26

Guide to Effective Auto-Generated Spatial Queries

Eric Johnson

26.1 Introduction

Intelligent position selection for agents—that is, analyzing the environment to find the best location for a given behavior—has evolved rapidly as spatial query systems such as CryENGINE's Tactical Point System and Unreal Engine 4's Environment Query System have matured. Once limited to evaluating static, preplaced markers for behaviors such as finding cover or sniping posts, dynamic generation gives us the ability to represent a much wider and more sophisticated range of concepts. The ability to generate points at runtime allows us to sample the environment at arbitrary granularity, adapting to changes in dynamic or destructible environments. In addition, when used to generate a short-term direction rather than a final destination, we can represent complex movement behaviors such as roundabout approaches, evenly encircling a target with teammates, or even artificial life algorithms such as Craig Reynold's boids (Reynolds 1987), all while navigating arbitrary terrain.

Originally developed as a generalized, data-driven solution for selecting pregenerated points in the environment, Crysis 2's Tactical Point System (TPS) is now freely available to the public as part of CryENGINE, while Bulletstorm's Environmental Tactical Querying

system is now integrated into Unreal Engine 4 as the Environment Query System (EQS), making these techniques accessible to a massive audience (Jack 2013, Zielinsky 2013). As game environments grow increasingly complex, other studios are also adopting this approach with implementations like the Point Query System in *FINAL FANTASY XV* and the SQL-based SpatialDB in MASA LIFE (Shirakami et al. 2015, Mars 2014).

Designing effective queries is the key to maximizing the quality of agent position selection while dramatically reducing the amount of work required to implement and tune these behaviors. Done well, you can consolidate the majority of a game's position selection logic into a library of queries run on a spatial query system, rather than managing a collection of disparate and independent algorithms. However, the functionality of these systems has become increasingly sophisticated as they gain wider adoption, presenting developers with more possibilities than ever before. This introduces new challenges to use the array of tools and techniques at our disposal effectively.

In this chapter, we present a selection of tricks and techniques that you can integrate into your agent's queries to ultimately deliver higher quality, more believable behavior. Each component of a spatial query is covered, from sample generation to failure resistance, to improve the effectiveness of spatial queries in your project.

26.2 Overview

In modern implementations, a single spatial query generally consists of the following components:

- *Sample points*: Locations in the world which we want to evaluate in order to determine their suitability for a particular movement task.
- *Generator*: Creates the initial set of sample points in the environment. For example, one type of generator might create a 100 m 2D grid of points along the floor of the level, whereas another might create a ring of points at a radius of 10 m.
- *Generator origin*: The location around which we want to run the generator—for example, the center of the grid or ring of points that are created. Most often, the generator origin is either the agent itself or some target that it is interacting with.
- *Test*: Measures the *value* of a sample point, or defines an *acceptance condition* for it. For example, the sample's distance from the agent can be a measure of value, while its visibility to the agent's target can serve as an acceptance condition.
- *Test subject*: A location, object, or list of locations/objects that serve as the subject of comparison for a test. For example, a distance test might compare each sample point's location against the querying agent, its destination, the set of nearby enemies, recently discovered traps, etc.

To get an idea how these components work together, consider a scenario in which we need to implement a typical approach-and-surround behavior for a group of melee enemies (Figure 26.1). Our goal is to get them into attack range quickly while at the same time fanning out in a circle around the player. To accomplish this, we might begin by using a *ring generator*, using the player as the *generator origin* to create a set of *sample points* in range of our target. Next, by using a series of *tests* measuring the distance from each sample point to the player, the agent, and the agent's teammates (as *test subjects*), we can combine their

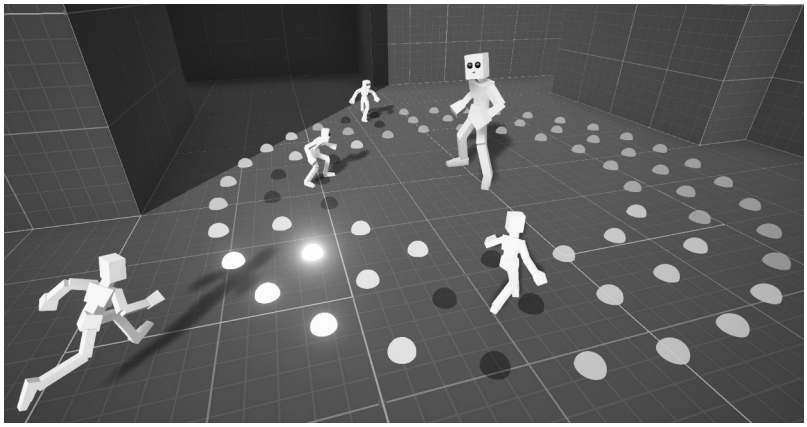

Figure 26.1

Four enemies using spatial queries to approach and surround the player.

value to find positions that get the agent closer to the player from its current location while avoiding areas occupied by teammates. A visibility test from each point to the player, used as an *acceptance condition*, can additionally discard destinations where the agent would be unable to see the player. Finally, the location with the highest total score across all tests is returned as the query result.

The remainder of this chapter assumes a basic familiarity with publicly available query system implementations such as TPS and EQS. For a more in-depth explanation of how an environment query is structured and executed, please refer to Chapters 26 and 33 of *Game AI Pro* (Jack 2013, Zielinsky 2013).

26.3 Generating Sample Points

The first step in selecting a useful destination for an agent is to generate a set of potentially viable locations to evaluate. When using pregenerated points this is trivial; we typically collect all marker objects in a given range and move on to the ranking phase. For dynamically-generated points, things are more complex as the generation method itself can heavily impact the quality of the final result.

26.3.1 Generation on the Navigation Mesh

The simplest method of dynamically generating a set of sample points is to create a localized 2D grid on the surface of the agent's environment. Although it is possible to use collision raycasts against level geometry to map out the level floor, this is not only computationally expensive, but the generated points may not be reachable by the agent (e.g., if they lie on a steep slope or narrow corridor). By sampling along the surface of the navigation mesh instead of the actual level geometry, we can both reduce generation cost and ensure that the sample position is reachable by the agent.

However, the overhead of finding the navmesh surface for a large number of sample points can still be significant. To be practical at runtime, we can further minimize generation cost by localizing our projection test to a limited set of navmesh polygons that match

as closely as possible the area to be sampled by the generator. The caveat is that there are multiple valid techniques we can use to define this subset, and the one we choose can significantly affect the outcome of the query. For example, two common approaches are either to gather the set of navmesh polygons within a bounding box centered on the query origin, or to gather the navmesh polygons within a given path distance of the query origin, and then to generate points only on those polygons. The bounding box approach is straightforward to implement, but can generate positions that, measured by path distance, are distant or even unreachable (Figure 26.2a). For behaviors such as finding ranged attack locations, this can be a good fit. Using path distance on the other hand ensures that the origin is reachable from all positions, but ignores locations that are spatially indirect, even if they are physically close (Figure 26.2b). Thus the bounding box approach may work better for behaviors that only require line-of-sight (such as ranged attacks), whereas the path distance method is preferable for behaviors dependent on spatial distance, such as following or surrounding.

Other options exist as well. For instance, we can merge both techniques, relaxing the path distance requirement to find reachable points within a given radius even when the path to that location is long and indirect. For example, given a radius r, we can gather all navmesh polygons within some multiple of that radius (say, $2r$). Then, during generation, we can eliminate sample points with a linear distance greater than r, giving us better coverage over an area while still ensuring a path to the generator origin.

After we have selected the most appropriate method for gathering navmesh polygons, we have a few different methods for generating samples that will impact the effectiveness of our final query:

1. *One-to-one mapping*: Some common navigation libraries, such as Recast/Detour, provide functionality to find the nearest point on the navmesh, given a point and bounding box. We can thus run a search over the gathered polygons at each (x, y) position on the grid, with some reasonably large z value, to verify that a point lies on the section of the navmesh gathered in the previous step. Although efficient, a weakness of this technique is that if your environment has vertically overlapping areas, such as a multi-floored building or bridges, only one level will be discovered (Figure 26.2c).

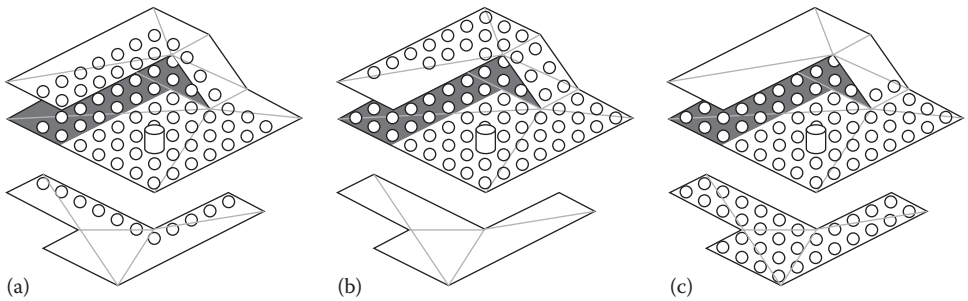

Figure 26.2

(a) Navigation mesh-based sample generation restricted by bounding box distance. (b) Sample generation restricted by path distance. (c) Sample generation restricted to a one-to-one mapping from the original grid coordinates to the nearest location on the navmesh.

2. *One-to-many mapping*: A second technique is to use a vertical navigation raycast over the gathered polygons at each (x, y) position, generating multiple hits along the z axis whenever we pass through a gathered navmesh polygon. Here, we trade efficiency for accuracy, handling multi-level terrain at the cost of some performance.

26.3.2 Generation Structure

Grids are not the only way to arrange our generated sample points. A custom generator can produce items along walls, arranged in rings, hexes, along waypoint graphs, inside Voronoi cells, or countless other configurations depending on the situation. This decision is important; a poor layout can introduce bias into your query, causing agents to cluster around or avoid certain locations. For tests that are intended to create a smooth scoring gradient, such as distance from a target, it is immediately noticeable when this distribution becomes uneven as agents will begin to approach targets only from specific directions, or settle into locations at specific intervals from the target.

For example, consider a query that wishes to find a location that is as close to an agent's target as possible, while leaving a 3 m buffer zone around the target. With a grid-based approach, we can first generate a set of sample points around the target, discard those closer than 3 m away, and rank the rest based on their distance from the target. Unfortunately, this exposes a problem, as illustrated in Figure 26.3. Depending on the desired radius, the closest points to the target invariably lie either on the diagonal or cardinal directions. As a result, agents not only cluster around four points, they may also approach the target at an unnatural angle to do so—that is, instead of moving directly toward the target to a point that is 3 m away, they will veer to one side or the other to get to one of the "optimal" points found by the search. In addition, selecting a grid layout for a circular query is intensely inefficient; a large portion of the sample points will either be too close (and thus

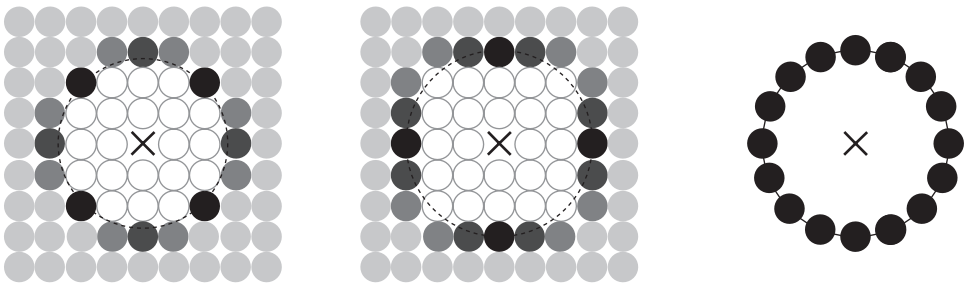

Figure 26.3

(a, b) Examples of diagonal and cardinal distance bias introduced by range tests over grid-generated points. (c) Elimination of distance bias by using a ring generator. Darker circles indicate higher priority positions. Empty circles indicate points that were generated but discarded, whereas light circles indicate positions that were ranked but have no possibility of being used.

immediately discarded by the distance test) or too far (and thus will never be selected, because a closer valid point exists).

In this instance, we can replace the grid generator with a ring generator, eliminating distance bias by guaranteeing that all points closest to the target are the same distance from the target. In addition, we gain an efficiency boost, as we need only generate a fraction of the sample points to perform the same test.

In our projects, this category of query was by far the most common. By changing approach/surround queries to use ring generators, agents selected more natural destinations, and the improved efficiency allowed us to enhance these queries with more complex sets of tests.

26.4 Testing Techniques and Test Subjects

Tests are the building blocks that allow complex reasoning about the environment, and are thus the most crucial components of a query system. Although projects invariably require some domain-specific tests, knowing how to combine and reuse simple, generic tests to produce complex results is the key to rapid development. For example, by only mixing and matching the two most versatile tests in a query system's toolkit, distance and dot product, we can support a surprisingly wide range of tasks beyond the common but simple "move within X meters of target Y" or "find the closest cover point between myself and the target" behaviors. Section 26.8 provides several practical examples of queries built with these two tests.

26.4.1 Single versus Multiple Test Subjects

Some query systems, such as EQS, allow a test to be run against multiple reference locations. By preparing specific concepts such as "all nearby allies," "all nearby hostiles," or "all agent destinations," we can add tests to our queries to improve the final result.

For example, a minimum distance test (Section 26.4.3) weighted against both ally locations and ally destinations can prevent agents from attempting to move not only into currently occupied locations, but also into locations that *will be occupied in the near future*. For agents that do not require advanced coordinated tactical movement, this single powerful addition can eliminate most location contention without the need to implement specific countermeasures such as point reservation systems.

26.4.2 Distance Test Scoring

When performing a test against multiple test subjects, we have a choice to make: What score do we keep? For example, is it better to record the shortest distance or the average? As shown in Figure 26.4, each can be used to express a different concept. Minimum distance helps us create local attraction or avoidance around the test subjects; this allows us to keep our distance from any area of the map occupied by a test subject. Conversely, average distance gives us the centroid of the subjects, useful for enforcing team cohesion by prioritizing samples within a specific distance from the centroid.

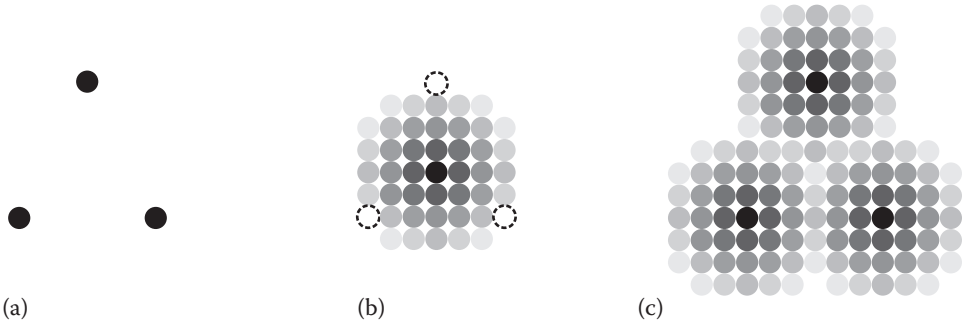

Figure 26.4

(a) Multiple targets used as the context of a distance test. (b) Score using average distance from targets. (c) Score using minimum distance from targets.

26.4.3 Dot Product Test Techniques

Within our project, this test is the most frequently used after the distance test, and these two as a pair have been used to express more movement concepts than all other tests combined. The dot product test measures the angle between two directions, allowing us to, for example, prioritize sample points in front of or behind an agent. By choosing our test subjects carefully, we can handle virtually any direction-related weighting by stacking dot product tests. Some examples:

- Testing the vector from the agent to a sample point against the agent's orientation allows us to prioritize locations in front of (or behind) the agent.
- Similarly, testing points against the agent's right vector instead of its orientation (forward vector) lets us prioritize locations to its right or left.
- We can use one of the tests above to prioritize a heading in front of, behind, to the left or to the right of the agent. However, using both together will prioritize the area where they overlap, allowing us to represent a diagonal heading instead.
- Testing against the world forward and right vectors, instead of the agent's, can give us prioritization along cardinal or ordinal directions.
- Using a target as the test subject, rather than the agent, gives us the ability to position ourselves in a specific direction relative to that target—for instance, to stand in front of an NPC vendor, to attack an armored enemy from behind, or to walk alongside the player in formation.
- For even more flexibility, we can accept an optional orientation offset in the dot product test itself: By applying a user-defined angle to the set of forward vectors above, we can prioritize points in any direction, not just the cardinal and ordinals.
- By defining both directions as vectors between two subjects, rather than the orientation of the subjects themselves, we can go even further:
- Comparing the vector from the agent to a sample point against the vector from the sample point to an agent's target prioritizes locations between the agent and the target. This provides us with locations that get us closer to the target from our current position, ranked by the directness of that approach.

- By using a sine scoring function (Section 26.5.1) over the same vectors, we prioritize locations where the dot product value approaches zero, generating destinations ranked by *indirectness*. While still approaching the target, these locations allow us to do so in a curved, flanking manner.
- Flipping the direction of the first vector (i.e., using the vector from a sample point to the agent instead of the agent to a sample point) reverses the prioritization, providing retreat suggestions ranked by directness away from the target (Figure 26.5).

We can even apply these concepts beyond actors in the scene. For example, ranking sample points based on the dot product of the vector from the camera to a sample point against the camera's orientation provides us with locations near the center of the screen (though potentially obstructed). Used with a minimum threshold and low weight, this can provide encouragement for buddy AI characters or other agents we want the player to see as much as possible.

26.4.4 Subject Floor Position

In action games where the player can fly, jump, or climb walls, an agent's target can easily become separated from the navmesh. When used as a generator origin, this results in the entire query failing, as there is no navmesh at the origin location to generate sample points around. On our project, we used two techniques to resolve this issue:

1. We provided a "Target Floor" test subject to supplement Target (the default). This modified version projected the agent's position down to the navmesh floor, if present.
2. We provided a "Closest Navmesh Point to Target" test subject, which scanned the immediate area when the target was off mesh.

Both of these techniques allowed agents to find a suitable location to approach the player when jumping or performing off-mesh traversal. For ground-based enemies, this solution was robust enough to become the default test subject used for engaging the player.

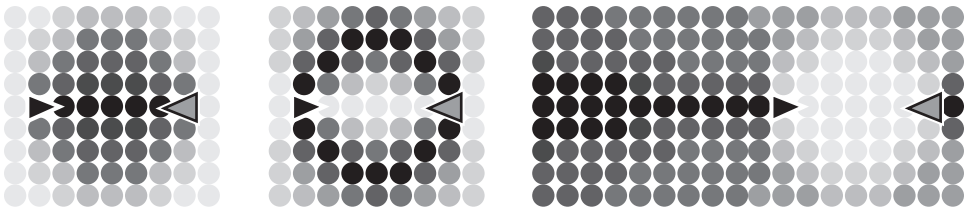

Figure 26.5

(a) Approach locations prioritized by directness. (b) Approach locations prioritized by indirectness. (c) Retreat locations prioritized by directness.

26.5 Test Scoring Functions

Once all sample points have been scored, they must be normalized and ranked. Most commonly, we use this value as-is, inverting the priority when needed with a negative test weight. However, as the final post-processing stage in a test's evaluation, we can pass the normalized score of each sample point to a scoring function to transform its value. Doing so allows us to increase or decrease the influence of certain samples, adding precision to our test's intent, or transforming the concept it measures entirely.

- *Linear scoring* (Figure 26.6a) is the backbone of most tests, returning the value of normalized test scores exactly as they were passed in.
- *Square scoring* (Figure 26.6b) strongly deemphasizes all but the highest ranked samples in the test. Useful when we want emphasis to drop off rapidly.
- *Square root scoring* (Figure 26.6c) does the opposite; overemphasizing all but the lowest-ranked samples in the test.
- *Sine scoring* (Figure 26.6d) differs from other methods, in that it emphasizes mid-range values, and de-emphasizes both the highest- and lowest-ranked sample points.
- Where scoring functions describe the *rate* at which emphasis should change, a test's weight determines the *direction* of change. When a test's weight is negative, an increase in score is replaced with a corresponding decrease, inverting the scoring curve (Figure 26.6b and e, c and f).

Queries typically require several tests to express a useful concept. In these cases, the highest ranked location will almost always represent a compromise between multiple competing goals. The role of scoring equations is to allow each test to define how tolerant it is of suboptimal locations, and how quickly that tolerance changes. In conjunction with the test weight, this lets us define how that compromise should be met.

For example, if we want an agent that steps away from others as its personal space is encroached, how should we express its level of discomfort? We might approximate it using

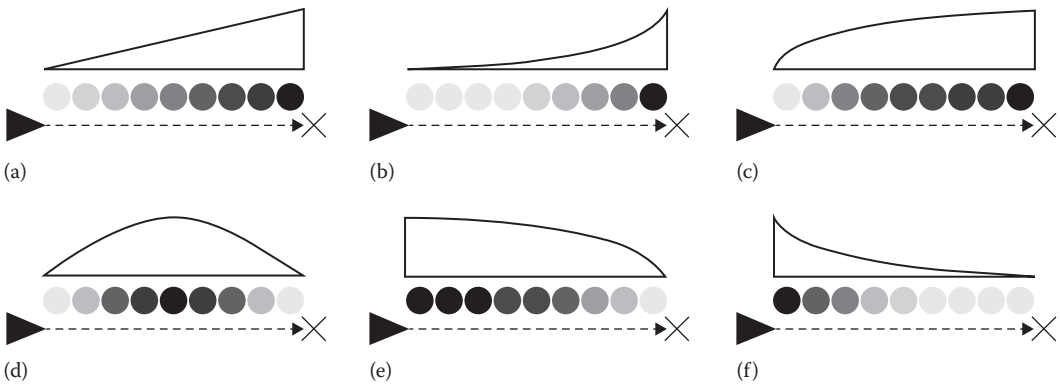

Figure 26.6

Normalized result of a distance test from sample points to an agent's target, after applying linear (a), square (b), square root (c), sine (d) scoring functions. Response curves become inverted when a negative scoring weight is used, as shown in (b) and (e), and (c) and (f), respectively. Darker shades indicate better locations.

two tests: A distance test against our current location, expressing our desire to move as little as possible, and a second distance test, negatively weighted against other actors in the scene, expressing our desire to move as far away from them as possible. The balance of these two tests determines when the agent will react. For example, if we use square scoring with a negative weight on the second test (Figure 26.6e), in general other actors will have little effect on the agent's evaluation of its current location, but when approached extremely closely its desire to stay in its current location will be outweighed by its desire to avoid others and it will try to find a slightly less crowded position. Alternatively, if we instead use square root scoring with a negative weight (Figure 26.6f) then even the influence of distant actors will quickly become overwhelming, creating a nervous agent with a strong desire to keep far away from anyone in the area.

The advantage to expressing satisfaction with scoring functions is that it allows us to produce a dynamic, natural response that is not easily expressed by the hard edges of an acceptance condition. If, instead of measuring satisfaction, we simply invalidated all locations within 2 m of another actor, our agent's response becomes predictable and artificial. However, by defining a level of comfort for all sample points in the test, our response can change along with the environment. For example, when entering a quiet subway car the agent in our scoring equation example will naturally maintain a polite distance from other passengers, but will gradually permit that distance to shrink as it becomes packed at rush hour, continuously adjusting its reaction as the environment becomes more or less crowded.

26.5.1 Sine Scoring Techniques

Although square, square root, and other monotonic scoring functions can be used to tune test results by compressing or expanding the range of suitable positions, sine scoring gives us an opportunity to use existing tests in altogether new ways. For example, applied to a distance test with a minimum and maximum range, we can define a specific ideal radius to approach a target—the average of the two ranges—while still accepting positions closer or further from the target, but with reduced priority.

When applied to the dot product, we have even more options:

- When used against the agent's orientation, we can express preference for positions to both the left and right, or both forward and behind with a negative weight.
- If we use the absolute value of the dot product with an agent's orientation, this produces the same result. However, when both are combined, we can now represent preference for either the cardinal or intermediate directions.
- As described in Section 26.4.3, applied to the dot product *(Agent→Sample Point)·(Sample Point→Target)*, we can create a circle between the agent and the target, representing a roundabout approach.

There are many other instances where the most interesting samples are those that lie in the mid-range of a test's scoring function; sine scoring is the key to discovering them!

26.6 Continuous versus Sequential Updates

Most queries are designed to be executed once, at the start of a behavior, to provide the agent with a suitable destination for its current goal (Figure 26.7a). To adapt to changing world conditions, such as a cover point becoming exposed, it is common to periodically run a validation test on the agent's destination while en route, but for efficiency we typically do not execute another full query until after we have arrived. In some cases, however, it is worth the expense to update continuously, periodically reexecuting the original query without waiting to arrive, and thus generating new recommendations as world conditions change (Figure 26.7b). Not only does this allow us to react more dynamically, it opens the door to new types of query-based behaviors that previously could only be expressed in code. Common concepts like surrounding, orbiting, zig-zag approaches and random walks can all be expressed as a single, repeatedly executed query without any programming required.

26.6.1 Continuous Query-Based Behaviors

By periodically rerunning the same query, providing frequent updates to the agent's destination, we can create the illusion of sophisticated navigation or decision-making. For example, as shown in Figure 26.8, by generating a ring of points on the navigation mesh around the agent's target, then simply prioritizing samples a few meters away as well as those in front of our current position, an agent will begin to circle-strafe around the target, avoiding obstacles as it moves and even reversing direction when it becomes stuck.

Traditional positioning can be enhanced by this technique as well. For example, when approaching a target as part of a group, not only can we maintain ideal distance from the target as it moves, but by negatively weighting the area around the agent's teammates the group can dynamically reposition themselves in relation to each other, creating a natural and responsive surround behavior (Figure 26.9).

26.6.2 Continuous Querying versus Destination Validation

While promising, there are caveats to this method. Compared to destination validation, continuous querying is responsive and can produce high-quality results, but is also

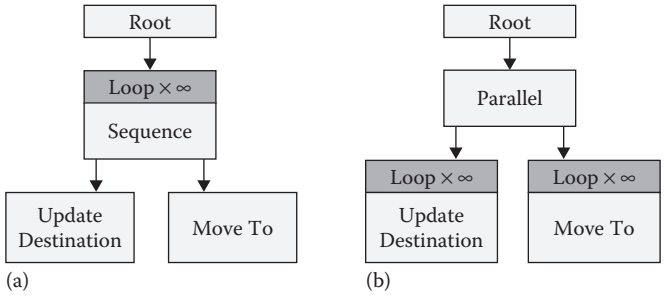

Figure 26.7

Behavior tree implementation of a sequential query-based behavior (a) versus a continuous query-based behavior (b).

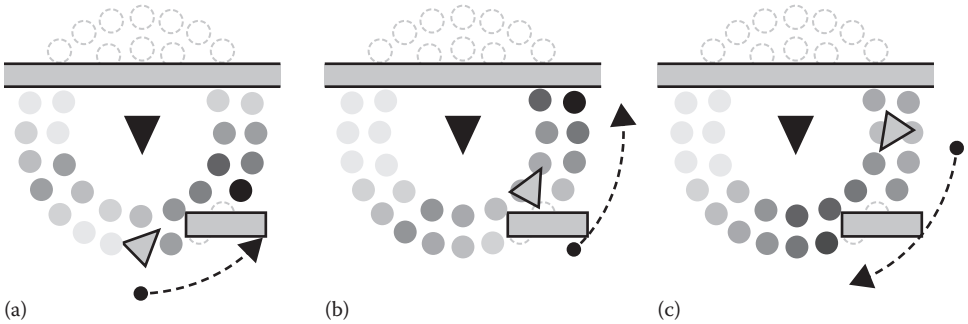

Figure 26.8

Orbiting a target with a continuously updated query (a). As positions in front of the agent become increasingly unsuitable, positions behind the agent gradually gain utility (b), ultimately causing the agent to automatically reverse direction when it can no longer proceed (c).

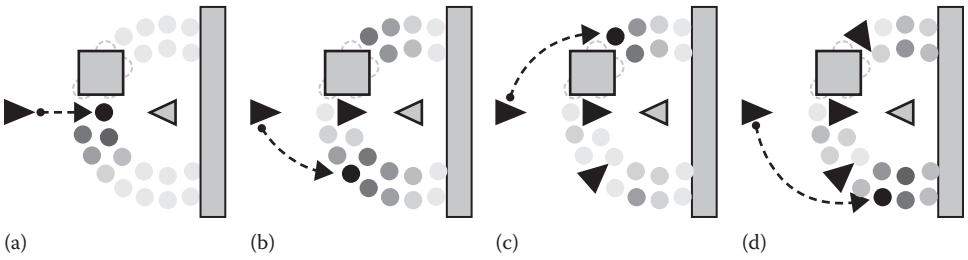

Figure 26.9

A group of agents approach and surround a target in sequence. In this example, agents prefer locations that are near the target, as well as along the vector between the agent and the target (a), but negatively weight locations near other agents to prevent clustering (b, c, and d). This produces an organic surround behavior that maintains formation continuously as the target moves, and adapts naturally as the number of agents increases or decreases.

computationally expensive. If too many agents in the scene are issuing too many queries, you can easily burn through your AI's CPU budget. It is also more challenging to avoid degenerate behavior: agents becoming stuck in local minima, oscillating between destinations unnaturally, or moving in a stop-and-go fashion by selecting destinations too close to their current position. Nevertheless, the benefits can be substantial and are well worth consideration.

26.7 Reducing Query Failure

Using the techniques thus far, we have been able to reason about the ideal layout for generated points, apply tests on single or multiple subjects, and adjust their scoring based on our needs. In theory, this should be enough to produce high-quality results from a spatial

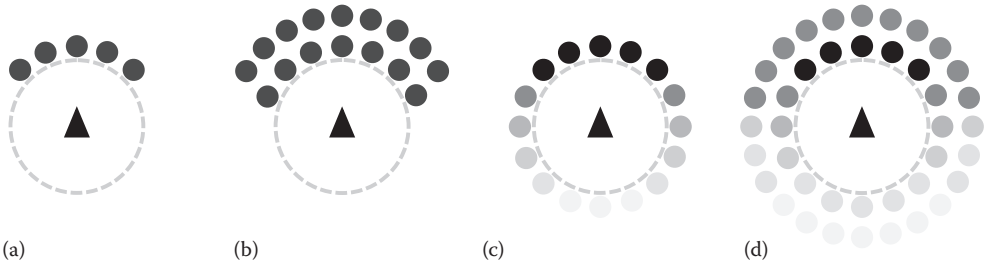

Figure 26.10

Reducing query failure of a strict, brittle query (a) by increasing robustness (b), permissiveness (c), or both (d). Whenever possible, strategies (c) and (d) will return the same result as the original query.

query system for most position selection tasks. However, in practice we rarely are so lucky. For example, the player may be in a narrow corridor, causing all samples in our flanking query to fail, or they may be facing a wall, making a dramatic surround from the front impossible. A query is useless if it never works and, unfortunately, it is common to design one that fails too easily in unexpected circumstances like these. Fortunately, by making some simple adjustments, a brittle query can be adapted to provide graceful degradation of position quality in unfavorable conditions. In this section, we show how a query can be modified to give the AI the ability to execute a behavior in a wider range of conditions while still returning the ideal result when possible, making them resilient to the complexities of a modern game environment (Figure 26.10).

26.7.1 Increasing Permissiveness

The first action we can take to make a brittle query more failure-resistant is to make it more *permissive*. That is, we can relax our success conditions so that we have more sample points that can serve as a destination, but use tests to give them a lower final rank so that they are only selected when the ideal conditions are unavailable. In Figure 26.10, we have an example query that attempts to find an attack position in front of the agent's target. If it is acceptable to attack from behind, but not preferred, we can add additional sample points around the target, but weight them with a dot product test so that the samples behind the target receive a low rank. Done this way, agents will still approach the player from the front, unless it is impossible due to the player's location (near the edge of a cliff, facing a wall, etc.).

26.7.2 Increasing Robustness

The next action we can take is to make the query more *robust*. In this case, we relax our concept of the ideal case entirely, providing a larger pool of sample points to draw from under ideal circumstances. For example, our query may specify a position 5 m away and up to 30° from the front of the target, but in reality it may be the case that it will actually work fine at any distance between 5 and 8 m away, and up to 45° in front of the target. Figure 26.10 also shows an example of this solution.

26.7.3 Fallback Queries

Some query systems, such as EQS, provide multiple query *options*, or strategies, that can be defined as part of a single query. These can be thought of as fallback queries, providing alternative location suggestions to consider if the initial query fails. Thus if the initial option has no usable samples, subsequent ones are executed in order until one succeeds. Only when all options fail does the query itself fail. Clever use of fallback queries can also create opportunities for optimizing high-quality behavior: By defining a narrow initial sample set, we can run more expensive tests in the primary option that would normally be cost prohibitive, such as collision raycasts. In the fallback query, with a wider range of sample points, we can remove these tests to find a more mediocre, but still acceptable, location.

26.7.4 Preserving Quality

Taking advantage of these techniques, we can adapt our queries to be both permissive and robust, while still returning the same results as the initial query when possible. In Figure 26.9, the result of the rightmost query can be achieved in two ways:

1. Combine permissive and robust testing strategies: Pair a dot product gradient ranking with a larger valid sample area, then further add a distance test to weight the closest points higher. This layering results in the original set of points receiving the highest rank whenever they are available.
2. Define the leftmost query as the initial query strategy; if this fails, execute a fallback query that combines the permissive and robust testing strategies. This has the benefit of only incurring additional cost when ideal conditions are unavailable, at the expense of a higher overall cost of running both queries in suboptimal conditions.

26.8 Example Behaviors

In this section, we provide a handful of the many behaviors possible using only the most basic tests and the continuous movement behavior tree in Figure 26.7. For each behavior listed below, the only difference in the agent's AI is the query itself.

26.8.1 Directed Random Walk

By selecting a random item from the set of sample points, instead of the one that has the highest rank, we can add variety and unpredictability to a query-based behavior. For example, the classic NPC random walk (moving a short distance in an arbitrary direction, stopping briefly between moves) can be represented as a query by generating a filled circle of points up to a specified radius, ranking them by sine-scored distance to define an ideal move distance, and finally selecting randomly among the top 25% highest ranked points. By adding a dot product test to favor the agent's current direction, we eliminate harsh turns and unnatural oscillations, creating an agent that takes a long winding path through its environment. Finally, a minimum distance test weighted against other agents keeps agents evenly distributed and avoids collisions in crowded environments. By including our current position in the set of generated points, an agent that is boxed in can choose not to move until a reasonable location becomes available (Table 26.1).

Table 26.1 Directed Random Walk: Moves Relatively Forward, Avoiding Others

Weight	Type	Parameters	Scoring
N/A	Ring generator	0–8 m around agent	
1	Distance	Relative to agent	Sine
1	Dot	(Agent→Sample)· (Agent rotation)	Sigmoid
1	Minimum distance	Relative to other agents	Linear, 2–10 m range

26.8.2 Stay on Camera

If we want to position agents where they can be seen by the player, we can use the location and orientation of the camera to prioritize sample points based on their distance from the center of the screen. A camera frustum test can be approximated with a dot product, using a lower clamp to define the angle of a view cone. A negatively weighted distance test relative to the agent ensures the agent moves as little as possible to stay in view. Optionally, adding a raycast test between each point and the camera will eliminate points in front of the camera but hidden behind other objects or scenery, improving behavior quality at the cost of performance (Table 26.2).

26.8.3 Orbit

To walk or run in a circle around a target, we generate a ring of points within the minimum and maximum acceptable radius, then use a set of tests that when combined generate forward movement around the ring. The first, a sine-scored distance test around the agent, defines an ideal movement distance a few meters away from its current position; far enough that we should not arrive before reexecuting the query, but close enough to ensure small, smooth adjustments in heading. Next, a dot product test prioritizes items in the direction that the agent is currently heading, which encourages stable forward movement along the ring (clockwise or counter-clockwise). A second sine-ranked dot product test prioritizes points on the tangent line from the agent to the ring. This serves two purposes: It directs the agent to approach the ring along the tangent line (rather than head on, then turning 90° to begin orbiting), and it strongly prioritizes items directly in front of and behind the agent, allowing the agent to reverse direction when blocked while further stabilizing forward movement. Finally, clamped minimum distance tests around the positions and current destinations of other agents provide local avoidance (Table 26.3).

Table 26.2 Stay on Camera: Agent Attempts to Stay on Screen While Moving as Little as Possible

Weight	Type	Parameters	Scoring
N/A	Grid generator	20 m around agent	
–1	Distance	Relative to agent	Linear
1	Dot	(Camera→Sample)· (Camera rotation)	Linear, 0.85–1.0 range

Table 26.3 Orbit: Moves in a Circle around a Target Avoiding Others

Weight	Type	Parameters	Scoring
N/A	Ring generator	5–9 m around target	
8	Distance	Relative to agent	Sine, 0–6 m range
4	Dot	(Agent→Sample) (Agent→Destination)	Linear
2	Dot	(Agent→Sample)·(Agent→Target)	Sine
1	Minimum distance	Relative to other agents	Sigmoid, 0–5 m range
1	Minimum distance	Relative to other agent destinations	Sigmoid, 0–5 m range

Note: If forward movement is obstructed by the environment or other agents, the agent will turn around and continue orbiting in the opposite direction.

26.8.4 Boids

Spatial queries can even represent artificial life simulations. Craig Reynold's historic boids program, simulating the flocking behavior of birds, produces complex, emergent group behavior from an unexpectedly simple set of rules (Reynolds 1987). By implementing these rules using spatial query tests, we can recreate the original boids simulation as a continuous query behavior. In the original SIGGRAPH paper, individual boid movement was produced by combining the influence of three separate rules:

- Separation, to avoid crowding
- Alignment, to coordinate the flock direction
- Cohesion, to prevent the flock from dispersing

Within a query, separation can be represented as a minimum distance test against other agents, alignment as a dot product test against the average heading of the group, and cohesion as a distance test against the group centroid. By tuning the weights and ranges of these tests, we can adjust the emergent properties of the behavior (Table 26.4).

Table 26.4 Boids: Simulates Boid Flocking Behavior

Weight	Type	Parameters	Scoring
N/A	Ring generator	1–20 m around agent	
1.2	Minimum distance	Relative to other agents	Linear, 0–4 m range
0.5	Dot	(Agent→Sample)· (Average rotation of other agents)	Linear
–1	Distance	Relative to other agents	Linear

Note: Simulates boid flocking behavior using minimum distance, dot product, and average distance tests to represent separation, alignment, and cohesion respectively.

26.9 Conclusion

Once a novel alternative to traditional techniques, over the past five years spatial query systems have evolved into indispensable tools for AI development. Now commonplace and supported by multiple widely used game engines, integrating spatial query systems into your AI is more practical than ever, providing faster iteration time and higher quality position selection in dynamic and complex environments. By understanding the strengths and weaknesses of each component of a query, we can improve query quality and flexibility over a wider range of environmental conditions. Single queries, executed continuously, can even express traditionally code-driven movement behaviors, making query systems an increasingly versatile tool, able to single-handedly support most, if not all, destination selection in a project.

References

Jack, M. 2013. Tactical position selection: An architecture and query language. In *Game AI Pro*, ed. S. Rabin. New York: CRC Press, pp. 337–359.

Mars, C. 2014. Environmentally conscious AI: Improving spatial analysis and reasoning. *GDC 2014*, San Francisco, http://www.gdcvault.com/play/1018038/Spaces-in-the-Sandbox-Tactical.

Reynolds, C. W. 1987. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4) (SIGGRAPH '87 Conference Proceedings), 25–34.

Shirakami, Y., Miyake, Y., Namiki, K. 2015. The decision making Systems for Character AI in Final Fantasy XV -EPISODE DUSCAE-. *CEDEC 2015*, Yokohama, http://cedec.cesa.or.jp/2015/session/ENG/5953.html.

Zielinski, M. 2013. Asking the environment smart questions. In *Game AI Pro*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 423–431.