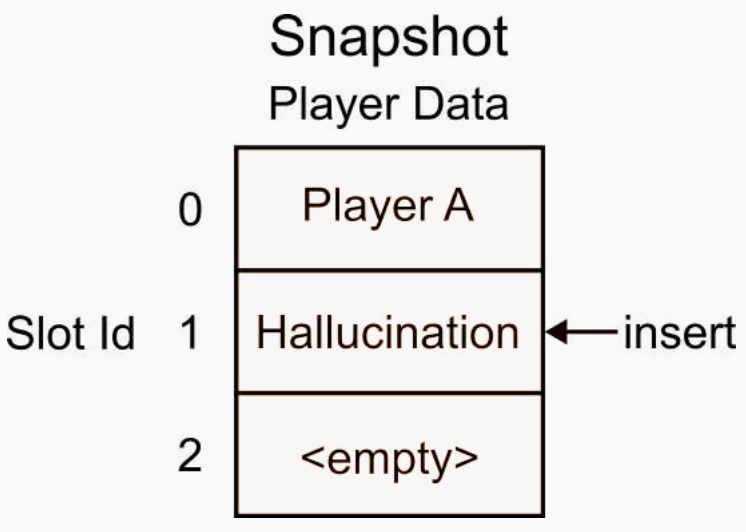
Option 1: Modify In-Place

We could modify a human player's entry in the snapshot. We would change the entry during every snapshot creation. This is efficient but complicated. The operation removes the original player from the snapshot. We now need to track multiple snapshots if we want both the hallucination and the original player. The complexity violates our maintainability constraint.



Option 2: Modify Server State

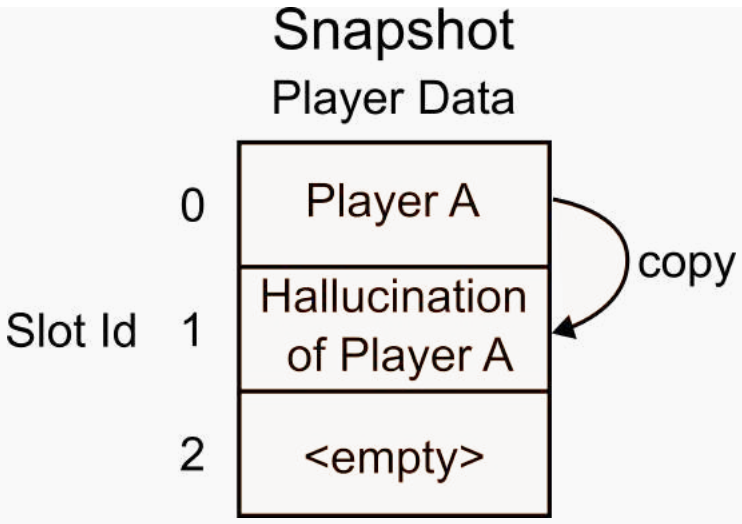
We could modify the server state so the snapshot is created with the hallucination. This is simple to implement because we can reuse the existing code to generate the snapshot. We don't need to modify it after creation. This approach violates our constraints because modifications to the server's state can cause bugs that are observable by non-cheating players.



Option 3: Copy and Update

We implemented hallucinations by creating fake players copied from real players. Every frame, we find a real player's entry in the snapshot and an empty snapshot entry. We copy the real player's entry into the free one and

update the copy to create the hallucination. This approach satisfies all our constraints. The copy is fast and simple. The resulting snapshot entry looks like a real player in memory, as it has all the properties of a real player. The process enables configurable behaviors since we can dynamically update the copy's properties every snapshot. We rely on the behavior generator to hide the hallucination from the non-cheating players.



Configurable Behavior Policy

We provide an API for bot designers to configure hallucinations' behaviors. The API generates behaviors by modifying the fake player every snapshot creation. The below videos demonstrate a range of behaviors expressible with our API. We start with the simplest behaviors, like placing a hallucination in front of a possibly cheating player. We have deployed these behaviors and expect that cheat developers will detect them. After several iterations around the detection-generation cycle, designers can implement behaviors that more closely imitate humans. The last video is a prototype future behavior. Future work may integrate imitation learning techniques to generate these behaviors from human game logs.

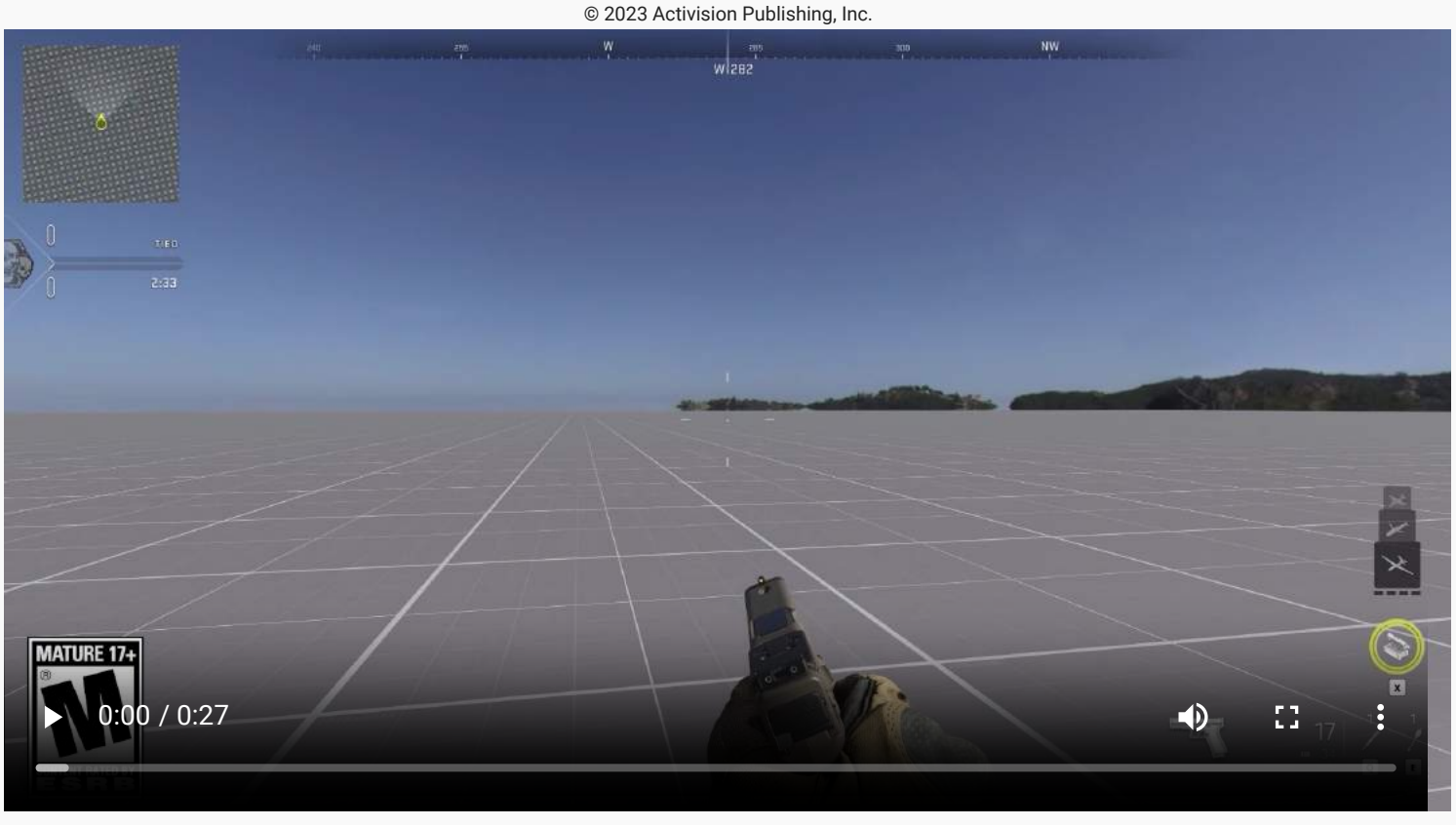
Behavior generators deployed in production ensure that the hallucinations aren't observable by non-cheating players. The generators below are visible to non-cheating players for demonstration purposes.

Deployed Option 1: Position

The hallucination can be positioned directly in front of the possibly cheating player at different distances, like beyond max draw distance.

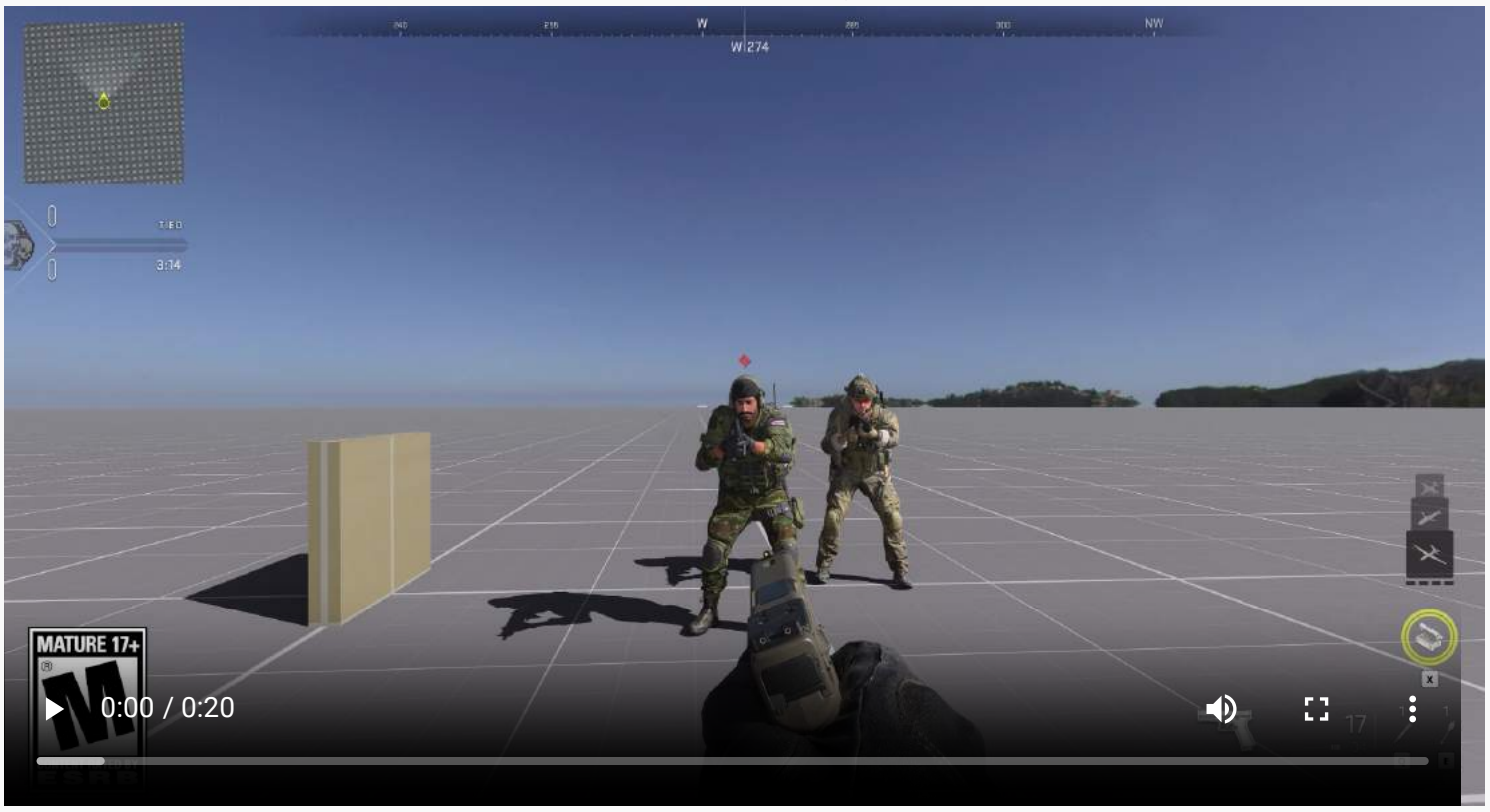
Deployed Option 2: Angle

The hallucination can be angled behind the possibly cheating player. This behavior is slightly more realistic than the positioning one. (Note: Please ignore the ground-level enemy. They are a human player used in the Emerge prototype below.)



Prototype: Emerge

The hallucination can start inside a human player and emerge out of them. The character with tan clothes is a hallucination, and the one with green clothes is a real player. The cheat developer must guess which is real and which is fake based on their trajectories through the map. Future work will make the hallucination's trajectory more realistic.



Results

We verified that our hallucinations look like real players in memory by testing them against combinations of features from three popular cheat programs. If the hallucinations only worked for a subset of features or programs, that would indicate our hallucinations don't look like real players in memory. The cheat programs treated our hallucinations like real players for all accessible feature combinations.

	ESP Basic	ESP Properties	Radar	Aimbot Not Behind Wall	Aimbot Behind Wall	ESP + Aimbot	ESP + Aimbot FOV Out of Range	ESP + Aimbot FOV in Range
A	Success	Success	Success	Success	Success	Success	Success	Success
B	Success	Success	Radar not available	Success	Success	Success	Success	Success
C	Success	ESP Properties not available	Success	Aimbot not available	Aimbot not available	Aimbot not available	Aimbot not available	Aimbot not available

The first five experiments test individual cheat features: (1) extra sensory perception (ESP) that shows enemies behind walls and displays their loadouts, (2) radar that always shows enemies, and (3) aimbot that targets enemies automatically. The last three experiments test combinations of cheat features. FOV is a feature that prevents aimbots from activating when the target is out of a field-of-view limit. This prevents too blatant lock-ons. We want the aimbot to not lock-on when the FOV limiter is active and the hallucination is out of range.

The following images and videos contain no actual hallucination-cheat tests. They are recreations of the tests. We used engine debug features, photo editing tools, and video editing tools to recreate the tests for public release.

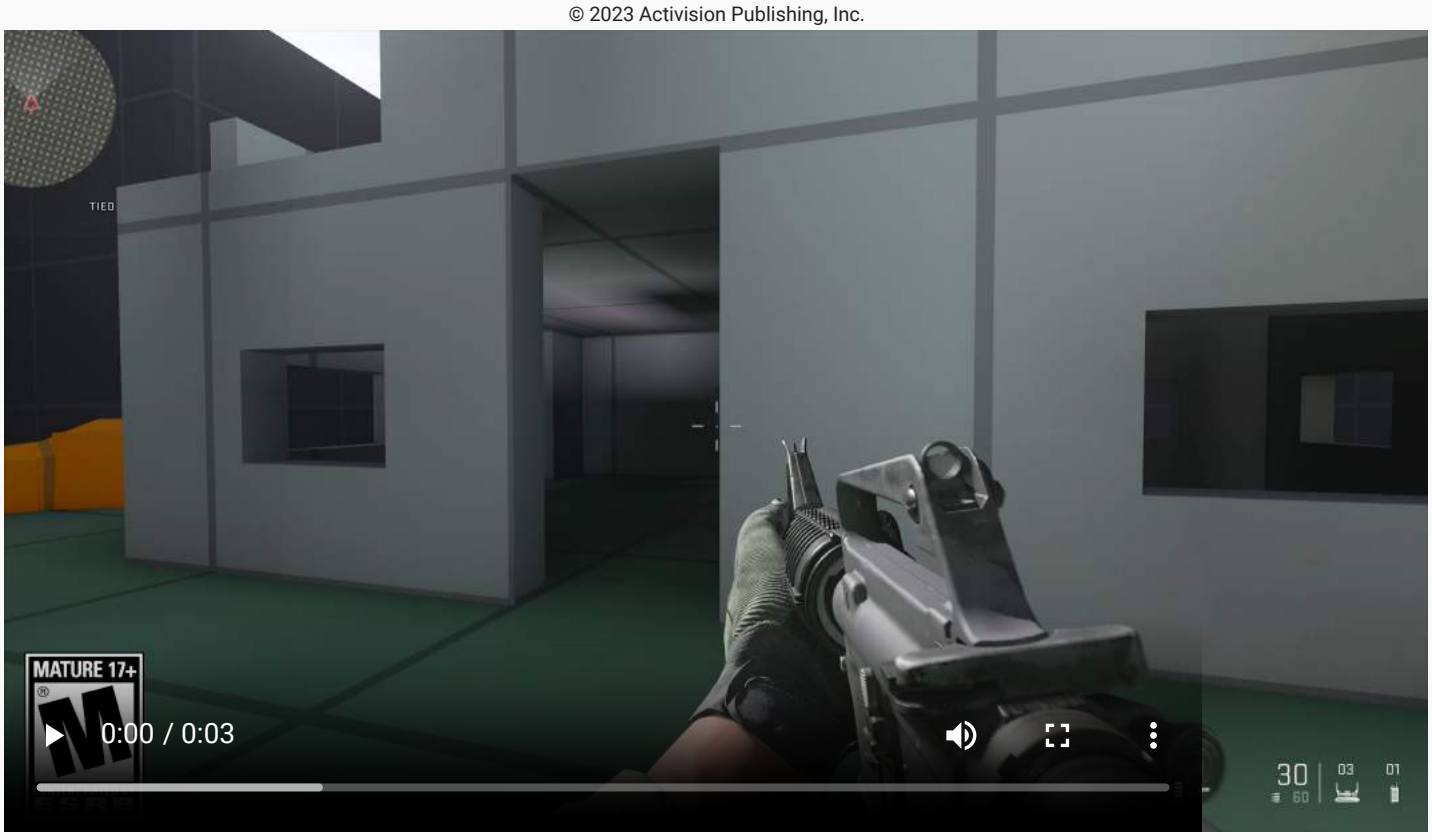
1. **ESP Basic**

2. **ESP Properties**

3. **Radar**

4. **Aimbot Not Behind Wall**

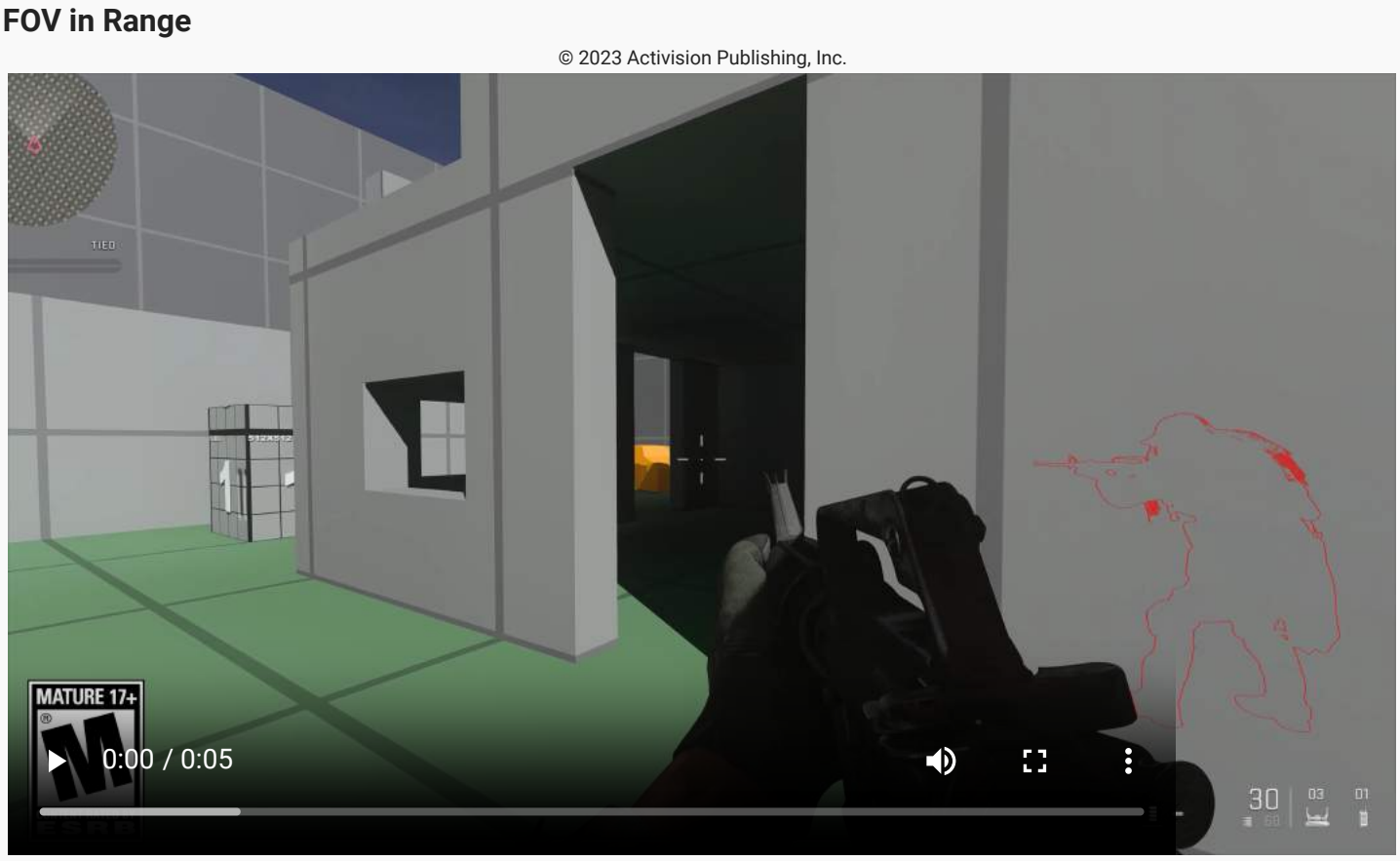
5. **Aimbot Behind Wall**



6. **ESP + Aimbot**

7. **ESP + Aimbot FOV Out of Range**

8. **ESP + Aimbot FOV in Range**



Conclusion - Hallucinations are Bots for Anti-Cheat

Hallucinations reverse anti-cheat's detection-generation cycle. Anti-cheat developers generate hallucination behavior, and cheat developers detect it. The key assumption is: generating imitation human behavior is easier than detecting it. Hallucinations reduce anti-cheat to the bot design problem of imitating human behavior. David Durst's Stanford PhD research explores techniques for creating bots that imitate human behavior. Visit his blog for research on future imitation techniques that may improve hallucination behavior.

© 2023 Activision Publishing, Inc.