26

Tactical Position Selection

An Architecture and Query Language

Matthew Jack

26.1 Introduction

Agent movement is arguably the most visible aspect of AI in any game, and this is particularly true of shooters. Choosing between positions—and generating those positions to consider in the first place—is critical to the success of these games. Not only is it key to an agent's effectiveness in combat, but it also visibly communicates his role and status in that combat. More generally in games, an agent's movement helps define his personality and often much of the core gameplay.

In this article, we describe a complete architecture for choosing movement positions as part of sophisticated AI behavior. We outline a query language for specifying those positions, consider how position selection integrates with our behaviors, and give some specific building blocks and best practices that will allow us to develop queries quickly and to the best effect. Performance is given special attention, both to ensure our results reach the final game and to allow us to employ more powerful query criteria. We also discuss techniques for handling group movement behavior and the tools required for effective development.

Techniques developed for the Tactical Position Selection (TPS) system, a CryEngine component used in *Crysis 2* and other upcoming titles, provide the core of this chapter [Crytek 11]. We supplement this with a number of approaches seen in other games, as well as promising directions for future work.

26.2 Motivation

Any system used for choosing movement locations faces design pressures from many directions. It must be flexible and expressive, as it will define the movement possible for our agents. It must be capable of rapid, iterative development, and it must include powerful tools, as these will limit the quality of our final behaviors. Finally, since it is used frequently and in reaction to the player, it must be a fast and efficient workhorse, delivering results within a few frames while always remaining within CPU budgets.

The core problem of tactical position selection comes down to the question that designers will ask you when they are working with your behaviors: "Why did he move here, when it's just common sense that he should move there instead?" It's a question you should be asking yourself as you test your AI, because it's the question that players will ask themselves too.

Modeling that "common sense" is what makes this such a tough problem. Indeed, many shooters decide to use triggers and scripting to orchestrate the movements of their AI, leaving it to designers to provide that human judgment in each and every case—and this can be highly effective. However, this is a time-consuming process not suited to all development cycles, inherently limited to linear gameplay, and unsuitable for open sandbox games.

With the right abstractions and efficient processing, we can describe that human intuition for the range of contexts that our agent will encounter, and strike a powerful balance between specification and improvisation. In the process, we greatly speed up our behavior prototyping and development and gain a versatile tool for a wide range of gameplay applications.

26.3 Fundamentals

At their core, systems of this kind typically take a utility-based approach [Mark 09, Graham 13], evaluating the fitness of a set of points with respect to the requirements of a particular agent in order to identify the best candidate. Sources for these points are discussed in the Generation section, but they may be placed by designers, generated automatically, or some combination of the two. Usually we will collect or generate points within a specified radius of our agent's current position before beginning evaluation.

Evaluation will first filter out unsuitable points based on criteria such as the minimum distance from a threat and the direction(s) from which they provide cover. It will then score the remaining points based on desirability criteria such as how close they are to a goal point or how exposed they are to a number of threats.

We then choose the highest-scoring valid point as the result and use this as a movement target for our agent. By combining various criteria in different ways and weighting their effect on the score, we can produce effective queries for different agent types, environments, and behaviors. We can also use the same system for other applications, such as targets to shoot at or spawn locations. Good examples of tactical position selection have been given in previous work [Sterren 01, Straatman et al. 06] and in this book [Zielinski 13].

26.4 Architecture

Our tactical position selection system comprises a number of subsystems, and it must also integrate with the rest of the AI framework. Figure 26.1 shows the overall structure of the architecture.

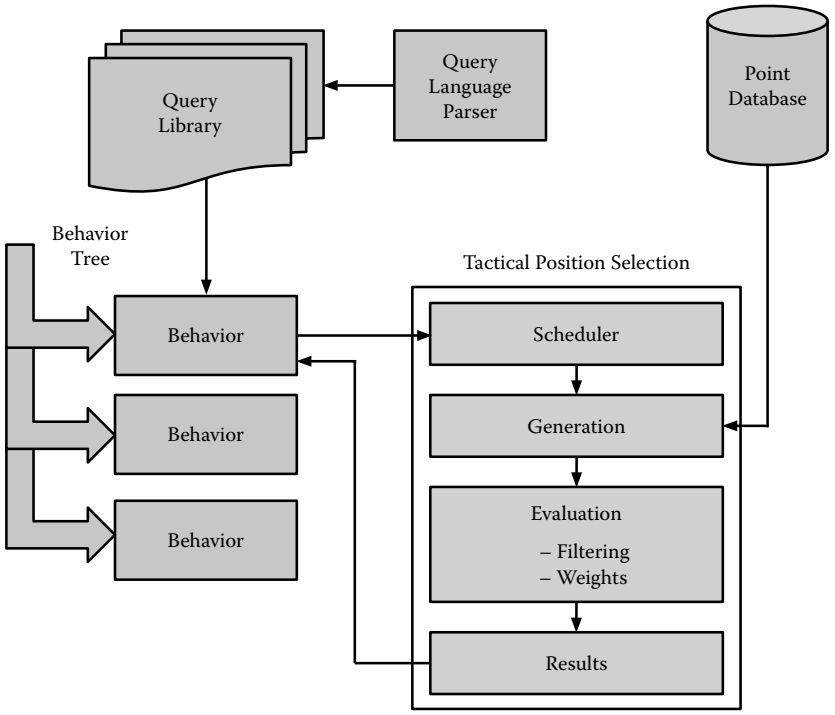

Figure 26.1

An architecture for tactical position selection.

The specification of individual queries can be thought of as the first step in the process of position selection, and a powerful *specification language* forms the first component in our architecture. We use this to build a *library* of context-specific queries to express each of our agent's behaviors in each of the environments it will encounter.

We employ a high-level AI framework, such as a behavior tree or finite-state machine, to provide the current *context*, which is the combination of the agent's current behavior and its current environment. We then use the context to select the appropriate query from the library. Should a query fail to find a valid location, this can serve as an input to the high-level framework, allowing it to respond should circumstances change.

We use a *database* of points, which may be manually placed or procedurally generated, to form the main input data for the evaluation process. A *scheduler* will often be required to help manage asynchronous queries and to amortize their cost.

Evaluation itself begins with *generation*, where points are collected from the database or generated on the fly, as required by the query specification, to form the set of candidate points for each query. We then *filter* those candidates with conditions, and apply *weights* to those that remain, as described above. In practice these two tasks may be interleaved for more efficient evaluation and spread over multiple frames by the scheduler. Finally, the full *results* of that evaluation may be displayed in the world graphically and summarized in the log for debugging before being reduced to the one or more points required by the query. In addition, each of these stages may be monitored by profiling systems.

26.4.1 A Query Specification Language

Expressing common-sense spatial reasoning is not something C++ or any common scripting language was designed for. The case for Domain-Specific Languages (DSLs) has been eloquently argued at the Game Developer's Conference [Evans 11] and here a powerful means of expression allows us to focus on the intrinsic problem and achieve rapid iteration. The key attributes of a successful query language for position selection are:

- *Abstraction*—We gain a lot of power if we can reapply existing criteria (keywords) in new ways.
- *Readability*—The intent of a query should be understood as easily as possible by developers.
- *Extensibility*—It should be easy to add new criteria to the language.
- *Efficiency*—The language itself should not add overhead to the evaluation process.

Many DSLs are built on top of existing languages. When developing the query language for tactical position selection within Crytek, we exploited Lua tables to provide the basic structure, but then parsed the content into a bytecode-like format so that no Lua was used by the system at runtime. The following is a simple example in that syntax.

Listing 26.1 shows a query that is made up of two subqueries, called "options." The first option is preferred but, should it fail, subsequent options will be tried in the order that they are specified before failure is reported. In this example, the first option will collect hidespots effective against the current attack target within a radius of 15 m around the agent, discarding any that are less than 5 m from the agent and discard any point closer to the attack target than the agent himself. Of the points that remain, it will prefer any hard cover that is available to any soft cover and prefer the closest point. The second option

Listing 26.1. A simple example in the Crytek query language syntax.

```
Query_CoverCompromised_FindNearby =
{-- Chiefly for use when our current spot has been compromised
    {   -- Option 1
        -- Find hidespot from our target a short distance from us
        Generation = {hidespots_from_target_around_puppet = 15},
        Conditions = {min_distance_from_puppet = 5,
                      canReachBefore_the_target = true},
        Weights = {softCover = -10, distance_from_puppet = -1.0},
    },
    {   -- Option 2 (fallback)
        -- Move away from target, taking a position in the open
        -- but preferring any that blocks line-of-sight
        Generation = {grid_around_puppet = 10},
        Conditions = {min_distance_from_puppet = 5,
                      max_directness_from_target = 0.1},
        Weights =  {visible_from_target = -10,
                    distance_from_puppet = -1.0}
    }
}
```

(the fallback) generates a grid of points to move sideways or away from the target and prefers to block Line-of-Sight (LOS). Both options share the goal of moving to a new, safe location at least 5 m away. Thus, this query might be used in reaction to a grenade landing at the agent's feet.

The Generation section of each option specifies the source of our candidate points. The Conditions section contains the filters that should be applied during evaluation, which must all pass if a point is to be valid. The Weights section tells the AI how to score the fitness of those valid points.

Each line in those sections is a Lua table entry comprising a string and a value. The string, in the syntax of our DSL, is formed from a sequence of keywords joined by underscores, which is trivially parsed back into the keywords themselves. Usually the most important keyword comes first—for example, `hidespots` or `distance`—and specifies the evaluation or generation method that we are going to apply. This can be referred to as the *criterion*.

Criteria often take an object, which allows the same method to be applied in different ways. For example, rather than having a `distanceFromTarget` keyword, we allow the same keyword to be used with many different objects, such as `puppet` (the requesting agent), `referencePoint` (general-purpose position), `player` (in single-player contexts), `leader` etc., and in particular `target` (the agent we are attacking if any). Objects are key to turning a set of parameters into a versatile query language. Some criteria do not take any objects—e.g., `softCover`, which is simply a property of the point.

The `min` and `max` keywords may prefix a criterion string, such as `distance`, that represents a float value. This changes the way the criterion is applied and also where it is used. Without the `min` or `max` keyword, it would simply be applied as a weight, but with these keywords we instead produce a Boolean result, depending on whether the actual value is above or below the specified limit. This allows us to use the same criteria in either the Weights or the Conditions section, as needed. In the example above, for instance, `min_distance_from_puppet = 5` is used to specify that points must be at least 5 m from the requesting agent.

There is also a set of "glue" keywords such as `from`, `to`, `at`, and `the`, which must be put between the criterion and its object, serving only to produce a more human-readable sentence—such as `distance_from_puppet`. The user is free to pick the word that forms the most natural expression in each case; the glue word is simply discarded during parsing.

Each keyword is registered with type markers as to where it can be employed. This allows us to detect and reject invalid queries during the parsing process, and to provide useful debug output.

While Lua was the most natural choice in the Crytek codebase, JSON, XML, or a simple custom format would also be reasonable vehicles for this kind of language. Queries can, of course, be changed while the game is running, in this case by editing and reloading the appropriate script files. A similar language has been implemented with a Kismet-style graphical interface under *Unreal 3* [Zielinski 13].

Documentation of the Crytek TPS system can be found online [Crytek 11] and it is available for use in the *CryEngine Free SDK* [Crytek 12]. Should you try any queries from the article, note that the common keyword `target` is shortened here; the correct keyword is `attentionTarget`.

26.4.2 Contexts and Query Library

As we discussed in the Motivation section, our agents face a complex and dynamic world, and players expect the agents to consistently make judgments similar to the ones that they themselves would make. One approach to achieving believable, reactive movement behavior across the range of environments in our game is to build a very complex "über-query." However, the problem becomes much more tractable if we break it down into bite-size chunks, each consisting of a specific behavior (such as flanking, retreating, or providing covering fire) that we wish to perform in a specific environment (such as a dense forest or a postapocalyptic New York street). We refer to each combination of behavior and environment as a context, and we can form a library of such queries.

It is much easier to describe exactly where to move if we make assumptions about our environment. For example, while we might be able to tweak a query until it allows us to advance effectively both between trees in a dense forest and from car to car in a city street, if we consider them separately, then our separate queries become much simpler. In fact, with the right criteria available, query construction becomes very predictable to the developer, and with some thought good results can be achieved without spending a lot of time tuning the queries. We discuss this further in the Best Practices and Specific Criteria section.

26.4.3 Integrating with Behavior Selection

To make use of a library of queries specific to context—the combination of behavior and environment—we must be able to choose the appropriate context at any given time. This can be achieved by viewing each behavior as having a core movement query and incorporating the current environment type as part of our behavior selection framework. This maps well onto a "first-generation" behavior tree, such as that used at Crytek [Martins 11], consisting essentially of a decision tree with behaviors as the leaves; different environments can form branches of the tree. Alternatively, the same behavior might use different queries in different environments; which is best will depend how much behavior is shared between environments. In *Bulletstorm*, a FPS game from People Can Fly, a similar system was used with a more sophisticated behavior tree [Zielinksi 13], and it could also be done with other architectures.

Basic knowledge of our current environment can be based on simple high-level cues, such as a per-level setting, or designer volumes or triggers. However, the behavior selection framework can also receive feedback from the queries we run and use this both to guide the selection of behavior and to inform us reactively about our environment. This feedback takes the form of failed queries—queries that return no valid results—which can occur for a couple of different reasons.

One way failure could occur is that we have designed our query for a common case, and this is an unusual example. For instance, our behavior and thus our queries might be designed for a dense forest, but obstacles or a path could mean there are very few trees in the local area. This might be a level design problem, but we may choose to provide a fallback for such cases—a second "option" in the query.

The fallback might often relax the filters and/or extend the query radius, especially if we used that smaller radius simply for performance reasons. It might also use different generation criteria: perhaps generating some points without cover around us and looking for one that at least has blocked line-of-sight to our target. The fallback essentially hides

the failure from behavior selection and lowers our quality bar on this occasion, to get us smoothly past this tricky spot.

However, there is a more informative case: our context isn't as we thought, and what we are trying to do is wrong for this situation. If we have designed our query correctly, it has not simply ranked all the points in order of preference, but also eliminated all the points that were not appropriate. If, for example, we are flanking left around a target in a forest, failure indicates that there is no suitable location to our left—we have run out of cover or our way is blocked. We can acknowledge this—for instance, by flagging this state in our blackboard—in order to choose a different behavior or indeed, in a squad context, to inform allies that a group action may have to be aborted. In the Crytek system, a query specification can include a specific signal to send or blackboard state to set in the event of failure.

26.4.4 Forming a Database of Input Points

Most games will form a static database of hidespots for consideration, and these form a mainstay of the candidate points for evaluation. These may have been placed by designers—for instance, the oriented "hide anchors" of *Far Cry* and *Crysis*, which indicated a position for the point as well as having a directional cone from which cover was provided. They may be automatically embedded in certain objects, such as trees; manually embedded in static or dynamic objects, such as vehicles; or generated automatically around complex objects. All of these have been combined in Crytek titles—with *Crysis 2* introducing automatic generation based on designer hints [Martins 11]. Notably, *Killzone 3* pregenerated its hidespots with an automatic process [Mononen 11], while *Brink* had no such static database and relied entirely on dynamically generated points [Nelson 11].

In general, such a database allows more time to be put into offline processing or designer work and provides a cheap resource of reliable points as input to a TPS system. Of course, the database can contain other types of positions in addition to (or instead of) hiding locations. Locations representing choke points, doorways, sniper spots, vantage points, respawn points, etc. may also be used as candidates, using specific criteria to generate a particular type or to distinguish between them. This represents a rich source of designer hints about the environment.

26.4.5 Collection and Generation

The first part of evaluating any query is to form a set of candidate points, whether by collecting them from a database or generating them at runtime. Typically, our primary source is to collect them from the static database described above, using a criterion such as that in our example:

```
Generation = {hidespots_from_target_around_puppet = 15}
```

This finds hidespots from a target and illustrates three parameters that are key to this phase: the center of the query—here, the `puppet` object, the agent itself; a radius for collection/generation—15 m; and an object, which for this criterion represents the primary target from which we wish to hide—in this case, the `target` object.

The best basis for an efficient query is to generate points in the specific location or locations we're actually interested in and within as small a distance as possible; the choice of center and radius should be made with this in mind. Centering on the requesting agent is

not always the best choice: for instance, when entering close combat we might center it on our target or the center of our squad (as we will discuss in the Group Techniques section). A common technique for designer control is to restrict agent movement to an area—for example, to defend, patrol, or advance upon. We can provide generation criteria for this based on an associated point or polygon, rather than relying upon conditions.

If we provide a primary hide-target (in this case `target`, the current attack target), this enables a range of powerful and efficient generation criteria. In *Far Cry* and *Crysis*, hidespots around trees were neatly handled by generating a hidespot on the opposite side from the hide-target. These omnidirectional hidespots are also useful in that they were refreshed on each hide request, moving an agent around the obstacles as its hide-target moved. The hide-target also allows us to use directional hidespots effectively—immediately rejecting cover points that are not correctly oriented for this hide-target using a dot-product test.

As well as collecting from our database, we may generate points on demand—for example, as we do in our fallback option in the query above:

```
Generation = {grid_around_puppet = 10}
```

In this case, we generate the candidate positions in a 10 m square grid around the agent. These arbitrary points can then be analyzed for their potential as cover or other properties. For example, if there is no explicit cover nearby, we may still try to block line-of-sight to our target; we may simply want to escape an overriding threat, such as a grenade, or specifically to move out into the open.

Brink generated points in concentric rings to find all of its cover dynamically [Nelson 11]. We could simulate something similar with a query such as the following (which would find the nearest point to the agent which blocks LOS to the target):

```
Generation = {circles_around_puppet = 10},
Conditions = {visible_from_target = false},
Weights =    {distance_from_puppet = -1.0},
```

Cover rails, as used in *Crysis 2*, are a relatively recent development that avoids representing cover as discrete points and instead stores a path, anywhere along which an agent may take cover or move in cover. On demand, we generate locations on the rail for tactical position selection. This allows us to, for instance, generate at least one location at the closest point on the rail to the agent or to generate locations at an optimal spacing from other agents using the rail.

Given the benefits of these different kinds of points, the role of this phase is to collect or generate relevant candidates from all these possible sources and to present them in a uniform manner to the rest of the process. While explicit hidespots may come with properties such as flags for high or low cover, cover quality or direction, grid points will have none of these, but ideally we will otherwise be able to treat them just the same in our conditions and weights.

26.4.6 Filters

Filters (or conditions) are criteria that will check the validity of a point and may then reject it. We can think of them as logically forming a second phase of the process, but in practice

it can make sense to interleave them with weights for performance reasons (as we will discuss in the Performance section).

Filters may simply be a property of a point, such as whether it provides high or low cover, soft cover, or hard. They may also involve some kind of evaluation against an object or position, such as raycasts to check whether, from relevant firing stances, we would be able to shoot our target. We can require these criteria to be either true or false—it can be useful, for instance, to find points that are visible from a target or to find those that are not.

When points have been generated from multiple sources, some filters may not make sense on some points. For instance, the condition `providesHighCover` simply looks up a Boolean flag in a collected hidespot, while a point generated in the open has no concept of direct cover at all. The Crytek system handles this by criteria returning the value that makes the most intuitive sense—for instance, a non-cover point returning false for `providesHighCover`—but a system might also be structured to skip over criteria that are irrelevant to a given point.

26.4.7 Weights

Weights are criteria that make a contribution to the score of a point. In most such systems the value returned by the evaluation itself is multiplied by a user-defined weight, allowing the contribution to be balanced against other weights. In the Crytek system, the values returned by all such criteria are first normalized to the 0–1 range, which makes balancing more intuitive. The user-specified weights we multiply those values by are over any range and can be either positive or negative. The results are summed and the best point is that with the highest, or least negative, score.

Most commonly, criteria that return a continuous value, for instance representing distance, will be used as weights but Boolean criteria may also be employed, returning simply 0 or 1. This allows us to give a fixed advantage or disadvantage to points in an easily balanced manner, for instance based on high or low cover.

Many weights are based on distances, and here clamping to known limits makes an enormous difference to the ease of specifying a query. If we simply return the raw distance, say, from a goal point or the closest enemy, then the maximum value is simply unknown, and while we may tweak for a common case of, say, 0–20 meters, when less common cases occur, our scores may be quite unbalanced. We can avoid this problem by choosing an appropriate maximum useful range for our game—say, 30 meters—and clamping to this, normalizing all distance criteria within that range. This means that we can then confidently predict the range of values when choosing our multiplier and comparing those to the expected effect of other weights in the query.

To achieve this, each criterion that returns a continuous value has declared limits for its output, so that when used as weights all such criteria are normalized in output to the [0–1] range. This also facilitates some of the optimizations we will discuss in the Performance section. When `min` or `max` are applied to make it a condition, we do not apply these limits; instead, we use the original, unnormalized value.

26.4.8 Returning Results

We have considered our context and chosen the relevant query, generated the appropriate candidate points for that query, applied conditions to filter them down to those that

are acceptable, and evaluated their weights to score their fitness. We can now take the top points and return them as the result of the query.

In the ideal case, we are only interested in one point: the point that our agent will now move to. However, there can be good reasons to return multiple results, such as to help with hidespot contention, as we will discuss later. In general we should not need to return a set of points for a subsequent system to choose between; if our API or query language is flexible enough, we should be able to do this within our system. Further, each point returned forces us to fully evaluate its criteria and lose the potential optimizations that we will discuss in the Performance section.

We can widen the applicability of the system by returning more than just a position vector. For example, we might return the type of point (i.e., cover point, point generated in the open, etc.), and any type-specific metadata. If the point is a hidespot from a database, we can specify this and give its unique ID, useful when marking the hidespot as occupied. In the Crytek system, points can be generated at the position of all entities of a specific type, and with these we return the original entity ID. Regarding results as objects in the world, rather than just points, allows us to consider reusing the architecture for a host of other AI and gameplay applications: from choosing which of our squad mates to signal for assistance, to prioritizing between opponents in target selection, to finding nearby objects to interact with.

26.5 Best Practices and Specific Criteria

For great results from your tactical positioning system, you don't just need a good architecture, you also need powerful criteria that fit the task at hand, and you need to use those criteria effectively. Here, we discuss some criteria and approaches that have proven effective in the development of games at Crytek or elsewhere.

26.5.1 Weights and Balancing

Queries can be built by using a large number of weights and tweaking and tuning them until your desired results are seen. However, more rapid and robust development can be achieved by preferring to use criteria as conditions rather than weights, by making use of fallbacks, and by further breaking down our queries to be more context-specific.

For example: when we want to flank left, rather than weighting all points according to "leftness," just invalidate any point that is not left of your current position; rather than tweaking a tradeoff between hard/soft cover and distance, write an option that only considers hard cover, then a fallback that only considers soft.

Keeping the number of weights down—ideally to one or two—results in queries with predictable results for given contexts and saves a lot of tuning time.

Effective criteria will also help you keep your queries simple—we examine some examples now.

26.5.2 Directness

When approaching a goal, an obvious basis for our query would be a weight based on inverse distance to the goal:

```
Weights = {distance_from_referencePoint = -1.0}
```

However, in many cases we do not wish to reach the goal directly, but to approach it in a series of hops—for example, ducking from cover to cover along the way. When this is the case, pure distance to the goal quickly becomes hard to work with and to balance other weights against. It is encouraging us to get as close to the goal as possible, when what we're really looking for is a succession of carefully selected waypoints. In other words, what we really need is a measure of progress towards the goal, with distances to be specified separately.

Directness is a measure of how much closer a point will get us to a goal for the distance traveled.

$$\text{directness} = \frac{\text{distance from agent to goal} - \text{distance from point to goal}}{\text{distance from agent to point}} \tag{26.1}$$

Using Equation 26.1, if we are 50 m from our goal, a point that is 10 m away from us and 45 m from the goal will score 0.5 on directness. This measure is very predictable: the closer to a straight line to our goal, the higher the score, regardless of distance. We can ignore distance and simply request the point closest to a straight line to the goal:

```
Weights = {directness_to_referencePoint = 1.0}
```

Even simpler, rather than using it as a weight, we can make it a condition by specifying that it should just be positive, so we will always make progress towards our goal, however small, or that it should be at least 0.5, so that for every 10 m we move, we will get at least 5 m closer to our target. These are easy to reason about and orthogonal to other criteria we might introduce, so we can apply it to existing queries without long sessions of retweaking.

```
Conditions = {min_directness_to_referencePoint = 0.5}
```

Further, this simple measure of `directness` actually offers a lot of power beyond goal-seeking. We can, of course, retreat from a goal in the same way by using a negative weight or requiring a negative directness:

```
Conditions = {max_directness_to_referencePoint = -0.5}
```

However, we can also get quite different behavior from the same criterion. By making a condition that the weight must fall in a set range around zero—for instance [–0.1, 0.1]—we specify locations where, by moving to them, we will have made no real progress towards or away from the target. This is a great basis for flanking, which we might specify as the following (note that this query does not imply any particular direction—which we can easily add):

```
Generation = {hidespots_from_target_around_puppet = 15},
Conditions = {min_directness_to_target = -0.1,
              max_directness_to_target = 0.1},
Weights =    {distance_from_puppet = -1.0}
```

Directness can also bring out zigzagging. In one case, core to gameplay on a Crytek project, designers desired agents to approach the player rapidly through forests, dashing from cover to cover, often without stopping. If they approached directly, their movements

would be very predictable and offer too easy a target. Here, we can specify directness both as conditions with a maximum value and as a negative weight:

```
Generation = {hidespots_from_target_around_puppet = 15},
Conditions = {min_directness_to_target = 0.5},
Weights =    {directness_to_target = -1.0}
```

Thus we take a route that is as indirect as it can be, while always making minimum progress of 1 m towards the player for every 2 m traveled, with irregularly placed trees and player movement providing a "random" factor. The result is zigzagging movement converging rapidly on the player.

26.5.3 CanReachBefore

If an agent runs towards a hidespot only to have an opponent occupy it before him, it can hurt us twice: first by the agent appearing to lack any anticipation of his opponent, and second by leaving him stranded in the open and abruptly changing direction towards other cover.

This is easy to prevent with a couple of provisions. The simplest is to focus on our current primary opponent and discard any point closer to our enemy than to us, effectively assuming that both agents will move with the same speed. We can implement this as a simple condition, such as:

```
Conditions = {canReachBefore_the_target = true}
```

This rule of thumb is so pervasive to good behavior that in the *Crysis 1* hiding system it was not optional; you might make it a default in all queries.

This focuses on just one opponent. *Far Cry* and *Crysis* also exploited perfect knowledge of which hidespots all agents are moving towards, never picking a hidespot that another agent had already claimed. Here prominent visual debugging is advised, since queries from one behavior or agent will have non-obvious interactions with others.

26.5.4 CurrentHidepoint

Often, it can be useful to generate just the point that we have already chosen, either to adjust it or to verify that it is still good enough.

Sometimes the exact position of a point will be determined on demand—for instance, the hidespots for trees in *Crysis*, mentioned earlier, that would always be generated on the opposite side from the hide-target. If we are using cover rails, we might just want to adjust our position on the rail, to reflect the movement of our target or to maintain separation from a nearby squadmate who is sharing the rail.

Once we have started to move towards a point, we usually want to show a certain amount of inertia behind that decision: changing direction if it becomes slightly less desirable, or, as a slightly better spot becomes available, will often look artificial to players. However, if circumstances change considerably—perhaps as a grenade lands or an enemy movement leaves a position clearly exposed—we do need to react.

Hence, once we have chosen a point, we can periodically check it with a simplified query that generates just that single point, where weights are irrelevant and just the critical conditions are employed—making it cheaper, simpler, and more tolerant than the original

query. This query is verifying that the point is still "good enough." When this does fail, we may fall back to the full version of the query to pick a new point with the same criteria, or go back to our behavior tree to reevaluate our context. Should the situation change dramatically—for instance, due to the presence of a grenade—we rely upon the behavior tree to switch us to a different behavior and thus a different query.

Bulletstorm used a similar validation of chosen hidespots, but implemented it as a specific type of criteria within a single query, rather than separate versions [Zielinksi 13].

26.6 Group Techniques

So far we have discussed individual agents making the best decisions about their own movement. When we start to consider agents as members of groups or squads, or indeed companions, then new criteria can help us build their behaviors while coping with this added complexity.

26.6.1 Spatial Coherency—A Squad Center

In considering a squad, our most fundamental property is spatial coherency—that is, keeping the squad together. We can do this by defining a center for the squad—for instance, its current average position—so that each member's queries can specify they should stay close to that point. We can implement this by reapplying our existing criteria with a new object, `squadCenter`.

Perhaps the most obvious approach is to weight our queries to prefer points inversely proportional to their distance from the center.

```
Weights = {distance_to_squadCenter = -1.0}
```

This is very useful when our squad needs to converge (perhaps reacting to being surrounded) or, conversely, to spread out (perhaps under threat of mortars). However, as a building block, it has the problem of being a weight that must be balanced against any others in our query. Coherency is not a behavior in itself, but a property of all the other behaviors we would like to show—hence, ideally we would specify this in a manner orthogonal to other criteria.

A simple way to achieve that is to simply limit the maximum distance. Our members are then entirely free to move within that radius of the squad center, which will travel with the squad as its members progress [Champandard et al. 11]. Note that we can achieve this very efficiently by centering our collection/generation on the squad center and setting the radius appropriately.

```
Generation = {hidespots_from_target_around_squadCenter = 10}
```

The results have some very nice properties. If some of our squad is trailing behind while moving towards a goal—for instance due to a scattered starting configuration or some members taking long paths around obstacles—then those at the front will find their queries return no new valid points, as the only ones that progress towards the goal are outside of range of the squad center. In this case they should handle this "failure" by waiting, until those at the rear catch up and the squad center moves forward. This gives rise to a

loose leapfrogging behavior without any explicit consideration of the relative position of squad members.

Here, we generally assume that the goal we are progressing towards is a common one across the squad—and by choosing that goal's location appropriately we can present that as the squad advancing or retreating, or even give designers direct control of the goal, allowing them to direct the squad according to scripted events.

However, individuals can also choose a completely different behavior and move independently for a time without disrupting the group. If, for example, an individual is avoiding a grenade, or collecting nearby ammo, he may well ignore the group coherency criteria in that query. While he may appear to leave the group, he still contributes to the squad center and so they will not proceed too far without him, allowing him to catch back up.

26.6.2 Hidespot Contention

Whenever we operate within a group while maintaining spatial cohesion, there will be more contention for available hidespots. This can present both behavioral and architectural problems.

Whenever individuals are responsible for the choice of location, they can only optimize choices, given their own preference and the points currently available. Hence, the rifleman who occupies the ideal location for a grenadier, or the agent who takes the nearest available cover for himself, leaving his squadmate to run awkwardly around him to reach any at all.

This can be resolved by performing queries at the squad level rather than the individual level, maintaining a central set of points that are assigned to squad members with the whole squad in mind. This approach was used effectively for the group behaviors in *Crysis*. However, this can lead to tightly coupled behaviors that are hard to extend.

There is also the problem that if we have queries in progress on multiple agents at the same time—for instance, running in parallel on multiple threads—they may both try to take the same point, which we must somehow resolve. One way to do this is to return multiple results for each query, so there are other points to fall back to. As we will note in the Performance section, returning the full list of results would mean we cannot take advantage of some important optimizations—but one or two extra results will usually be sufficient and can be generated with only an incremental increase to the cost of the query.

26.6.3 Companions and Squadmates

An NPC designed to interact closely with the player is one of the biggest challenges in game AI. When the NPC shares movement locations such as hidespots with the player, their movement choices must become like cogs in a machine, meshing closely with the unpredictable movements of the player if we are to avoid a painful clash of gears.

In first-person and third-person games, the problem is especially acute as we face the conflicting demands of remaining in the player's field of view to keep her aware of our movements and activity, while trying to avoid getting in the way or stealing a location from the player; in shooters, we also have to keep out of the player's Line-of-Fire (LOF) … the list goes on. There are some specific criteria that can help us in finding solutions:

`cameraCenter` returns a value indicating where this point is in the players' view frustum. It interpolates from 1 for the dead center to 0 at the edges and negative values

off-screen, allowing us the control to specify positions towards the edge of our view or just off-screen. Variations on this can specifically prefer positions to the left or right.

`crossesLineOfFire` takes an object to specify whose LOF to consider, usually the local player. Our implementation uses a simple 2D line intersection test, comparing the line from the specified object to the candidate position against the forward vector of the player, although more complex implementations are certainly possible.

Using these two criteria we can form a simple squadmate query:

```
Generation = {hidespots_from_target_around_player = 15},
Conditions = {crossesLineOfFire_from_player = false
              min_cameraCenter = 0},
Weights =    {distance_from_player = -1.0,
              cameraCenter = -1.0}
```

This query is for a context where we already have a target we are attacking and try to find a cover point from that target, insisting that it must not cross the player's LOF to get there and that it must be on-screen, but as close to the edge of the player's view as possible and balancing that against staying as close to the player as possible.

Of course, this is just a starting point in a long process of design and development. For instance, the player's view changes constantly as he looks around and our companions should not run back and forth to remain in view; nor does the player's forward vector at any given moment always represent his likely line-of-fire. We can begin to address these by averaging these vectors over time, by considering a "golden path" representing likely movement flow through the level and by considering the targets the player is likely to fire at.

As we try these new ideas, we can add them as new criteria, exploiting the rapid prototyping capabilities of our query system to help us try out all their permutations quickly and effectively.

26.7 Performance

In developing a system of this kind, work on its performance must go hand-in-hand with the expansion of its capabilities and its use. The more efficient the system and the more sophisticated its handling of expensive criteria, the more freely we will be able to use the system and the more powerful criteria we will be able to add to our toolset. Tactical position selection is one of the most productive uses to which we may put our AI cycles, but unless we have performance under control, our ideas will not make it into a shipped product. Here, we discuss ways to reduce and manage the cost of our queries.

26.7.1 Collection and Generation Costs

When collecting points from your database, an efficient lookup structure designed for cache coherency is essential on consoles. Without this, it is quite possible for the cost of collecting the points to exceed the cost of their evaluation. Crytek games have used various schemes for storing points and hidespots, including storing them in the navigation graph, using spatial hashes, and separating them by type. We could also maintain only the set of points relevant to this section of the game, swapping them out as we might an unused part of the navigation graph.

In generating points dynamically or verifying the cover of collected points, prefer to generate the candidates as cheaply as possible and defer any expensive tests—for instance, for occlusion from a target, for later in the evaluation process where we may be able to avoid their cost altogether.

26.7.2 Minimizing Raycasts

Raycasts are commonly used for a host of purposes, including establishing that you are hidden from a particular target, or establishing whether you could shoot at a particular target using a particular stance. In many games, they are the dominant cost of position selection or of the whole AI system, and as such deserve special consideration.

A physics raycast operation is always expensive and will generally traverse a large number of memory locations. Should it be performed synchronously, this will be painful in terms of cache misses, cache trashing, and possibly synchronization costs with any physics thread; on consoles this is quite prohibitive. Asynchronous operations allow batching and smooth offloading of the work to other cores such as PS3's SPUs. TPS systems thus should prefer an asynchronous API, even if only for the efficient handling of raycast-based criteria.

We may be able to form points in our database such that we can assume occlusion from some directions and avoid raycasts completely. In a static environment we can use a very simple approach of implied direction for every hidespot, such as the "hide anchors" in *Crysis*. We can then simply test if a target falls within a cone from the hidespot's direction. In a dynamic environment we may also need to periodically check that the cover object is still valid. Taking this a step further, *Crysis 2*'s cover generation system actually maps the silhouette of associated cover objects and is able to remap them upon destruction [Martins 11]. This allows us to test occlusion from a point against this nearby geometry without raycasts.

When minimizing an agent's exposure from multiple targets we will need to be able to consider cover from a number of angles and also consider what other geometry might be blocking line-of-sight, which means that we are likely to require raycasts. However, in many games such attention to exposure is not required, and in many contexts, in terms of believability, digital acting, and gameplay, it can be more effective to focus on hiding well from the single opponent we are actively engaged with, rather than attempting to hide from several.

One source of raycasts that is very hard to avoid is those which verify we can shoot at the target effectively from a location, usually by aiming over or around our cover geometry. After all, when we request a place to hide, what we usually really mean is a firing position with cover; we must verify that we can shoot at, or near, our target if gameplay is going to be interesting.

We need our system to handle the raycasts that we must perform as efficiently as possible. This means that we should leave raycast-based conditions until late in the position selection process, so that other conditions will screen out as many candidates as possible. We should also limit the number of raycasts that we initiate per frame, so as to avoid spikes, and allow the raycasts to run asynchronously. In many systems, raycast tests are treated specially for this reason—and this may be sufficient for your needs. In the Crytek system and in this discussion, we treat them as one example of a range of expensive and/or asynchronous operations we would like to support in our criteria.

26.7.3 Evaluation Order

The simplest thing we can do that will make a big difference to performance in general is to pay attention to the order we evaluate our criteria. The rule of thumb for evaluation order is:

- *Cheap filters first,* to discard points early
- *Weights,* to allow us to sort into order
- *Expensive filters*, evaluating from highest scoring down, until a point passes

It is reasonable to class many criteria as simply "cheap," because a great many criteria consist of simple Boolean or floating-point operations, with cache misses on data likely the dominant cost. These filters should be run first, as they can be used to quickly eliminate candidates, thus avoiding more expensive criteria. A few criteria, on the other hand, may be many orders of magnitude more costly (such as raycasts). We should defer these tests as long as possible, so as to minimize the number of candidates we have to run them for. By discarding points quickly with cheap filters, then evaluating weights, then performing expensive filters on the remainder in order from the highest-scoring to the lowest, as soon as a point passes we can stop evaluation, returning the correct result but often at much reduced cost.

We note that there is little we can do about expensive weights in this approach (such as a relative exposure calculation or an accurate pathfinding distance); also, that in the worst case we will fully evaluate all of our points before a valid one, or none, is found.

26.7.4 A Dynamic Evaluation Strategy

If we had the ability to skip expensive weights, as well as conditions, this would allow us a greater freedom in the development and use of such criteria. We might measure relative exposure to multiple targets, for example, or make use of accurate path lengths rather than straight-line distance. Of course, we would want to ensure that we return the same results as if full evaluation had been employed. Here, we present an approach that allows this.

We first note that in general we do not need to have the final score of a point to establish that it is better than any other may be. To illustrate this, consider two criteria that return floating-point values, normalized to the 0–1 range: A, assigned a weight of 2/3 in this query and B, assigned a weight of 1/3. If one point gains the full score on weight A, while the second scores less than half of that, then there is no reason to evaluate weight B on these two points; the highest score that B could provide would still not cause the second point to score more highly than the first.

Based on this observation, we should focus our evaluations on the point which currently has the potential to score highest out of all those considered. In order to do this, we need to know how many criteria have been evaluated so far on each point, and the maximum score that each point might achieve. We can use a struct to hold this metadata and the point itself:

```
struct PointMetadata
{
    TPSPoint point;
    int evalStep;
    float minScore, maxScore;
}
```

We employ a binary heap [Cormen et al. 01] to maintain a partial sort of these structures as we proceed, based on the `maxScore` value. Often used for priority queues, heaps ensure that the maximal element is kept at the top of the heap with a minimum of overhead. In a binary heap, removing the top element (`pop_heap`) is an O(log *n*) operation and, usually implemented in a single block of memory such as an STL `vector`, they are also relatively cache friendly.

Nonetheless, we noted earlier that many criteria are inexpensive to evaluate, and in these cases the overhead of heap maintenance would be comparable to the evaluations themselves. Hence, we deal with these "cheap" criteria before forming the heap in a straightforward manner similar to that discussed in the previous section: we evaluate all of the cheap conditions and discard any points that fail, then we score the remaining points based on the cheap weights—that is, summing the products of their normalized return values and the multipliers (weights) specified by the user.

This leaves us with a reduced set of points upon which to evaluate the more expensive conditions and weights. The next step is to establish the minimum and maximum possible score that each remaining point might achieve when all of the weights have been evaluated. Since all weights will be normalized to return values in the [0–1] range, we can do this just by reference to the user-defined multipliers. We sum the values of the negative user-defined multipliers (weights) in the query to find the greatest amount they could subtract from the overall score. We do the same for the positive user-defined multipliers to find the greatest amount they could add to the overall score. We then go through each point and add these amounts to the actual score that was computed when we evaluated the cheap weights, finding the lowest and highest potential scores for that point—the `minScore` and `maxScore` metadata described above.

Having dealt efficiently with the cheap criteria and established potential scores, we create the heap from these structs, populated as described above, and all further evaluation is based on that data structure. We now look at the core evaluation loop in detail. The blocks of code in this section can be combined to describe the whole loop, but we have split it into parts so that we can explain each step in the process.

Each iteration begins by checking if we have exhausted all candidates, in which case this query option has failed. Otherwise, we take the top point from the heap and check to see if it has been completely evaluated. If it has, we have a final result and either return immediately (as below) or remove that point from the heap and continue evaluation to find multiple results.

```
while (!empty_heap(pointHeap))
{
    PointMetadata& best = pointHeap[0];
    if (best.evalStep > finalEvaluationStep)
        break;
```

We then check what the next evaluation step is for this point. There is a single defined evaluation order, further discussed in the coming text, which may alternate between conditions and weights. If the next criterion is a condition, we perform that evaluation and then either remove the point from the heap (should the condition fail) or advance to the next evaluation step (should it pass).

```
if (isCondition(best.evalStep))
{
    bool result = evaluateCondition(best);
    if (!result)
        pop_heap(pointHeap);
    else
        best.evalStep++;
    continue;
}
```

When the criterion is instead a weight, we make use of the "heap property." In a binary heap this dictates that the second and third elements, which are the two child nodes of the first element, will each be maximal elements of their subtrees within the heap. Hence, by comparing with these two elements, we can check if the *minimum* potential score of the top element, `best`, is greater than the *maximum* potential score of any other point. If this is the case, we can skip this and any other weight evaluations on this point—based on our observation at the start of this section—but we must still check any remaining conditions.

```
if (isWeight(best.evalStep))
{
    if (best.minScore > pointHeap[1].maxScore
        && best.minScore > pointHeap[2].maxScore)
    {
        best.evalStep++;
        continue;
    }
```

If that is not yet the case, we must evaluate the weight criterion for the point, lookup and apply the defined range to normalize it to the [0, 1] range, and then fetch the user multiplier that will determine its final contribution to the score.

```
    float value = evaluateWeight(best.point);
    float normalized = normalizeToRange(value, best.evalStep);
    float multiplier = getUserMultiplier(best.evalStep);
```

We then need to adjust the minimum and maximum scores for this point. Performing a weight evaluation has narrowed the range of potential scores, in effect resolving some of the uncertainty about the score of this point. Where the user-defined multiplier for this criterion is positive, a weight that returns a high normalized value will raise the minimum score by a lot and lower the maximum score just a little; conversely, a weight returning a low normalized value will raise the minimum score by a little and lower the maximum score by a lot. Note that in both cases, `minScore` goes up and `maxScore` goes down.

```
    if (multiplier > 0)
    {
        //A positive contribution to final score
        best.minScore += normalized * multiplier;
        best.maxScore -= (1 - normalized) * multiplier;
    }
```

```
        else
        {
            //A negative contribution to final score
            best.minScore -= (1 - normalized) * multiplier;
            best.maxScore += normalized * multiplier;
        }
```

With the effect of that weight evaluation applied, we now reposition this point in the heap since it may no longer have the maximum potential score. This operation is equivalent to a pop_heap operation immediately followed by a push_heap operation, removing and then replacing the top element. We advance to the next evaluation step and begin the next iteration:

```
        best.evalStep++;
        update_heap(pointVector);
    }//Matches if (isWeight(best.evalStep))
}//Matches while (!empty_heap(pointHeap))
```

In deciding the evaluation order of criteria we are free to interleave weights and conditions. Ordering them by increasing expense is a reasonable strategy, but we might instead evaluate the largest weights earlier and likewise conditions that in practice usually fail. There is scope for considerable gains in performance based on profiling and feedback.

This approach improves over the simpler approach in cases where we have expensive weights that we hope to avoid evaluating. Where we have no such weights, the evaluations it performs will be the same as the approach described in the previous section. In the worst case, when none of the points are valid, it may be able to perform better since the evaluation order can be more freely adjusted, as above. However, if we are to be sure of returning any valid point that exists, this worst case will of course require us to check at least enough conditions to discard every point regardless of the approach we take to do so.

When a query takes significant time to evaluate, due to expensive criteria or a worst-case evaluation on a large number of points, we must be able to spread evaluation over a number of frames. It may also be useful to share evaluation time between multiple agents, to avoid all active agents waiting upon a single slow query. The heap structure described is suitable to pause evaluation at any stage to continue later—as we will exploit in the next section.

26.7.5 Asynchronous Evaluation and Timeslicing

A synchronous approach, returning results in a single function call, is the simplest execution model and the most convenient for the developer, but as we place greater load on our system and provide more powerful criteria, synchronous queries can easily lead to unacceptable spikes in processing time. Further, certain criteria may depend on asynchronous operations—for instance, efficient raycasts, as discussed. As a result, we will almost certainly need to support the ability for queries to be handled asynchronously.

In order to better handle asynchronous queries, we employ a scheduler, which will keep track of which requests are pending and which are currently being processed. *Crysis 2* employed a simple first-come-first-served scheduler working on a single query at any time—however, a scheduler that shared time between agents would be a useful improvement. With a scheduler in place, we can timeslice our evaluation, checking evaluation time periodically during execution and relinquishing control when our allocated budget for the frame has expired.

The Crytek system performs the entire generation phase and evaluates all of the cheap weights and conditions in one pass, before checking evaluation time. Since, by design, these stages are low cost, we do not expect to greatly overrun the budget by this point, and only synchronous operations are supported so we will not need to wait for any operations to complete. Once this is complete, the system forms the heap, which is compatible with timeslicing.

When we continue evaluation of the heap, we evaluate a single criterion—either a weight or a condition—on the point with the highest potential score, and then check the elapsed time before continuing with the next or putting aside the heap until next frame. Since by definition all of the criteria evaluated in this heap stage are of significant expense, the overheads of checking the CPU clock and heap manipulation are acceptable.

The same infrastructure provides the basis to handle criteria that use asynchronous operations such as raycasts. Whenever we reach one of these we start that deferred operation in motion before putting the heap aside much as if we had used up our timeslice. On the next frame we reevaluate the heap in much the same state as before—except this time we find our deferred operation is waiting for a result, which should now be available. We can then complete the results of that criterion on that point just the same as if it has been synchronous—and proceed to the next.

The latencies resulting from criteria that employ asynchronous operations are significant and could limit their use in our queries. There are a number of ways we can address this. First, we should evaluate asynchronous criteria last for each candidate point, so that we will avoid them if we can. Second, if we have remaining time available we could continue evaluation on another query, increasing our throughput. Finally, we could speculatively start deferred criteria on a number of other points near the top of the heap at the same time as the first.

26.8 Tools

As with all such systems, our TPS system is only as effective as the tools we build for it and the workflow we develop around it. The query language is one such tool, but we also need to be able to visualize the query results if we are to effectively develop and tune queries and track down problems.

The Crytek system allows us to render query results for a specific agent in the world in real time. A sphere is drawn for every candidate point, with color indicating their status: white for the highest-scoring point, green for a point that passed all conditions, red for a point that failed a condition, and blue for a point that would have been only partially evaluated. The final score is displayed above each sphere. At the same time, a representation of the parsed query is output to the log, to confirm which query was used and also to allow us to double-check what criteria were used in the evaluation.

While we did experiment with grading the colors by their relative score, we found that the differences in score were often small—though significant—and so very hard to judge visually.

Some criteria have custom debug rendering. For instance, criteria that include raycasts draw the corresponding line in the game world, and more complex dynamic generation criteria could draw indications of where and how they considered generating points.

BulletStorm used a similar visual debugging approach, but also annotated every invalid point with a short text string indicating the specific condition that failed [Zielinksi 13].

26.9 Future Work

There are some avenues for future work that are particularly promising.

As the industry standardizes on efficient navmesh solutions, a family of powerful new criteria become affordable in a TPS system. *Accurate path distances* for individual agents between locations would provide subtly improved movement choices across the board and a greater robustness to dividing obstacles such as walls and fences, compared to the Euclidean distances we take for granted. *Generation of points* could be done only in nav polys connected to our agent, ensuring that points generated in the open, such as the grid method described, are always reachable. *Navigational raycasts* could be very useful for maintaining line-of-sight within a group and might in some cases make a cheap approximation for full physics raycasts.

Game environments continue to become richer, more dynamic, and increasingly they make use of procedural content generation. All these trends represent a scalability challenge to conventional search methods. There is increasing interest in the use of *stochastic sampling*, seen in games such as *Love* [Steenberg 11], which could scale better. This comes at a risk of missing obvious locations, with a cost in believability and immersion that is often unacceptable in current AAA titles, but such games might be tuned to make this very rare, or such points might be used to supplement a conventional database of points.

Finally, when working on a squad-based game, there is currently a stark choice between individual position selection, offering decoupled interaction between members of a group in different roles; and centralized position selection that can coordinate the allocation of points to the advantage of the whole group, but at the cost of tightly coupled behavior code. Work towards *coordination between decentralized queries* would be an area of special interest to the author.

26.10 Conclusion

Tactical Position Selection is a keystone of shooter AI and a potential Swiss army knife of AI and gameplay programming. When we break out of rigid evaluation methods and provide an expressive query language, we can drive a wide variety of behavior with only small data changes. By creating a library of queries specific to the environment and desired behavior, and by considering best practices and making use of specific building blocks, we can keep queries simple and create them quickly. Feedback from our query results to our behavior and query selection allows our AI to adapt when circumstances change.

We have discussed how we can architect such a system, from query specification to final result and how we can integrate it with our behaviors. We have considered performance and tools for debugging that help ensure we make the best use of the system in the final shipped product. In particular, we have referred to details of the system used in *Crysis 2*.

Acknowledgments

The author would like to thank Crytek for their assistance with this article and AiGameDev.com for the valuable resources that they provide. Thanks also to the other developers involved, in particular, Kevin Kirst, Márcio Martins, Mario Silva, Jonas Johansson, Benito Rodriguez, and Francesco Roccucci.

References

[Champandard et al. 11] A. Champandard, M. Jack, and P. Dunstan. "Believable Tactics for Squad AI." GDC, 2011. Available online (http://www.gdconf.com/).

[Cormen et al. 01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001, pp. 127–144.

[Crytek 11] Crytek. "The Tactical Point System." http://freesdk.crydev.net/display/SDKDOC4/Tactical+Point+System, Crytek, 2011.

[Crytek 12] Crytek. "Crydev.net." http://www.crydev.net. 2012.

[Evans 11] R. Evans et al. "Turing Tantrums: AI Developers Rant." GDC 2011. Available online (http://www.gdcvault.com/play/1014586/Turing-Tantrums-AI-Developers-Rant).

[Graham 13] David "Rez" Graham. "An introduction to utility theory." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Course Technology PTR, 2009.

[Martins 11] M. Martins. Paris Shooter Symposium 2011. Available online (https://aigamedev.com/store/recordings/paris-shooter-symposium-2011-content-access.html).

[Mononen 11] M. Mononen. "Automatic Annotations in Killzone 3 and Beyond." Paris Game/AI Conference. 2011. Available online (http://aigamedev.com). Slides available online (http://www.guerrilla-games.com/publications).

[Nelson 11] J. Nelson. Paris Shooter Symposium 2011. Available online (https://aigamedev.com/store/recordings/paris-shooter-symposium-2011-content-access.html).

[Steenberg 11] E. Steenberg. "Stochastic Sampling and the AI in LOVE." Paris Game/AI Conference. 2011. Available online (http://aigamedev.com).

[Sterren 01] W. van der Sterren. "Terrain reasoning for 3D action games." In *Game Programming Gems 2*, edited by Steven Rabin. Boston, MA: Charles River Media, 2002.

[Straatman et al. 06] R. Straatman, A. Beij, and William van der Sterren. "Dynamic tactical position evaluation." In *AI Game Programming Wisdom 3*, edited by Steve Rabin. Boston, MA: Charles River Media, 2006.

[Zielinski 13] M. Zielinski. "Asking the environment smart questions." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.