Taming Spatial Queries – Tips for Natural Position Selection

Eric Johnson

1 Introduction

Spatial query systems are powerful tools when leveraged effectively, giving developers the ability to craft sophisticated, complex movement behaviors quickly and concisely, while giving agents the power to move with intelligence and respond to changes with resilience. Even so, a query system brings with it unique challenges that can make it difficult to realize its full potential. In this chapter, we cover four specific position selection issues that can manifest themselves in spatial queries: Oscillation, artificial behavior boundaries, artificial spatial boundaries, and priority inversion via normalization. By recognizing and handling these cases, we can prevent several common classes of unnatural or immersion-breaking behavior, as well as improve existing behaviors with subtle nuances that make movement feel more natural and organic.

Note that this chapter assumes a basic understanding of spatial query systems and query-based movement behaviors. For additional information, please refer to chapters 26 and 33 in Game AI Pro, and chapter 26 in Game AI Pro 3 (Jack 13, Zielinsky 13, Johnson 17).

2 Preventing Destination Oscillation and Instability

Because the purpose of a spatial query is to evaluate multiple positions around the game world and return the best location for a specific behavior, it is reasonable to expect that by executing the same query again, an agent can detect if it is out of position and reposition if needed. Further, by executing the same query repeatedly, agents can automatically maintain an ideal position for their behavior as the environment changes. Most of the time, this works as expected, and allows agents to move fluidly through a space over time. If the ideal destination is ambiguously defined, however, multiple locations can compete for the highest score, resulting in a destination that oscillates or moves erratically every time the test is updated. As a result, as we re-run the query, instead of staying in an optimal location, the agent continuously changes course, moving between two or more high-scoring locations but never arriving at either.

Consider the task of creating a simple query that directs an agent to approach a target at an angle as the start of an orbiting behavior. We might start by generating a ring of points around the target, then rank them by directness using the dot product $(agent \rightarrow sample\ position) \cdot (agent \rightarrow target)$. The locations with the smallest scores will give us the most indirect approach angle. However, the results of this query will be symmetric: Approaching from the left is just as good as approaching from the right. Which direction will our agent prioritize?

In practice, it will prioritize neither. As the agent moves, slight changes in the angle to the target can cause the opposite side to be favored, and if the query is executed at a high

enough frequency, every time we move left the right-hand side will gain value, and every time we move right the left-hand side will gain value. As the agent continuously changes its approach direction, it ultimately commits to neither and approaches the target head on. As a result, the behavior we intended to create fails to materialize entirely.

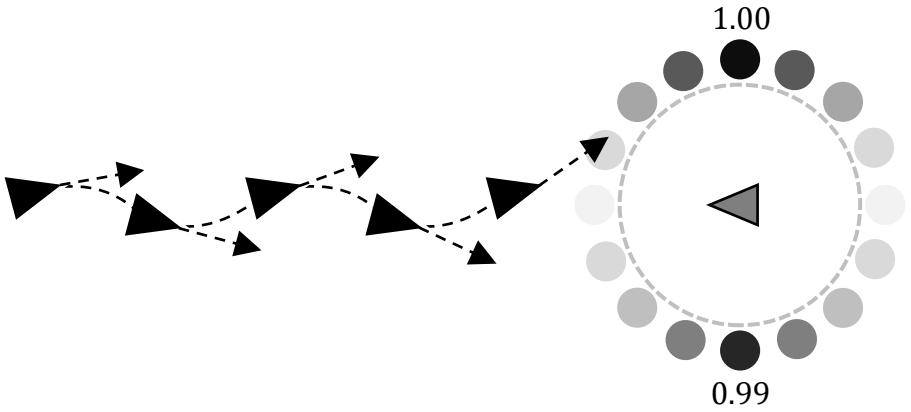

Figure 1 Oscillation between two flanking positions on the left and right of a target cause an agent to approach the target head on while weaving back and forth.

There are several techniques we can use to reduce goal oscillation. First, if a query suffers from oscillation due to symmetry, we can eliminate it by establishing a very small preference for one side. Adding a dot product test that applies a constant score to all positions on one side of the target acts as a tiebreaker when both sides would otherwise be equal. However, symmetry is responsible for only some types of oscillation; queries with competing tests, such as “get close to the player, but stay away from other agents”, or queries with tests that are strongly affected by movement or direction, such as “prefer positions behind me” will require other approaches.

As a general-purpose solution, we can add hysteresis to the result to stabilize a fickle query. Hysteresis, or decision momentum, is the idea that once we have made a choice, we want to stick with it until we have a strong motivation to abandon it. When the top-ranking locations in a query all have similar scores, but are in wildly different locations, this will ensure that our agent ignores small fluctuations in the results and continues moving towards a goal as long as it remains one of the best locations in the query. However, there is more than one way to implement hysteresis in a query system, and each one brings with it different benefits and caveats.

The most basic approach is to simply add a low-weighted distance test that prioritizes locations near our current destination (the previous winning location), creating a preference to keep moving towards the same general area. This method has two primary advantages: The size of this area can be tuned to give the agent more freedom to reposition locally while still maintaining overall global stability, and secondly, because it can add bias over an area, it works well in queries where the local optimal position can drift over time (for example, maintaining a position relative to a moving player). However, since this method implements hysteresis as simply another test, we are unable to precisely control the actual amount of bias. For example, if we have three tests, each with a weight of 1.0, the winning sample position can have a value anywhere between 1.0 (highest utility in one test,

lowest in the other two) and 3.0 (highest utility in all tests), depending on the state of the world when the test was executed. If we then add a destination bias test with a weight of 0.1, our current destination will receive a variable amount of decision momentum: 10% score bonus in the former case, but only 3% in the latter.

A more consistent method is to implement hysteresis explicitly, comparing the fitness of our current destination against the new winning position in the query. To do this, we must modify our query to re-insert our current destination into the generated set of sample positions in order to recalculate its score when the query is executed. We then compare its updated score against the score of the winning position. If the top-ranking location is better than our current destination by more than some threshold (say, 10%), we abandon our current destination and update it with the new highest scoring position, otherwise we discard the results, keeping our current destination. Compared to score biasing methods, which allow the destination to move freely within a localized area, hysteresis tends to create "sticky" destinations that resist any repositioning at all. This has the benefit of strong stability, and works especially well for queries that evaluate fixed locations such as pre-generated cover points, but the tradeoff is that we may hold on to stale, suboptimal positions if the hysteresis threshold is too high. Additionally, for queries where generated points may change position frequently, it is poorly suited to removing asymmetric oscillation: In our toy example, if the player moving towards us causes our current destination to lose enough value to be replaced by a new winner, it isn't certain which side the agent will pick; instead of eliminating oscillation, we've only reduced its frequency by discarding the results of some of the query updates.

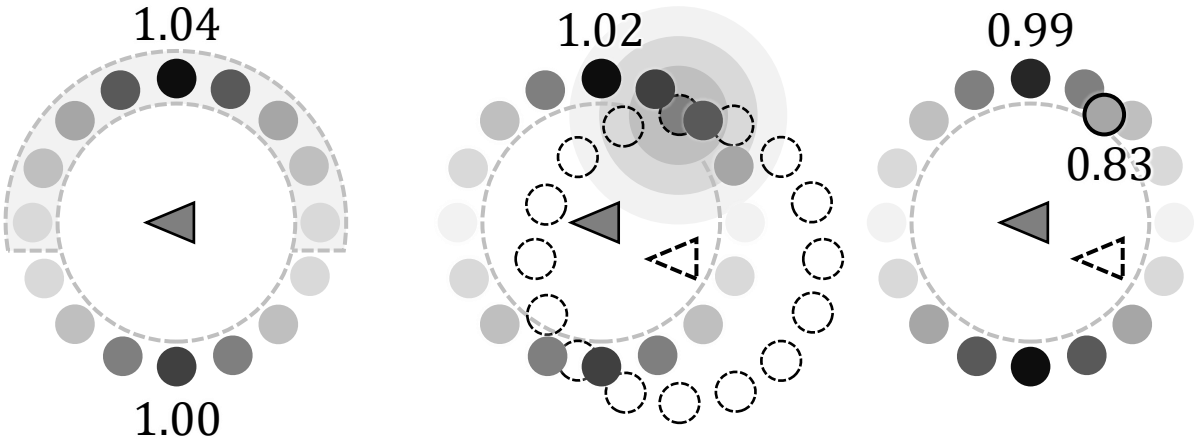

Figure 2 Oscillation mitigated by biasing points on one side of a symmetric test by 5% (left), by biasing points by their distance from the previous winning location the last time the query was executed (center), and by using hysteresis to require a new destination to be at least 20% better than our current one (right).

Oscillation and instability are commonly occurring problems in both simple and complex query-based movement behaviors, leading to chaotic character movement and interfering with an agent's ability to execute important behaviors reliably. Even when we identify oscillation in a behavior, eliminating it is not always straightforward: The techniques

described above are each effective at preventing a specific type of instability, but none will work in all cases. Used appropriately, however, they are indispensable tools that give us the means to express high quality, elegant behavior that would otherwise be unachievable.

3 Adding and Removing Tests at Runtime

When creating a spatial query, our workflow is focused on how to best balance a group of tests that measure different aspects of a position to accurately determine the utility of different locations with respect to a specific goal: Finding suitable cover, surrounding a target, retreating from danger and so on. As the environment changes, the utility of a specific location increases or decreases accordingly. While robust and responsive, it limits us to a single definition of fitness for a specific goal: We will always find the same type of cover, settle into the same type of formation, retreat to the same type of safe spot. In this section, we will see how enabling and disabling tests at runtime allows a single query to switch between two or more definitions of fitness dynamically, how we can manipulate test weight to blend between these states seamlessly, and how to use increasing levels of indirection to manage the blend transitions themselves.

3.1 Controlling Test Weight with Test Utility

Let's start with a hypothetical movement behavior for a group of low-level enemies designed to encourage a fast-paced, active playstyle: When the player is actively moving around the scene, they should spread out around the level, giving the player easy targets to pick off one by one. However, when the player lingers too long in one area, they should close in, descending on the location aggressively to punish idling. It does this with two separate queries, one that prefers locations away from other agents of the same type, and another that prefers locations near the player.

Traditionally, we would implement these as separate branches in a behavior tree, using a decorator condition to determine which behavior to run. Expressed as code, it might read something like the following:

```
if time_since_player_stopped_idling < 5.0:
  moveToSwarmPlayer()
else:
  moveToSpreadOut()
```

In this example, the decorator condition causes agents to give up chasing a moving player after a short while, but the boundary condition is easily visible to players: Enemies begin swarming instantly after even the slightest pause, but give up entirely if the player can continue running for exactly five seconds. Without increasing the complexity of the behavior tree, how can we make this transition look more natural?

To begin, let's move the transition logic out of the behavior tree entirely, and into the query itself. If the only difference between our two query behaviors is that one has an additional test that prefers locations near the player, then by enabling or disabling this test at

runtime we could author a single common query that activates this test only when it is relevant. In doing so, we can simplify our behavior tree to a single node that simply updates our position. We can accomplish this by assigning the distance test a measure of utility tied to the current movement goal, and multiply the test's weight by this value. When it set to zero, a test has no utility and is disabled, while setting it to 1.0 enables the test, applying its full weight to the query:

```
if time_since_player_stopped_idling < 5.0:
  approachPlayerTestImportance = 1.0
else:
  approachPlayerTestImportance = 0.0
```

By calculating this value in advance and assigning our test's weight to this external variable (for example, via a blackboard key in the behavior tree), we can create queries that activate a test only when it is relevant, changing their behavior dynamically in response to the environment. We can now start to improve our behavior by changing the transition logic to be a bit more forgiving:

```
if time_since_player_stopped_idling > 5.0:
  approachPlayerTestImportance = 0.0
elif time_since_player_stopped_moving > 5.0:
  approachPlayerTestImportance = 1.0
```

This is a good start. Now the player must remain still for several seconds before enemies begin to swarm, and must remain in motion for the same amount of time to get relief. Short bursts of movement and small pauses will both be ignored, adding some degree of stability to the transition between behaviors.

3.2 Discrete Utility vs Continuous Utility

Now that we can use test utility to change query behavior on the fly, we can continue to improve the quality of the transition by replacing the abrupt events that enable or disable the test with a continuous measure of utility that increases or decreases the test weight based on our confidence that the player is idling or in motion. One way to estimate this is to take a rolling average of the player's current velocity, then compare the length of this vector against the player's maximum speed. When the player is moving in a straight line, this value will approach 1.0, but when the player is stopped, moving erratically, or spinning in circles, this value will decrease and approach zero. Now, instead of a query that can toggle between the two different behaviors, we now can seamlessly fade between them by setting our test weight to this value, adjusting the relative strength of one or the other to match to the player's perceived level of activity. In our example, the longer a player idles, the stronger the swarm behavior will become, and the longer a player moves, the stronger the scattering behavior will become.

3.3 Tuning Results with Response Curves

While our behavior is far more natural and responsive than when we started, it can still be improved further. As we control a test's final score by multiplying it by a weight, and control a test's weight by multiplying it by a measure of test utility, and control test's utility by multiplying it by a measure of confidence such as a rolling average, so can we control the measure of confidence by multiplying it with a response curve.

Similar to how easing functions are used in animation to express different types of natural movement, adjusting the rate at which we blend in and out of different query behaviors lets us fine tune how our agents react to a changing environment. For example, by applying a sigmoid to the results, we can create a sticky or magnetic transition, where a test is slowly pulled away from zero or full utility, then quickly snaps to the other extreme. For our swarm behavior, we might want to use a cubic curve to adjust the leniency of the behavior, keeping enemies docile until the player stays in the same spot for a long time, while also quickly abandoning the swarm behavior once the player begins moving again. In this way, we can continue adding indirection indefinitely to control an agent's behavior at increasingly higher levels of abstraction. For example, if we wanted our enemies to behave more aggressively later in the game, there is no reason why we can't use multiple response curves to produce different levels of difficulty. We might then add yet another response curve to continuously tune the level of aggression at every point in the game. We can also continue expanding indirection laterally, creating queries with more than one utility-weighted test to encode increasingly broad or abstract goals like "travel with player" or "apply pressure", allowing us to create much more powerful and expressive behaviors than would be possible with fixed test weights.

4 Selecting the Appropriate Test Normalization Method

Spatial query systems generally process a request in the following order: First, sample points are generated to evaluate. Next, for each test in the query, each item is scored, the results of the test are normalized, and then each result is multiplied by a test weight. Finally, for each position, the query calculates the sum of weighted scores from each test, sorts the results and returns the highest-ranking location.

While test normalization is necessary to compare the relative importance of different aspects of a position, *how* we normalize is just as important as performing normalization itself. The normalization method we choose defines how our original measurements should be transformed into a degree of fitness, and by changing the method, we can change the definition. Should a specific value decrease in fitness if better candidates appear? Is there a minimum, ideal, or maximum level of fitness? By aligning the normalization method with the purpose of the test, we can ensure our test results match our intent.

4.1 Relative Normalization

Relative normalization is the easiest method to implement as it requires no domain knowledge about the data it receives. Test values are simply shifted and scaled such that the lowest-ranking sample position is assigned zero utility (0.0), while the highest-ranking position is given full utility (1.0). This is the default method to apply when we don't have a

specific value or range of values in mind that represents a good or ideal result, and only want to express the subjective idea that larger values are better than smaller ones (or vice versa). However, as a consequence of this definition, as the range of values becomes larger the influence of a specific quantity diminishes.

For example, say we have a ranged attacker that wants to position itself to have line of sight to its enemies, while also staying out of their attack range. Further, to prioritize safety, we would like the amount of danger at a particular location to reflect the number of enemies able to attack the agent at that position. With relative normalization, the most dangerous location will thus always be ranked "100% dangerous", while less dangerous locations will receive a lower value. In practice, this means that when only one enemy is present, the area around it will be 100% dangerous, but when the attack range of multiple enemies overlap, the overlapping areas are technically the most dangerous location, and positions in attack range of only one enemy become relatively less dangerous by comparison. As seen in Figure 3, when two enemies overlap, positions in range of only one enemy are considered 50% dangerous, and when three enemies overlap, positions that once had a danger rating of 100% are now (relatively speaking) only 33% dangerous in the current environment.

As a result, the perceived importance of an absolute value can diminish suddenly and unexpectedly as conditions changes, affecting the behavior of the query itself in turn. In our example, once three enemies gather in the same location, the importance of having line of sight to the target outweighs the danger of being in attack range of a single enemy, resulting in a ranged agent that correctly keeps its distance from isolated targets, but is happy to approach enemies when they cluster together.

4.2 Clamped Normalization

Clamped normalization defines the best and worst values possible for a test; anything beyond these ranges will be clamped to zero or 100% utility. For example, if we are looking for nearby cover positions, but all available cover is far away, then a distance test should return low scores for all positions to reflect that not even the closest among them are effective at satisfying our goal of being close to the agent. By using clamped normalization with a range of 0 to 20 meters, a query with two nearby cover positions at one and two meters away will assign them 95% and 90% utility, respectively, while one with two distant cover positions at 19 and 20 meters away will be assigned 5% and 0% utility. In this way, where relative normalization is a subjective measure of quality, by specifying in advance the desired best and worst case for a test we can ensure that the strength of a sample position's score is objective and grounded to a concrete definition of utility.

The caveat to this method is that determining a suitable range requires domain knowledge about the game, and even when tuned carefully may still not return viable results in all cases. In our example above, because all values outside this range will be treated equally (either zero or 100% utility), a cover point 21 meters away will appear just as desirable as one that is 100 meters away. If we add a new encounter where enemies stream in from the distance before entering cover, all cover points in the encounter area may appear to be equally valid, causing agents to attempt to take cover in seemingly random locations around the stage.

4.3 Unclamped Normalization

Like clamped normalization, unclamped normalization defines values of zero and full utility, but does not restrict the utility of positions that exceed these limits. As a result, not only will a specific value always return the same amount of utility, its utility can be less than 0% or greater than 100%. This is especially useful when we want to allow a test to exert more influence than usual over the query results to respond to rare or extreme conditions.

In our example, by using unclamped normalization we can express the idea that staying out of attack range of an enemy is always twice as important as having line of sight, but if a position can be attacked by more than one enemy, it is even more important to avoid that location. Further, this importance should grow without limit as the number of enemies in range increases to reflect the greater danger of the position.

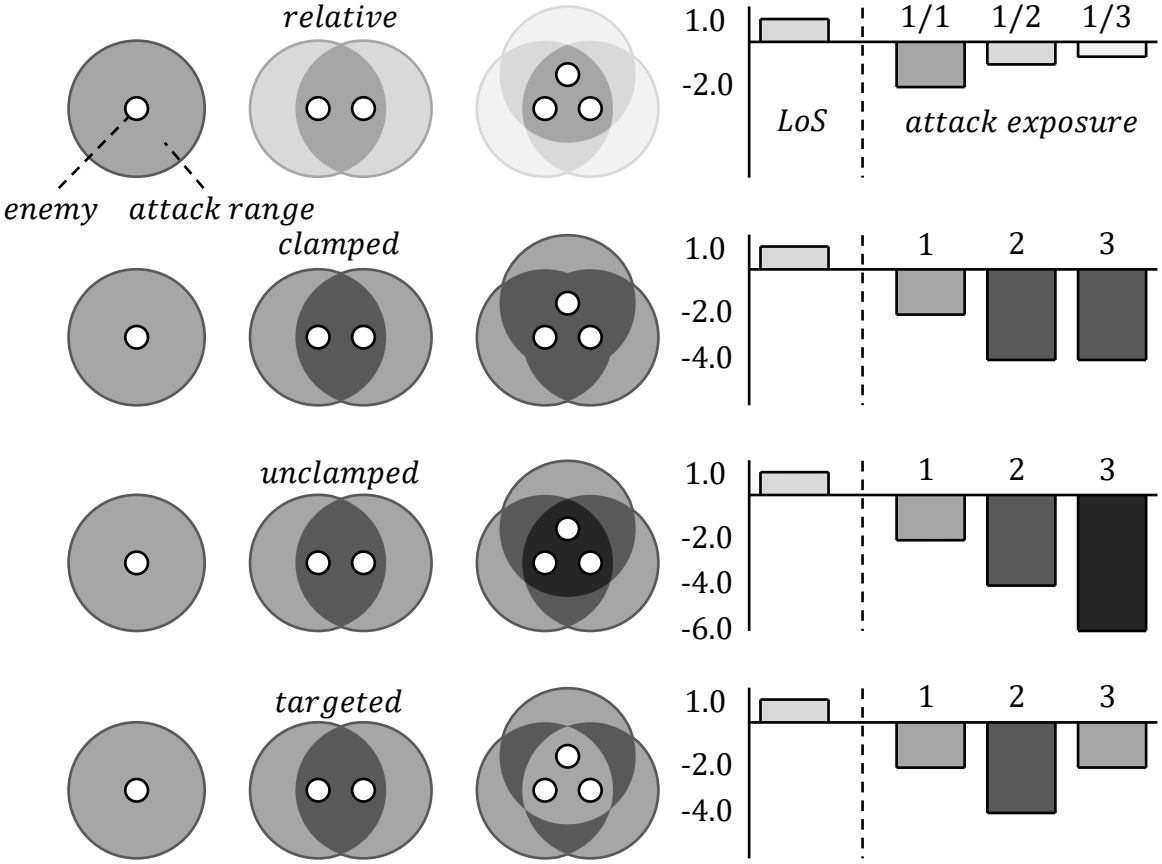

Figure 3 Comparison of different normalization methods, and their effect on a location's final test score relative to other tests. Each query has a line-of-sight test with a weight of 1.0, and an enemy attack range test with a weight of -2.0. The normalization method used changes the perceived importance of a particular level of danger relative to the importance of maintaining line of sight. From top to bottom: Relative normalization, clamped normalization, unclamped normalization, and targeted normalization.

4.4 Targeted Normalization

Targeted normalization defines an ideal value for a test that provides maximum utility, scaling the utility of locations based on the absolute difference between a location's score and this optimal target. For example, if an agent's ideal attack range is thirty meters, we can use targeted normalization to decrease the utility of areas both closer and farther than this distance from the target. This can also be combined with clamped normalization, if we want to specify a maximum distance from the ideal value at which the utility should drop to zero.

5 Recovering Spatial Information Via Post Processing

In any query system, each location is examined in isolation; each test measures the quantity of some particular quality without any knowledge about other locations in the query. For concepts that are inherently spatial, such as visibility, the lack of knowledge about neighboring positions causes us to lose information about the structure of the environment such as the size of a hiding spot or the distance to an exposed area. Without this awareness, a query is unable to recognize positions inappropriate for the intended behavior, such as hiding locations that are too small, or are not deep enough in cover and only hide the agent's center, leaving half of its body exposed (Isla 09).

Once our set of sample positions has been scored, however, we can use the location of each result to recover these spatial features from the environment, allowing us to produce a more realistic evaluation of a position and even to test for entirely new concepts.
To recover this spatial information, let us borrow techniques from image processing, applying morphological operations such as blur, erosion, dilation, and gradient to find rough area boundaries and the distance from those boundaries, then manipulate those discovered features. The basic premise is simple: We first gather the set of results from a test, then we apply a morphological operation against it, looking at each position and its neighbors to compute a transformed score. Finally, we apply the results of the operation back to the original test set, overwriting it.

The first and most basic operation is **erosion**: shrinking the size of a valid group of positions, so that potentially inappropriate locations near the edge of the group are removed from consideration. To apply this to a pass/fail test, such as a visibility raycast, we discard any successful position in the result set if there is at least one failed neighbor within a specified range. When complete, only locations that were completely surrounded by positions that passed the test will remain. As an alternative example, if we want to avoid a collision volume that applies poison damage to agents inside, after filtering out positions inside the volume itself we might use this operation to create a safe buffer zone around it by eroding away locations that are outside the volume, but close enough to the boundary to be dangerous if the agent were to overshoot its destination.

Listing 1 Pseudocode implementation of the erosion operator over the results of a test.

```
def postProcess(results, range):
  processedResults = results.copy()
  for index, result in enumerate(results):
    for other in results:
      if (result.pos – other.pos).length() <= range:
        processedResults[index].passed =
          processedResults[index].passed and other.passed

  return processedResults
```

Dilation works in reverse, converting a failed item to a passing one if any of its neighbors in range have also passed the test filter. This expands the size of a group of positions, making it useful as a complementary operation.

The **gradient** is the intersection of erosion and dilation. We apply both operations to the test set, and keep only those positions which were added by dilation or removed by erosion. This leaves us with points that define the border of the original area, helping us perform tasks such as exploring the frontier of a heatmap, or restricting wildlife to the beach and shallows surrounding a lake.

Finally—but arguably most useful—is the **blur** operation. By blending the score of a position with its neighbors, we soften the edges of the area defined by a group of points, creating a smooth transition from desirable to undesirable areas. Not only does this encourage agents to move more definitively inside an area instead of holding positions at the boundary, it allows suboptimal positions to remain in the set but with reduced utility, allowing the agent to adapt resiliently to poor environmental conditions. A small hiding spot might be undesirable in most cases, but when nothing else is available an agent would still be able to perform a best-effort attempt.

Blur differs from the previous tests in that it is not a binary operation and treats results as continuous values. To implement it, we weight the contribution of each point in range proportionally by their distance to the target point, then calculate a final blurred score as a fraction of the total possible contribution. The following is a pseudocode implementation of a Gaussian blur that performs this operation:

Listing 2 Pseudocode implementation of a gaussian blur operator over the results of a test.

```
def postProcess(results, range):
  processedResults = results.copy()
  for index, result in enumerate(results):
    weightedScores = 0.0
    totalWeight = 0.0
    for other in results:
      if (result.pos – other.pos).length() <= range:
        weight = gaussian_pdf(result.pos, other.pos, range)
        totalWeight += weight
        totalWeightedScore += weight * other.score

    processedResults[index].score = totalWeightedScore /
totalWeight

  return processedResults

// Normal distribution probability density function
def gaussian_pdf(x, m, s):
  return exp(-0.5 * pow(((x – m).length() / s), 2)) * (1 /
(sqrt(2 * PI) * s))
```

Adding a post process operation to a spatial query exposes new spatial information, allowing us to add new types of subtlety and robustness to movement behaviors. However, we can go even further, chaining them together and applying multiple operations to a test in sequence. Following erosion with a gradient operation on a visibility test can produce a shrunken border representing all locations that are solidly hidden, but able to move into a location with line of sight easily—for example, when developing a procedural spawning system that should place new enemies not only out of view, but also where they can quickly engage the player. Following dilation with blur can produce buffer zones with a smooth falloff, giving agents the flexibility to position themselves closer to dangerous areas when no other suitable position is available.

Finally, we can apply morphological operations at different scopes to gain additional flexibility over how spatial areas are manipulated. The examples above operate on a single test, but are also powerful when executed over the results of a subset of tests in a query, or over the final results of the entire query, acting as a global post-processing step.

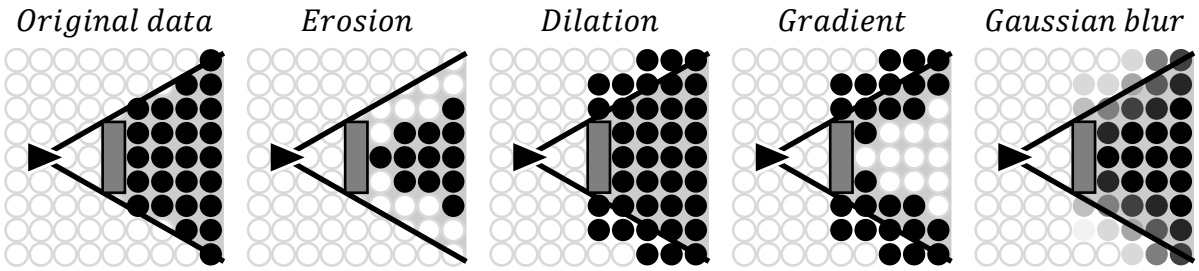

Figure 4 Post-processing operators applied to the results of a visibility test that scores locations hidden from the player. From left to right: Original data, post processing with erosion, dilation, gradient, and Gaussian blur operations.

5.1 Performance and Optimization

In order to identify all points in range of each sample point in the query, the naïve implementations in Listings 1 and 2 execute in $O(n^2)$ time, and as a result will be too slow to use in practice for all but the smallest queries. For a production-ready implementation, it is necessary to minimize this lookup time by first placing the test results a spatial partitioning data structure such as a quadtree or octree, then perform nearest neighbor search over the partitioned space. Even for queries that span large areas, by taking this step the cost of performing a post processing operation can be reduced to a small fraction of the total execution time.

6 Conclusion

A query is more than a collection of tests. Every component, from test weights to normalization to result selection plays an important part in producing rich, nuanced movement behavior, and by knowing when to adjust each piece we can leverage the full expressive power a query system provides. In this article, we learned how to prevent goal instability with hysteresis, blend between multiple query behaviors using test utility, ensure that our test scores are weighted correctly by selecting different normalization methods, and extract additional spatial information from tests to make decisions based on the shape of the results using post processing operators. In this way, by tuning individual components to ensure that the results reflect our intentions, we can make finicky behaviors reliable, good results even better, and put complex, sophisticated behaviors within practical reach.

7 References

[Isla 09] Isla, D., Dill, K., Champandard, A. 2009. Lay of the Land: Smarter AI Through Influence Maps. GDC 2009.

[Jack 13] Jack, M. 2013. Tactical Position Selection: An Architecture and Query Language. In Game AI Pro, ed. Steve Rabin, 337-359. CRC Press.

[Johnson 17] Johnson, E. 2017. Guide to Effective Autogenerated Spatial Queries. In Game AI Pro 3, ed. Steve Rabin, 309-325. CRC Press.

[Zielinsky 13] Zielinski, M. 2013. Asking the Environment Smart Questions. In Game AI Pro, ed. Steve Rabin, 423-431. CRC Press.