34

Human Enemy AI in *The Last of Us*

Travis McIntosh

34.1 Introduction

In the previous chapter, we discussed the overall design philosophy and the AI techniques behind the Infected. In this chapter, we will discuss a question that relates to the human opponents in *The Last of Us*. When we started prototyping the human enemy AI, we began with this question: *How do we make the player believe that their enemies are real enough that they feel bad about killing them?* Answering that one question drove the entire design of the enemy AI.

Answering that question required more than just hiring the best voice actors, the best modelers, and the best animators, although it *did* require all of those things. It also required solving an AI problem. Because if we couldn't make the player believe that these roving bands of survivors were thinking and acting together like real people, then no amount of perfectly presented mocap was going to prevent the player from being pulled out of the game whenever an NPC took cover on the wrong side of a doorway or walked in front of his friend's line of fire.

To begin with, our enemies had to be dangerous. If they acted like cannon fodder, the player would treat them like cannon fodder, so the player had to feel like each and every human they encountered was a threat. They also needed to coordinate, or at least to appear to coordinate. A roving band of survivors needs to work together to survive, just as Joel and Ellie must work together, and without some sort of coordination, they would appear subhuman. They also needed to care about their own safety. These were not suicide bombers. They were survivors. They should be as careful with their own lives as the player would be with theirs.

They needed to make good choices about where to go and when to go there, and more than that, they needed to be intelligent about *how* to get there. When they lost the player, they needed to communicate that fact to each other in a way that would be obvious to the player, and when their friends died, they needed to notice.

The design of *The Last of Us* also called for completely dynamic gameplay. Rarely were we guaranteed the location of the player when combat began, and at any point, the player could force the NPCs into a brand new setup from a different location. This meant that little of the NPC behavior could be scripted by hand. Instead, the NPCs had to be able to understand and analyze the play space, then adapt to the actions of the player.

Putting these concepts together with a number of visual and audio bells and whistles produced human enemies that could be enjoyable to play and just believable enough that, sometimes, every now and again, the player cared about who they were killing.

34.2 Building Blocks

Every AI system builds upon several key low-level systems. *The Last of Us* uses triangulated navmeshes, which are a fairly straightforward approach to navigation. The navmeshes are fairly coarse, and so we have a second-pass system that uses a 2D grid centered around every character on which are rasterized all static and dynamic blockers. This allows for short but very good paths, while the high-level system allows us to plan our overall route between distant points.

Pathfinding on navigation meshes is fast, especially utilizing the PS3's SPUs. We did between 20 and 40 pathfinds every frame utilizing approximately 4 ms of SPU time. Pathfinding through navigation maps (a fixed sized grid that surrounded each NPC for detailed path analysis), by contrast, was expensive enough that we limited the game to one per frame, with each NPC needing to wait for their turn.

One system that was new to *The Last of Us* was the *exposure map*. Early in the project, we found that in order for the AI to make good decisions about which path to take, we needed information about what the player could see and what he couldn't see. Exposure maps were our representation of this information.

We initially implemented visibility checks by casting rays toward the player from a number of different points on the NPC's current path and then using that information to decide whether the path was a good one or not. Unfortunately, this didn't work very well. Not only was it slow, but it didn't allow us to choose different paths based on the visibility information, which is what we really wanted. We then came up with concept of an exposure map, as shown in Figure 34.1. An exposure map is simply a 2D bitmap overlaid on the navigation mesh. In the exposure map, a one indicates visibility and a zero indicates occlusion.

In order to make calculating the exposure map fast, we embedded a simple height map inside of every navigation mesh. The height map used an 8-bit integer to represent the height of the world at every point on every navigation mesh. On the SPUs, we could then do very simple raycast out from the origin point in a 360° circle. Because we were working only in integer space and on the SPUs, we could parallelize this fairly easily. We then allowed the job to take multiple frames to complete. The end result is that we could continually calculate

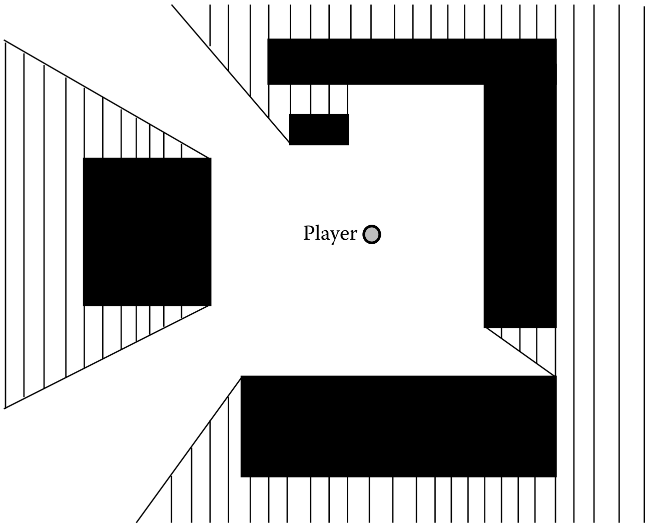

Figure 34.1

The exposure map covers everything an entity can see.

an exposure map for what the player could see, as well as an exposure map that showed everything any NPC could see as a simple bitmap using around 2–3 ms of SPU time.

Because the exposure map was, in fact, a bitmap, we often used it as a cost in our navigation function. The cost of traveling from one node to another was increased by integrating across this bitmap and multiplying by a scale factor. So, for example, we could use our standard pathfinding algorithm to find the best route from an NPC to the player, or we could add the exposure map as an additional cost and the path would minimize the NPC's exposure to the player's line of sight. Our first implementation of flanking was done using this exact algorithm, and it produced surprisingly good results in the static case.

34.3 AI Perception

One of the fundaments of AI in any game, especially a game focused on stealth, is AI perception. In the *Uncharted* series, AI used a few simple systems to determine their awareness of the world.

First, their sight we determined by a simple frustum and raycasts to check for occlusion. At the start of *The Last of Us*, we used this same system, as shown in Figure 34.2.

Problems arose in playtesting, however. Often, players right next to the NPC would be unseen, while NPCs too far away were noticed, simply because the cone we used for testing was not adequate to represent real vision. The fundamental issue was that, when close, we needed a larger angle of view, but at a distance we needed a smaller one. Using a simple rule—the angle of view for an NPC is inversely proportional to distance—we reinvented our view frustum to be much more effective, as shown in Figure 34.3.

Just because the player could be seen on one frame did not mean the NPCs had instant awareness of him. When an NPC saw the player, he would start a timer. Each frame the

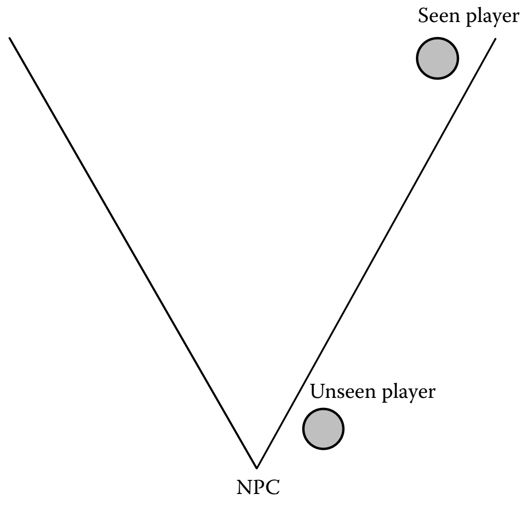

Figure 34.2

The simplest form of perception testing for NPCs had issues.

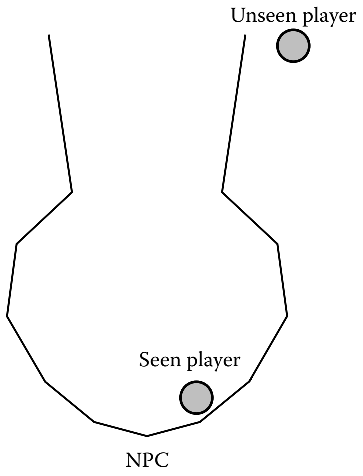

Figure 34.3

This more complex form of vision cone produced better results.

player was seen, the timer was incremented. Each frame the player was unseen, the timer was decremented. The player did not count as perceived until the timer reached a specified value (around 1 or 2 seconds for the typical NPC). When in combat, this threshold was much lower, and when the NPC had yet to perceive the player in its lifetime (i.e., the player is in stealth), this threshold was much higher.

We tried a number of different approaches to casting rays to check for occlusion. Our initial implementation involved rays to nearly every joint on Joel's body. Each joint was weighted, and the weighted average was compared against a threshold (typically 60%). If the sum was higher, then the player is counted as visible.

This produced mixed results. Although it was a decent approximation of when Joel was visible and eliminated the edge cases of being seen when just your head or finger was visible, players found it difficult to anticipate whether a given cover point was safe or not, because of the complexity of the casts.

After some experimentation, we found that we could use a single point on Joel's body instead. The location of that point would vary depending on whether we were in combat or stealth, as shown in Figure 34.4. If the player was in stealth, then the point is located in the center of the player's chest. If the player has engaged an NPC in combat, the point moved to the top of the player's head. This allows for player favoring perception in stealth while maintaining good visibility in combat.

Of note, the NPCs did not cheat with regard to knowing the player's location in most circumstances. When the player was perceived, the NPC would create an entity object with location and time stamp and then signal all other NPCs with the player's new location. If the player was not perceived by any NPCs, his location was never updated, instead remained in the previous location.

The combat cycle of *The Last of Us* was then as follows: The player showed himself, either visibly or by shooting his gun. The NPCs would surround him as best as they could and then began to advance on his position. If they had advanced as close as they could and they hadn't seen the player in a long enough period (10 s or more), a single NPC was chosen to approach the player's position to see if he was still there. If he was not, the NPCs then transitioned into the search behavior.

In our original focus tests, this combat cycle took far too long—nearly 2 min on average. Quite often, the player would have moved on long ago and would feel like the NPCs were not responsive. Our solution to this problem was to cheat. If the player moved further than 5 m from where the NPCs thought he was, he was considered to have snuck away, and we forced an NPC to approach his position immediately, so that they could enter search more quickly. This reduced the combat cycle to about 30 s and worked very well for pacing purposes.

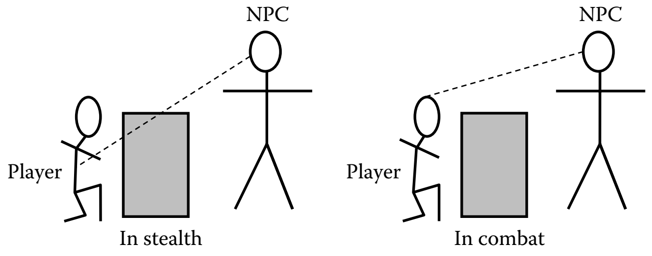

Figure 34.4

The left image shows that in stealth, the raycast point is placed on the player's chest, in order to favor the player. The right image shows that once combat has been initiated, the raycast point is placed much higher onto the top of the head.

34.4 Cover and Posts

Where to stand/take cover is one of the most fundamental decisions AIs need to make, and it is also one of the hardest. In order to properly rate and evaluate the best cover and open locations, you first need to gather a set of potential locations. We called these *posts*.

We had two distinct types of posts. The firsts were cover posts. These were gathered for each NPC in a radius around the NPC's location. Candidates were any cover spot facing away from the threat (all possible cover spots were precalculated by a tool analyzing the collision mesh). After we gathered the closest 20 cover spots for each NPC, we submitted them as a single job. Each frame, we would cast up to 160 rays to these different spots, each cover spot requiring 4 rays to determine whether the NPC could shoot and hit their target. When all of the rays for a given set of cover were complete, those covers where every ray was blocked were rejected, as shown in Figure 34.5.

We called the second type of post as an open post. Open posts were points in the world around the player. Primarily, these were used to find a good location for the NPCs to check the player's last known location when they were sent forward to begin a search. We again cast rays from these locations to the last known player position and rejected any whose raycast failed. In addition, we did a pathfind, limited to 20 per frame, from every NPC to every viable post for use in our post selectors.

Once we had a valid set of posts, we could then do analysis to select the location the NPC should use. Since every NPC behavior is significantly different, this used different criteria depending on what the NPC was doing at the time. We called these *post selectors*. Post selectors were defined in our LISP-based scripting language, with an example shown in Listing 34.1. We had 17 different post selectors when we shipped *The Last of Us*.

Each post selector defined what type of post it was interested in (in this case, cover and a number of different criteria). The criteria were different for each different selector and could easily be iterated on at runtime by reloading the script.

Of particular interest is the criterion **ai-criterion-static-pathfind-not-near-player**. Many times, during focus tests, players would complain that NPCs would rush forward to take cover. With some debugging, we determined that the issue was that a particular cover

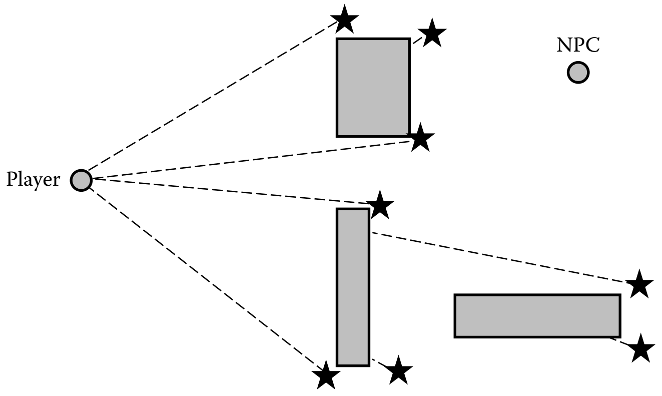

Figure 34.5

Any cover post whose raycast is blocked is rejected.

Listing 34.1. This script defines a post selector which locates a good place to hide.

```
(panic
   :post-type (ai-post-type cover)
   :criteria (ai-criteria
               (ai-criterion-path-valid)
               (ai-criterion-within-close-in-dist)
               (ai-criterion-available)
               (ai-criterion-static-pathfind-not-near-player)
               (ai-criterion-not-behind-the-player)
               (ai-criterion-distance
                  :curve (new-ai-point-curve
                            ([distance 3.0] [value 0.0])
                            ([distance 5.0] [value 1.0])
                          )
               )
             )
)
```

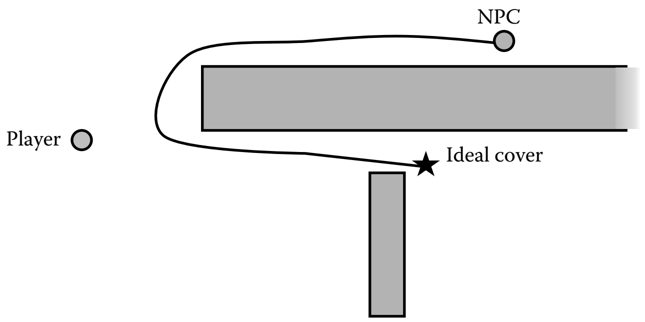

Figure 34.6

Sometimes the best cover could not be determined without a path. If the path to the cover required running toward the player for too long, as in this example, then the cover would be rejected.

was the best cover choice, but in order to pathfind there, the NPC would need to move toward the player, so that they could then move away again, as shown in Figure 34.6.

The solution was to write a criterion that used the pathfind information we had for every NPC to every viable cover. These paths were calculated in a round robin fashion and took about a 1/2 s to refresh, gathered at the same time as the cover-to-player raycasts. We would then analyze the path the NPC would take to each cover point, and if that path involved running toward the player for an extended period, then we would reject that cover.

There were, in fact, two major systems operating in parallel. The first system gathered pathfinding information, raycasts, etc. The second simply processed these data using the post selectors. Since the source data were all gathered on previous frames, these post selectors could be evaluated for very low cost on the SPUs. Each criterion would produce a float value normalized between zero and one; all of the criteria for a given post selector and a

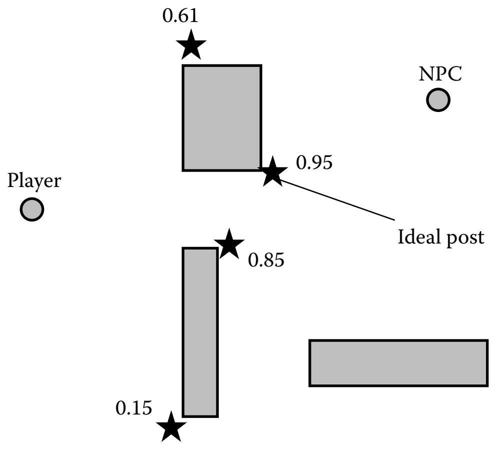

Figure 34.7

Every post is rated and the one with the highest number wins.

given post would then be multiplied together, and the resulting value would be that particular post's rating for that particular post selector. Next, all the posts would be sorted by rating, and the ideal post determined, simply by being the post with the highest rating, as shown in Figure 34.7.

Ratings would be different for each post per post selector and unique NPC. A post that was rejected for *panic* might be the ideal for *advance*. Note that all of the criteria for all of the posts and post selectors were evaluated continuously, so there was no delay when the NPC switched states—the ideal post for any given post selector was always available.

34.5 Skills, States, and Behaviors

The NPC's top-level states, which we called *skills*, were part of an FSM. Skills were prioritized, and each skill queried every frame whether it wished to run. The skill with the highest priority won. Examples of skills include *panic*, *advance*, *melee*, *gun combat*, *hide*, *investigate*, *scripted*, and *flank*.

Each skill included its own state machine. So, for example, *gun combat* could be in the *advance* or the *back away* state. Note that these were high-level states and didn't typically directly interface with the animation or pathfinding systems. Instead, a separate object known as a *behavior* would be pushed onto a behavior stack. Behaviors were on much lower level and were much simpler than the top-level states. Examples include MoveToLocation, StandAndShoot, and TakeCover.

34.6 Stealth

Stealth was handled by two separate skills. The *investigate* skill understood how to respond to a distraction sound and had a custom post selector. If the NPCs were in their standard scripted states—fully controlled by designer-created scripts that would tell them where to move, when to move, and even what audio dialog to play—then when an audio gameplay

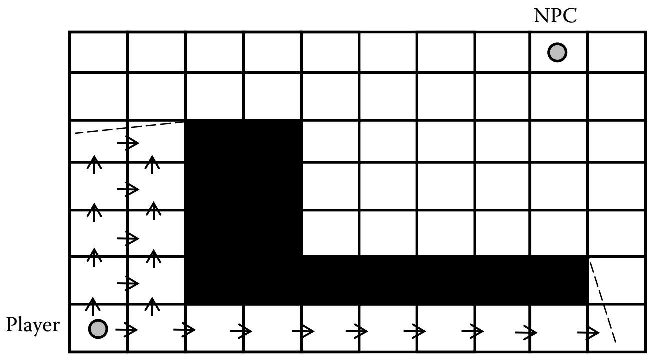

Figure 34.8

Search map locations spread out until within an NPC's line of sight.

event signaled a distraction, NPCs would request the role of an Investigator. Whichever NPC got that role would then walk to the ideal post as specific by the custom post selector and play an animation. If they found nothing, they would return to their previous location and pick up their script where it left off.

This was the case if the player had not been located yet. If the NPCs were already aware of the player, they entered the *search* state. *Search* involved procedurally partitioning the map and sending the NPCs to search it. Where to search was solved by the *search map*. The search map was a series of grid cells. If the player's location was known, all cells would be empty except the player's current location. Once the player's location was lost, however, the active cells would then bleed into their neighbors over time, and the potential location of the player would spread out to cover the map, as shown in Figure 34.8. Using the exposure map, any cells currently visible to an NPC would be cleared each frame. The result was a grid of cells that represented, roughly, the potential locations of the player from the NPC's perspective, who could then search in a relatively intelligent fashion.

34.7 Lethality

Games can create threatening enemies in a few ways. Enemies can take a lot of damage—in *The Last of Us* this broke immersion since you were supposed to be fighting humans. The number of enemies could be high—this directly contradicts our goal of making the player care about each and every kill. The enemies could deal a lot of damage—a possibility. The enemies could be very good at being hard to hit—another possibility.

We began by focusing on a concept we called *lethality*. Lethality meant that if a single enemy could kill the player, then every shot was frightening. One of the simplest and most successful things we did was make every shot the player received play a full body hit reaction. This meant that getting shot would not only deal significant damage but also take away control while the animation played out. In fact, it was the loss of control that most affected players. Those few moments of helplessness meant that every shot became a punctuation mark, a pause in the flow of the action that didn't let them forget their mistake.

Another way we made enemies lethal was by making sure to provide a threat whenever possible. This meant whenever an NPC had the ability to shoot the player, they would always choose to do that. With that said, it was only necessary for one NPC to be shooting the player at any given time; all other NPCs could spend their time taking cover, flanking, etc.

What this meant was that we needed a way for NPCs to coordinate with one another. We created a system we called the *Combat Coordinator*. The Combat Coordinator was simply a global object that managed each NPC's role. The roles include *Flanker*, *Approacher*, *Investigator*, *StayUpAndAimer*, and *OpportunisticShooter*.

Whenever a particular NPC desired a given role, they called the `RequestRole()` function on the Combat Coordinator. If that role was available, the function returned success, the NPC called `AcknowledgeRole()`, and no other NPC could take that role until they released it.

The purpose of the *OpportunisticShooter* role was to make sure there was at least one NPC focusing on shooting the player at any given time. If any NPC was able to see and shoot the player from their current location, they requested this role. Once they had the role, they instantly began shooting the player. This greatly increased the lethality of the NPCs. Note that when an NPC had this role, they would instantly stop whatever they were doing—even mid animation—and blend to standing and shooting at the player. In earlier playtests, they were noticeably slow in transitioning to shooting, with the result that oftentimes the player would be almost completely untouched when rushing.

34.8 Flanking

The role of the Combat Coordinator was not simply to be a gatekeeper to a few conceptual roles. In some cases, the coordinator would only allow a single, ideal NPC to take a given role. The best example of this is the *Flanker* role. Each NPC would run a pathfind in every frame to determine their best flank route. Each flank route would then be rated based on cost, and the coordinator would choose an ideal *Flanker* for the frame. If any NPC requested to flank the player but wasn't the ideal *Flanker*, their request would be rejected. Sometimes, as in the case of the *OpportunisticShooter*, we simply wished for the role to be taken as quickly as possible, so we would simply assign the role on a first come, first serve basis.

Although we originally used the exposure map to determine flanking, in practice this produced a number of issues. Because the exposure map changed as the player moved, often the flank routes could vary wildly from one frame to the next, and a corridor the algorithm identified as unexposed last frame could become exposed very quickly if the player was just around the corner.

The solution was to use a cost function based on the current *combat vector*. The combat vector was simply the current direction of combat from the player's perspective, calculated by averaging the NPC positions weighted by any shots that had been fired recently. Given the current combat vector, the cost function for flanking a given NPC was a fixed shape in the direction of that vector, as shown in Figure 34.9.

The closer to the center line (the line directly in the path of the combat vector), the higher the cost for pathfinding. The result of using this cost function was that flanking paths immediately attempted to move a large distance to the side and come around from behind, which was precisely what the player expected. In addition, the obstacles in the way were immaterial to how the cost function was created, and we instead let the pathfinding handle finding the path.

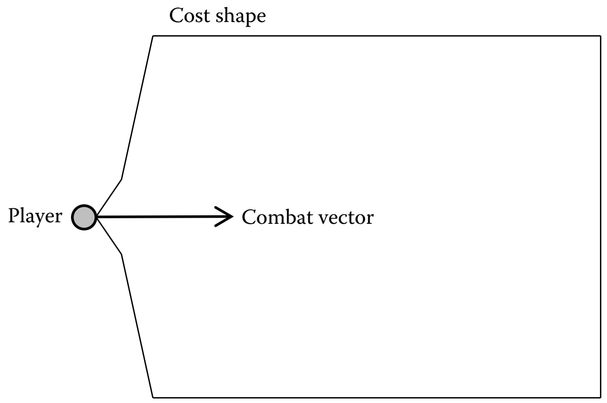

Figure 34.9

The combat shape rotates with the direction of the combat vector.

34.9 Polish

Once the AI decision making was in place, dialog could be layered in, animations could be polished, and setups could be scripted. Dialog in particular allowed the AI to communicate the decisions they make to the player, so that the NPCs could appear to be as intelligent as they sometimes were.

Then came focus testing, and more focus testing. Indeed, half of our implementation decisions were made in response to videos of players playing the game. In general, spatial analysis of the play space was perhaps the biggest win for us in terms of improving the AI's decision making. Combining that with the Combat Coordinator produced NPCs that made coordinated, informed decisions in a dynamic play space and appeared to be working together as a group.

34.10 Conclusion

In the end, we achieved our goal. The enemies had personality and had some sort of life, and most importantly, killing the enemies in *The Last of Us* was hard—not in the sense of difficulty but in a more emotional, more visceral sense. Because every enemy was a threat, every enemy felt more real and more alive, and because they felt alive, the player was able to build a small connection with them. The violence in *The Last of Us* was disturbing, not merely because of its graphic nature but because the player cared about the people they were attacking. The NPCs worked together, like real people. The NPCs fled and hid when threatened, like real people. The NPCs made good choices about where to go and how to get there, and most of the time they didn't destroy the illusion the player had immersed themselves in.

What we ended up with was not perfect by any means, but it answered the question of whether players cared about the people they were killing. They did.