31

Crytek's Target Tracks Perception System

Rich Welsh

31.1 Introduction

All our knowledge has its origins in our perceptions.

—Leonardo da Vinci

Perception is something that we all take for granted on a day-to-day basis. Without the ability to see, hear, smell, touch, and taste, we would know nothing of the world around us, nor would we have the means to communicate with one another.

One of the greatest challenges in writing good, believable AI is making "artificial intelligence" appear as "natural intelligence." As developers we have omnipotence in the virtual world that we create—passing information to AI agents without the limitations of mimicking human perception is as easy as calling a function.

```
pAIAgent->ThePlayerIsHere(position);
```

A scene in the film *The Matrix* features the protagonist being guided out of an office to avoid capture by a disembodied voice over a phone. With our access to the state of the game world, we could easily pass knowledge about targets and/or threats to AI agents in this way; however, once players realize that the AI are cheating, then they naturally tend to feel cheated. In this sense, "natural intelligence" could also be considered "artificial stupidity." The AI agents have access to any and all information about the state of the

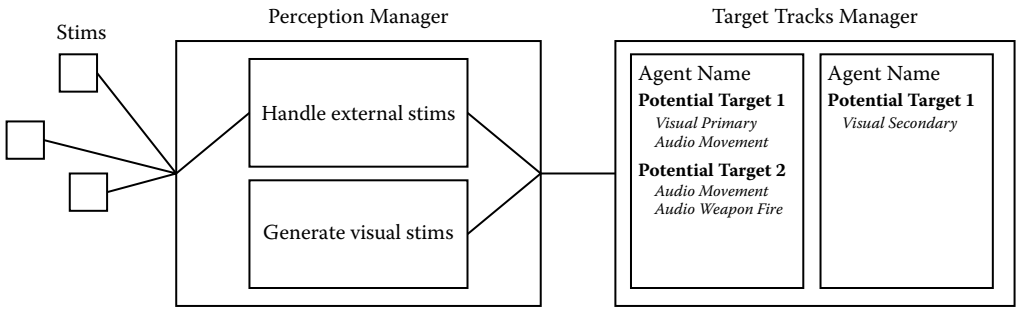

Figure 31.1

System overview. (The term *stim* is an abbreviation for "stimulus.")

game world at any time, but in order to make them act more believably, more human, we intentionally withhold this knowledge from them.

So if allowing the AI agents to directly query information from the game gives them too much knowledge, what other options are available? Quite simply, the best way to keep agents from knowing too much is to give them limitations similar to those of a human player's. Since the player determines the state of the game world through interpreting sights and sounds, the AI should attempt to simulate that as convincingly as possible.

While trying to simulate the perception of stimuli by AI, the primary focus of this system is target selection and prioritization. When faced with several potential targets (e.g., hostile agents, the player, etc.), which should an AI agent select to engage?

The system that Crytek implemented to both simulate limited perception and handle target selection is called the Target Tracks Perception System (TTPS). Originally written for *Crysis 2* by Kevin Kirst, the TTPS has proven robust and adaptable enough to be used again for all of Crytek's current projects, becoming an integral part of the *CryENGINE 3's* AI system.

31.2 System Overview

The TTPS comprises three main sections:

- *Stims*: These are data representations of perceivable events (or stimuli) that occur in the world. (See Figure 31.1.)
- *Perception manager*: This is responsible for handling, processing, and, in some cases, generating stims.
- *Target Track manager*: This manager is responsible for storing and updating every agent's perceptions over time, as well as containing logic for prioritizing targets.

31.3 Stims

Any in-game stimuli originating from a potential target (any current targets included) need to be handled in a consistent manner. By creating a simple structure that contains

data about perceivable events and a handler that can interpret them, we're able to create an interface that allows AI agents to start making sense of the world around them. In the TTPS, this structure is referred to as a stim (an abbreviation of the word stimulus).

Stims need to contain the following information in order to describe an event: The type of stimulus being generated (discussed later in this section), the entity ID of the source, the position or direction from which the stimulus originated, and a radius within which that stimulus can be perceived. Additional information such as a second entity ID can be useful as more special case events are added to the game, though the majority of events can be described using the initial set of data outlined above. For example, a stim generated by footsteps would contain the type `AUDIO_MOVEMENT`, the position of the footstep, and the entity ID of the character that generated it. However, a stim generated for a box that was thrown will need the type `AUDIO_COLLISION`, the position in which the box collided with the world, the entity ID of the box, and the entity ID of the character that threw the box (in order to determine whether the action was performed by a friendly character or a hostile one).

Before stims even reach the perception handling logic, there must be a way to filter them to avoid unnecessary processing of unimportant stims. Not every agent is interested in every event, so ignoring those stims from the start is an immediate way of cutting down the amount of stim traffic that needs to be handled. Having agents subscribe to event types that they are interested in and comparing that to the event type of a stim makes this a very cheap test.

The second way that stims can be filtered is by looking at both the perception range of each agent and the perceivable radius that is part of the stim data. If the agent is too far away to perceive this stim, then it can be discarded.

Finally, in each agent's perception configuration, event types can be registered with flags such as "hostile only," allowing events of this type generated by friendly agents to be ignored. This can make quite a big difference in reducing the volume of stims that need handling, as in a lot of FPS games the only hostile target that AI agents will encounter is the player, meaning that any events generated by other AI agents will not be processed.

31.3.1 Types of Stims

With a well-defined stim structure and a simple interface to pass stims through to the perception manager, stims can be created from anywhere in your codebase. While the majority of stims that get generated tend to fall into the categories of sight and sound, there are some that don't fit either category and as such are a more special case. The three types are discussed in the following sections.

31.3.1.1 Sound

While initially it may seem like every sound would generate a perceivable event, generally there are only a few sounds that the AI would be interested in. For example, most environmental sounds are of no interest to AI agents—ambient sounds such as birdsong or mechanical whirrs add ambience to the game for a player, but they don't give the AI any useful information about potential threats and targets. On the other hand, aggressive sounds such as explosions and weapon fire are of high importance to the agents, as identifying the source may keep them from harm or perhaps allow them to come to the aid of a colleague in combat.

In *CryENGINE*, the code that handles weapon firing generates these stims in parallel to firing the weapon. Since the "fire" event is triggered from the animation system, this means that the animation, audio cue, visuals (muzzle flash, tracer, etc.) are synced up with the gunshot audio stim. Although it might be too late for the agent to react and dodge, it lends itself to a more realistic perception model. Explosions are handled similarly, with the code that is responsible for generating the explosion generating the stim as well.

Collision sounds are a more complex type of event. While they are generated in the same way as regular stims, the notion of a collision having hostility is difficult to represent. In these cases, we "cheat" by having both the entity ID from the source of the sound (the item that caused the collision) and an additional entity ID of the character that was responsible for the collision stored in the stim. For example, an AI agent bumping into a barrel would send a stim with the barrel's ID as the source, and the clumsy agent's ID as extra data. By doing this, the perception manager is able to use the secondary ID to determine if a friendly agent caused the collision (and thus potentially ignore the stim). If a player caused the collision noise (by throwing a prop or knocking something over), then the collision would be treated as suspicious by the AI, since the collision was generated by a hostile.

31.3.1.2 Sight

In order to tell whether an AI agent is able to see something, most games tend to use a raycast from the agent to the target. While this does test whether there's a clear line of sight between the two, it can start to get expensive very quickly. For example, in our games we try to limit the number of active AI agents to 16. This means that every agent can potentially see 15 other AI characters. In the worst case scenario, this could mean 120 raycasts being requested to check visibility between all the AI agents, even before the player is considered!

In the AI system used for *Crackdown 2*, each agent could register how hostile a target needed to be before visibility checks should be done. These hostility levels were *hostile*, *neutral*, and *friendly*. By having AI agents register an interest in only hostile targets, this meant that any nonhostile targets became invisible to the agent, dramatically reducing the amount of raycasts required. Should a target change hostility (for example, going from being a neutral to a hostile target), the AI system will start or stop visibility tests as required.

When originally developing the AI for *Crackdown 2*, every agent tested against every other agent and the player. This would mean that in an environment with 16 active agents and the player, 136 raycasts would be required. After the optimization of having agents only register interest in hostile targets, only 16 raycasts would be required (assuming all of the AI agents were regarded as non-hostile to one another).

Further optimizations can be made to the generation of visual stims. In the *CryAISystem*, every agent has a view distance and a field of view. A lot of unnecessary raycasts can be avoided by doing these much cheaper tests to see if potential visual targets are even within an agent's view cone before requesting a raycast.

31.3.1.3 Special Case

The remaining few stims tend to be events that you want to make your AI aware of, but don't fit under the normal categories of sight or sound. For some of these events, you can effectively treat them as a dog whistle—create a sound stim without playing any audio and send that to the perception manager. These stims are then just handled in whatever way you need them to be.

Table 31.1 Examples of events that could generate stims

Stimulus type	Stimulus name	Description
Visual	Primary FOV	An agent is visible within the agent's primary FOV
Visual	Secondary FOV	An agent is visible within the agent's secondary FOV
Visual	Thrown object	An object that was thrown has been seen moving
Visual	Dead body	A body has been seen
Audio	Movement	Target has made sound while moving, e.g., footsteps
Audio	Loud movement	Target has made a loud sound while moving, e.g., landing from a fall, footsteps while sprinting
Audio	Bullet rain	Bullets are passing nearby
Audio	Collision	A physics collision has occurred
Audio	Loud collision	A "loud" physics collision has occurred
Audio	Weapon	A weapon has been fired
Audio	Explosion	An explosion has occurred

For stims that can't be treated as a sound, extra data is usually required. An example from our games is *bullet rain*. When bullet rain is occurring, the AI don't necessarily know the point of origin (or the hostility, though we pass the shooter's ID with the stim so that we can add the bullet rain to the appropriate *target track*—as explained in Section 31.5 Target Tracks); however the direction the bullet rain is coming from is known. As such, this type of stim needs to be handled slightly differently to a sound, having the agent react to being under fire without knowing the shooter's position immediately (Table 31.1).

31.4 Perception Manager

The perception manager is the middle management of the TTPS. It provides a single point of entry for stims into the system while also doing some specific case handling and stim generation of its own. Since stims can come from anywhere in the codebase, having this single entry point means that only one interface needs to be opened up to the rest of the game. Stims all share the same basic structure, which means that further encapsulation is possible by having only a single function exposed within that interface. After this point, the TTPS can become a black box to the rest of the code base, with a minimal interface consisting of functions for registering stims and querying the best current target for a given agent.

As mentioned earlier in the article, the perception manager is responsible for not only receiving external stims, but also generating some internally. As it's a centralized place to forward valid stims to the target tracks manager, it makes sense to put this logic here.

The stims that are generated from within the perception manager are all visual. Having a list of all active AI agents, the perception manager can iterate through each in turn, testing to see whether that particular agent can see any of the known observables (in the case of the *CryAISystem*, the list of known observables is a list containing all active characters, including players). If an observable is within the view distance and FOV for the active AI, an asynchronous raycast is performed. (As raycasts are expensive to perform and the visibility tests aren't urgent enough to require the results within the same frame, requesting deferred raycasts is perfectly acceptable.) On receiving the result of the raycast, a clear line of sight means generating a visual stim and passing that to the target tracks manager.

Having sight stims generated in such a way works fine for the majority of cases, but doesn't allow agents to have a peripheral vision. In order to accomplish that, we use another set of values for a secondary view distance and FOV. If an observable is within this secondary range of vision but not the primary (and still receives a clear raycast), then a secondary visual stim is sent to the target tracks manager. This stim has both a lower peak perception value (this value is used for stim prioritization when selecting a target, and is explained further in Section 31.6 ADSR Envelopes) than the primary stim and a longer attack time. By having a longer attack time, a target that is in the peripheral will take longer to identify than one that is in the primary FOV. Should the target move from the peripheral FOV into the agent's primary FOV, the primary visual stim will take priority over the secondary. These values are explained in more detail in a later section.

31.5 Target Tracks

Each agent in the game world needs to keep track of any targets that they identify. This is important when developing the behavior of the agents, as it can be used to identify which target to prioritize in a situation where several are present.

Once a stim has been sent from an event and passed through to the target track manager, that target has been perceived by the agent and becomes tracked. When keeping track of a target, any and all stims received (and converted into envelopes, as explained in Section 31.6 ADSR Envelopes) from that target are then stored together in a single *Target Track*. This container of envelopes represents all the perception that a particular agent has of a single target. By storing the envelopes in this way, we can make *Target Tracks* responsible for the envelopes that they contain—updating the perception values over time and eventually removing them once they have expired.

Each *Target Track* will only remember a maximum of one envelope per event type. As each *Track* is associated with a specific target, new envelopes received with an event type equal to that of an existing envelope in that *Track* will simply update the outdated one.

By using the highest envelope value within a *Track* as the overall value of that *Track* (and therefore the value for that *Track*'s target), finding the best target for an agent is as simple as choosing the *Track* with the highest value.

31.6 ADSR Envelopes

Once the stims have been filtered by the perception manager, the data used to test their validity isn't required by the target tracks manager. The only information that is needed at this point is the source of the stim. Rather than store stims directly, the target track manager creates ADSR envelopes (the terminology is based on the system used in music synthesizers; see Figure 31.2).

When a stim is initially received, an envelope is created and given a *perception value*, which will be used to compare the importance of this stim against others. This value is ticked over time, remaining constant as stims of the same event type from the same source continue to be received. Once stims stop being received, the event is no longer perceivable and as such the value starts to decrease.

Each event type can have a different peak value. This means that if a single target (e.g., the same agent or player) is the source of multiple different stimuli, some will have a

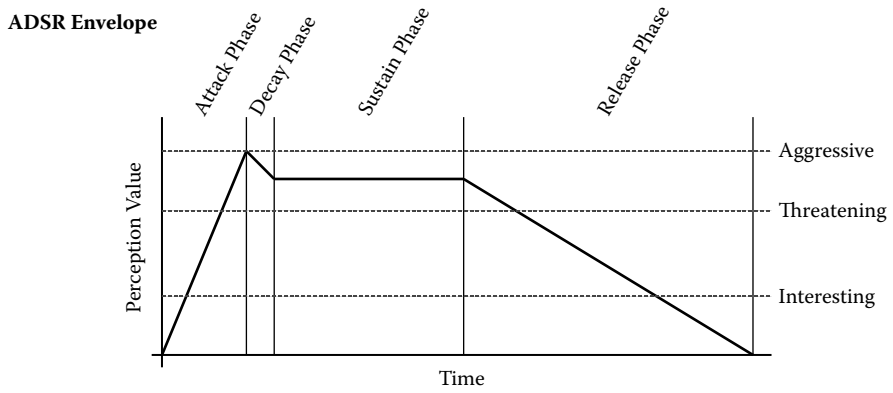

Figure 31.2

Graph demonstrating how the perception value for an ADSR envelope changes over time.

heavier weighting towards target selection. This peak value is an arbitrary number defined in a configuration file, but should be balanced across all event types. For example, in our current stim configuration, footsteps peak at a perception value of 25, weapon sounds peak at 50, and primary FOV visual stimuli peak at 100. This means that if one target can be heard firing a weapon and also seen, then the visual stim would be considered the most important event that's currently being perceived.

31.6.1 Envelope phases

ADSR envelopes are broken down into the following four phases.

31.6.1.1 Attack

The attack phase has three parameters, ignore time, peak value, and time. When a new stim is received, an envelope is created and started in the attack phase. During this phase the value of the envelope will rise from 0 to that peak value over the specified amount of time. The ignore time is the delay from after a stim is generated to when the AI will perceive it. This is usually set to 0 as most stims want to be registered immediately; however, in some special cases (such as faking speed of sound delays) it can be useful.

31.6.1.2 Decay

The decay phase only takes time as a parameter. After the peak of the attack phase has been reached, the envelope value can drop slightly over the time specified in the decay phase. The value that the envelope decays to is specified in the sustain parameters. This decay is so that new stims have a chance of being recognized as a higher priority over stims that have been sustained for a while.

31.6.1.3 Sustain

The sustain phase takes a fraction of the peak value. Once the decay phase has finished, the envelope will be sustained at the perception value equal to the fraction specified of the peak. This phase remains active for as long as the stim is still valid, and as such the

envelope value will remain constant throughout. By having a sustain fraction of 1, the envelope will remain at peak value until it enters the release phase.

31.6.1.4 Release

The release phase takes a time as a parameter. Once the stim is no longer valid, the envelope enters the release phase. For a sound this is immediately after the sound stops and for a visual it is when stims for that visual are no longer being received (usually because line of sight with the source has been lost). During the release phase, the envelope value slowly decreases until it either reaches 0 or the stim becomes valid once more.

31.6.2 Modifiers

While the base value of the envelope can be used to determine target prioritization, all stims of the same type (for different targets) that are in the sustain phase will have equal weighting. With this in mind, modification multipliers are applied to the base envelope values in order to help further weight them based on user specified criterion.

The TTPS supports a whole range of multipliers for different stims. The most beneficial of the multipliers that we use is one based on distance; targets that are close to the AI get a multiplier of 1, whereas targets 50 m or further away receive a multiplier of 0. This multiplier scales linearly based on distance, that is, a target 25 m from the AI would have its envelope value multiplied by 0.5.

31.6.3 Pulses

Pulses are artificial boosts to an envelope's value that are triggered through code. In the example configuration below (see Listing 31.1), there is a pulse set up to add an extra 12.5 to the primary visual envelope's threat value, that will decay down to 0 over 7.5 seconds. To trigger a pulse, a signal is sent to the perception manager with the name of the pulse and the agent to whom it's being applied. The one in the example is used if the agent is chasing ("sticking to") a target. By sending a pulse, it helps keep the target currently being chased the best target, even if a second target is spotted in the primary FOV.

31.6.4 Threat Levels

Threat levels are used more for the agent's behavior than in choosing a target. Once a target has been designated as the best available option for an agent, the threat level of that "best target" can be obtained and the behavior responds accordingly. Threat levels are stored as a percentage of the peak envelope value, as seen in the example perception configuration file in Listing 31.1.

31.7 Data Driving Perception

Stims need to be converted to ADSR Envelopes for the perception manager, though different agents may want to weight stims differently when it comes to target prioritization. For example, should you have an agent that is blind, then visual stims wouldn't register at all, but audio stims would have much higher weightings. The way that we've chosen to do this is by having a set of configuration files for stim/envelope setup written in XML. (In the case where all AI agents want to have the same perception, only a single configuration file would be needed.) Listing 31.1 is an example of how a stim is configured in the TTPS.

Listing 31.1. A perception configuration file.

```
<Stimulus name = "VisualPrimary" attack = "2" hostileOnly = "1"
peak = "100.0" sustain = "1.0" release = "40.0">
    <Modifiers>
        <Distance value = "1"/>
    </Modifiers>
    <Pulses>
        <Pulse name = "Stick" value = "12.5" duration = "7.5"/>
    </Pulses>
    <ThreatLevels>
        <Aggressive fraction = "1"/>
        <Threatening fraction = "0.625"/>
        <Interesting fraction = "0.25"/>
    </ThreatLevels>
</Stimulus>
```

Having such a flexible perception system has been very useful during development; it has allowed us to more easily tune the AI for various stealth and combat sections of the game by changing parameters in the perception configuration file. For example, increasing the time of attack on a visual stim gives the player more time to lean out of cover and look around, or dash to the next cover location without the AI spotting him immediately.

Exposing these values to the designers in data is incredibly important as well. It allows them to experiment and rapidly iterate on the AI's perception model without needing a new build of the game every time or requiring a programmer to assist them.

31.8 Conclusion

While there are plenty of different ways to model perceptions and prioritize targets, the Target Tracks perception system has proven itself to be both robust and flexible over the course of several different AAA titles. Since it was originally written, there have been very few modifications to the system, the most notable being the recent addition of the Threat Levels in the configuration file. This addition was made almost exclusively to extract more information from the TTPS, rather than to change the underlying logic or flow of the system.

Having a perception management system that is well encapsulated with a clean, minimal interface helps you keep the generation of stimuli simple. When working in large teams, the single-function interface to the perception manager makes it very easy for the rest of your team to start adding stims as and when they need to.

A free copy of the CryENGINE 3 SDK for noncommercial use is available online [CryENGINE 12], which uses the TTPS for perception handling.

References

[CryENGINE 12] CryENGINE 3 Free SDK, 2012. Available at http://mycryengine.com.