29

Escaping the Grid

Infinite-Resolution Influence Mapping

Mike Lewis

29.1 Introduction

One of the central elements of any robust AI system is *knowledge representation*. This encompasses a wide variety of techniques and mechanisms for storing and accessing information about an AI agent's perception of the world around it. The more powerful the knowledge representation, the more effective the AI can be.

A common form of knowledge representation involves measurable (or computable) quantities that vary throughout a region of space. The set of techniques for handling this sort of information is generally referred to as *spatial analysis* or *spatial reasoning*. One of the many powerful tools used in spatial reasoning is the *influence map*.

This chapter will look at the limitations with a traditional 2D influence map and introduce the concept of an infinite-resolution influence map. With this new representation, we'll then explore the issues of propagation, queries, handling of obstacles, and the third dimension.

29.2 Influence Mapping

To construct and use an influence map, three elements are typically involved. First, there must be some kind of *value* that varies throughout a spatial environment. Second, there is sometimes a *propagation method* by which these values change through space and/or time. Finally, there must be a *query mechanism*, which allows the AI system to examine and reason about the values stored in the influence map.

Classical influence mapping is generally performed on a regular 2D grid; more complex variants can also use arbitrary graphs, such as a navigation mesh (navmesh). In either case, the important element is not the representation of space, but the fact that space exists and is relevant to determining some value.

Propagation methods vary by application. For instance, a map used to represent the probability of an enemy occupying a given area (occupancy map) might use a simple combination of setting high values when an enemy is spotted and allowing values to decay and "spread outward" to nearby points over time. A more complex example is a visibility map, wherein the value of the map changes based on how well each point can be "seen" by a particular agent.

Some applications need not perform any propagation at all, such as tactical maps that store the instantaneous locations of various agents. Such maps can be queried directly for as long as their information is deemed up to date, and when the map becomes stale, it can simply be wiped and recomputed from scratch.

In any case, propagation generally consists of two elements: *placement* and *diffusion*. Placement is the mechanism by which new, known values are stored in the influence map. Diffusion is the process that allows influence to spread out, blur, smear, or otherwise travel across the influence map. Diffusion may occur over time or range smoothly across some distance from the nearest "placed" value, and so on.

See Figure 29.1 for an example of an influence map representing the AI's "guess" at the probability of a player being in a particular grid cell. Note that each cell contains a value from 0 to 9 and that these values increase toward areas where the player was last seen. Observe how the obstacles (dark squares) interact with the propagation of the influence.

Queries of the influence map are typically straightforward. Given a point in space (or a relevant point in the representational graph), determine the value of the influence quantity at that point. There may be some value in retaining historical data, so an agent can appear to "remember" recent influence states. A similar but more difficult trick is predicting future influence states, usually by simulating additional time-based propagation.

One final technique worth mentioning is *breadcrumbing*. In this mode, influence is deposited and diffused across the map in an ordinary fashion. When the map is updated, rather than resetting the values and recalculating them based on the relevant placement and diffusion rules, the old values are *decayed* by a given proportion. The new influence values are then added "on top of" the old values. This yields a sort of historical heat map. For example, rather than showing where agents are located *right now*, a bread-crumbed influence map can show where agents have generally tended to concentrate or travel routinely over a given span of time.

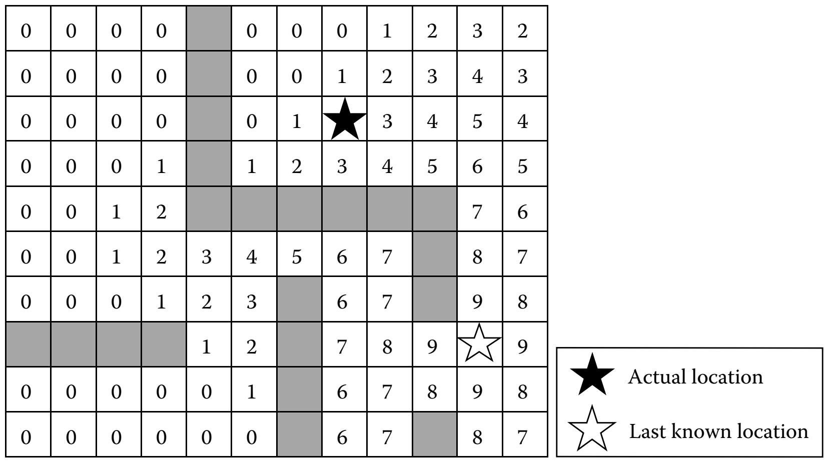

Figure 29.1

This map illustrates the AI's best guesses about the player's location.

29.3 Limitations

Representing a large, finely detailed world as a grid is often cost prohibitive. A graph might help, but even then, the trade-off between resolution and performance must be carefully examined. In many cases, the difficulty of using a graph is not justified, given that they do not solve the resolution issue, and can even compound the performance problems if not managed carefully, due to the need for nontrivial mappings between points in space and nodes in the graph.

Not only are large grids expensive in terms of memory, but they can also consume a vast amount of CPU time during propagation. Each cell must be updated with *at least* the surrounding eight cells worth of influence value, and even more cells become necessary if propagation must reach beyond a point's immediate neighbors. For many applications, this might entail processing hundreds of cells each time a single cell is updated. Even though grid lookups are constant-time operations, the propagation phase can easily become an issue.

One potential approach to mitigate this problem is to use subgrids to minimize the number of cells involved; this can be done in a few flavors. For instance, a quadtree can be established, and empty nodes of the tree can have no associated grid. Another technique is to store small grids local to relevant AI agents or other influence-generating points in the world.

Unfortunately, the quadtree approach requires constantly allocating and freeing grids as influence sources move about. Localized grids can avoid this problem, but at the cost of making propagation and lookups substantially trickier. Both techniques perform worse than a giant, fixed grid when influence must be tracked uniformly across a very large area, for example, due to having agents covering an entire map.

29.4 Point-Based Influence

Consider that most influence sources can be considered point based. They deposit a defined amount of influence onto the map at their origin point and then potentially "radiate" influence outward over a limited range. This spread of influence can occur instantaneously, or over time, or both. For instantaneous propagation of influence, a simple *falloff function* is sufficient to describe the influence value contributed by the influence source at a given location on the influence map.

If the falloff function for a point is well defined, there is no need to represent the influence of that source in a discretized grid. Instead, influence at a *query point* on the influence map is simply equal to the value of the falloff function. Suppose the falloff function is a simple linear decay out to radius r, as described in the following equation:

$$f(x) = 1 - \frac{x}{r} \quad \text{for } x \le r \tag{29.1}$$

The influence value ranges smoothly from 1 to 0 as the distance from the origin increases. For a 2D influence map, the equivalent falloff function is expressed in Equation 29.2, based on an influence source at the point (x_0, y_0):

$$f(x,y) = \max\left(1 - \frac{\sqrt{(x-x_0)^2 + (y-y_0)^2}}{r}, 0\right) \tag{29.2}$$

Naturally, influence maps with only one influence source are rather boring. On the plus side, adding in multiple influence sources is trivial: just add the values of the falloff functions. In formal notation, the computation looks like the following equation:

$$g(x,y) = \sum_i f_i(x,y) = \sum_i \max\left(1 - \frac{\sqrt{(x-x_i)^2 + (y-y_i)^2}}{r_i}, 0\right) \tag{29.3}$$

One highly useful observation is that the influence falloff function need not be trivial and linear; any differentiable function will suffice. (The importance of the ability to compute well-defined partial derivatives of the falloff function will be explored in a later section.) In fact, each individual influence source can use any falloff function desired; the value of the influence at a point remains the sum of the falloff functions for each individual influence source.

Another useful addition to this recipe is the ability to scale individual influence sources so that they do not all contribute the exact same maximum value (of 1, in this case). This is easily demonstrated in the linear falloff model as shown in the following equation:

$$g(x,y) = \sum_i \max\left(s_i \left(1 - \frac{\sqrt{(x-x_i)^2 + (y-y_i)^2}}{r_i} \right), 0 \right) \tag{29.4}$$

Given this set of tools, it is possible to replicate and even improve upon the discretized mechanisms typically used for influence mapping. Note that the value of the influence map can be queried at any arbitrary point with unlimited resolution; there is no loss of detail due to plotting the influence value onto a grid. Further, memory requirements are linear in the number of influence sources rather than increasing quadratically with the size and resolution of the grid itself.

Nothing comes for free, though; in this case, the cost of performing a query increases dramatically from $O(1)$ to $O(n)$ in the number of influence sources. Dealing with non-trivial topologies and obstacles is also significantly trickier, although, as will be explored later, some effective solutions do exist. There is also the matter of performing believable time-based propagation of influence, which is not immediately supported by the falloff function model as described. Last but not least, only one type of query has been detailed thus far; queries such as "find the point in a given area with the lowest (or highest) influence value" have not been considered.

Thankfully, all of these issues can be addressed well enough for practical purposes. Together with some optimizations and some mathematical tricks, this elimination of grids yields *infinite-resolution influence mapping.*

29.5 Making Queries Fast

Perhaps the most worrisome of the limitations of the influence mapping model described thus far is the need for comparatively expensive calculations for each query. Fixing this shortcoming is straightforward but takes some careful effort.

A key observation is that, in most practical cases, the vast majority of influence sources will not be contributing any value at an arbitrary query point. In other words, most of the influence sources can be completely ignored, drastically reducing the cost of a query.

The general idea is to use a *spatial partitioning structure* to make it possible to trivially reject influence sources that cannot possibly have an impact on a particular query. From among the many different partitioning schemes that have been developed, there are a few that seem particularly promising.

Voronoi diagrams are an appealing candidate. Constructing a Voronoi diagram optimally costs $O(n\log n)$ time in the number of influence sources, using the canonical *Fortune's algorithm* [Fortune 86]. The difficulty here is that it is often important for multiple influence sources to *overlap* each other, which fundamentally contradicts the purpose of a Voronoi diagram (i.e., to separate the search space such that each source is in exactly one *cell*).

Search trees are another category of techniques with great potential. Quadtrees are perhaps the most popular of these methods, but there is an even better option available: the *k*-d tree [Bentley 75].

Intuitively, a *k*-d tree is fairly simple. Each node in the tree has a *splitting axis*, a *left-hand branch*, and a *right-hand branch*. The splitting axis determines how the descendants of the tree node are organized. If the splitting axis is *x*, for example, all points in the left-hand branch have an *x* coordinate less than the parent node, and all points in the right-hand branch have an *x* coordinate, which is greater than that of the parent node.

A *k*-d tree is considered *balanced* if it has a (roughly) equal number of nodes in the left-hand branch as in the right-hand branch. Constructing a balanced *k*-d tree is relatively easy. Given a set of points, recursively perform the following procedure:

1. Pick an axis for the splitting axis.
2. Sort all of the input points along this axis.
3. Select the median of the input points from the sorted list.
4. This median node is now considered a *k*-d tree node.
5. Recursively apply this method to all points in the first half of the sorted list.
6. The resulting node becomes the left-hand node of the parent found in step 4.
7. Recursively apply this method to all points in the latter half of the sorted list.
8. The resulting node becomes the right-hand node of the parent found in step 4.

The splitting axis generally alternates between levels of the tree; if a given node is split on the *x*-axis, its two children will be split on *y*, and so on. Other methods for selecting a splitting axis exist but are generally most useful in fairly extreme cases. Experimentation is always encouraged, but for most uses, a simple alternation scheme is more than sufficient. A very simple *k*-d tree constructed in this manner is illustrated in Figure 29.2. The corresponding tree-style arrangement is shown in Figure 29.3.

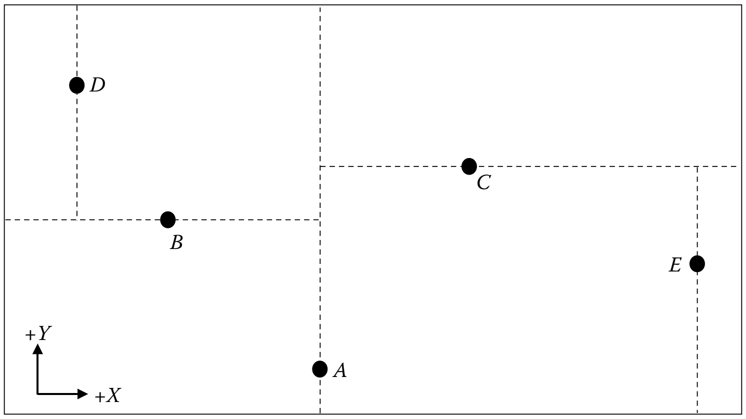

Figure 29.2

The points in this space have been partitioned into a *k*-d tree.

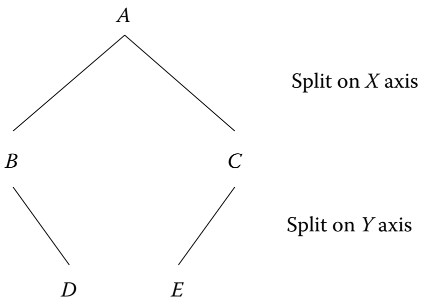

Figure 29.3

This is the same *k*-d tree as Figure 29.2, visualized as a tree structure.

Using this structure, the worst-case search time for points lying in a given range is known to be $O\left(2\sqrt{n}\right)$ for 2D *k*-d trees with *n* points [Lee 77]. The lower bound on search time is $O(logn)$ given that *k*-d trees are fundamentally binary trees.

With these characteristics, the application of a *k*-d tree can substantially increase the performance of the infinite-resolution influence map. However, constructing the *k*-d tree represents a nonzero overhead, largely due to the need for sorting points along each axis. The number of queries desired, frequency of influence source position updates, and number of influence sources must all be carefully considered to determine if this data structure is a net win. As always, performance profiling should be conducted rigorously during such evaluations.

Once the influence source points are partitioned into a *k*-d tree, it is necessary to perform a *range query* on the tree to find influence sources that can potentially affect a given location of interest. The implementation of a range query is simple but has a few subtleties that merit careful investigation.

A range query begins at the root node of the *k*-d tree and recursively walks the branches. If the query region lies in the negative direction (i.e., left or down) from the *k*-d tree node's associated influence source point, the *left* branch of the tree is explored. If the region lies in the positive direction (i.e., right or up), the *right* branch is explored. If at any time the space bounded by a branch cannot possibly overlap the query area, the entire branch and all of its descendants can be ignored. A sample implementation is provided in Listing 29.1.

Note that it is not sufficient to check if the query location lies closer to the left or right branch. Due to the potential for the query area to overlap the splitting axis represented by the *k*-d tree node, it is important to include branches of the tree that *might* intersect the query area. The final two `if` checks in Listing 29.1 represent this logic; since they are not mutually exclusive conditions, no `else` is present.

29.6 Temporal Influence Propagation

An interesting characteristic of traditional influence maps is the ability to propagate influence values through the map over time. This behavior can be replicated in the infinite-resolution approach. Once again, the cost comparison between a grid-based propagation method and the infinite-resolution method will vary widely depending on the application.

Listing 29.1. Example C# implementation of a k-d tree range query.

```
void FindPointsInRadius (
    KdTreeNode node,
    float x,
    float y,
    float radius,
    ref List<KdTreeNode> outpoints
) {
    if (node == null)
        return;

    float dx = x - node.x;
    float dy = y - node.y;
    float rsq = (dx * dx) + (dy * dy);
    if (rsq < (radius * radius))
        outpoints.Add(node);

    float axisdelta;
    if (node.SplitAxis == SplitAxis.X)
        axisdelta = node.x - x;
    else
        axisdelta = node.y - y;

    if (axisdelta > -radius) {
        FindPointsInRadius(
            node.Left, x, y, radius, ref outpoints
        );
    }

    if (axisdelta < radius) {
        FindPointsInRadius(
            node.Right, x, y, radius, ref outpoints
        );
    }
}
```

The number of influence sources, relative density, complexity of obstacles on the map, and so on will all contribute to the performance differences. While temporal propagation is certainly possible in the grid-free design, it is not always an immediate performance win.

To accomplish propagation of influence over time, two passes are needed. In the first stage, new influence sources are generated based on each existing source. These are generally organized in a circle around each original source, to represent spatial "smearing" of the values. A decay function is applied to the old source, and influence is spread evenly among the new sources, to represent falloff occurring over time and conservation of influence as it radiates outward, respectively.

On subsequent timesteps, this procedure is repeated, causing each point to split into a further "ring" of influence sources. If splitting a source would cause its contribution to the influence map to drop below a tuned threshold, the entire point is considered to have *expired* and the source is removed from the map completely.

It is possible to approximate this result by adjusting the falloff function, for example, to a Gaussian distribution function centered on the influence source point. This reduces the number of sources involved but introduces a complication that will be important later on when considering obstacles.

Once influence sources have been added, a simplification step is performed. The goal of this step is to minimize the number of *k*-d tree nodes required to represent the influence present on the map. It must be emphasized that this simplification is *lossy*, that is, it will introduce subtle changes in the values of the influence map. However, these changes are generally very minor and can be tuned arbitrarily at the cost of a little bit of performance.

Simplification is accomplished by looking at each node in the *k*-d tree in turn. For each node, the number of nearby nodes is queried. A *neighbor* is a node close enough to the test node to contribute influence, based on the limiting radius of the applicable falloff function. If too few neighbors exist, the test node is allowed to exist unmodified.

However, if more than a certain number of neighbors are found, their *centroid* is computed by independently averaging the x and y coordinates of each neighbor. The total influence at the centroid is then sampled. If this value is sufficiently large, all of the neighbors (and the test node) are pruned from the *k*-d tree and replaced by a *new* influence source node that attempts to replicate the contributions of the neighbors.

The net effect of this procedure is that tightly clustered groups of nodes can be replaced by smaller numbers of nodes without *excessively* disturbing the overall values in the map. (Note again that some loss is inevitable, although for some use cases, this simplification method can actually produce very pleasing effects.)

29.7 Handling Obstacles and Nontrivial Topologies

While this method works nicely on plain, flat, empty geometry, it lacks a critical component for dealing with *obstacles*, that is, areas of the influence map where values should not propagate. There is also no consideration of more interesting topologies, for instance, where the influence propagation distance is not strictly linear.

As described, things quickly break down when considering influence propagation, because it is important to conserve "energy" as influence spreads outward. Influence that hits an obstacle should not simply vanish, but rather "bounce off" and appear to flow *around* the boundary of the obstruction.

Fortunately, a pretty convincing effect can be implemented with relatively minor adjustments to the propagation/simplification technique. During propagation, before placing a new influence source, the new point is queried to determine if that point is obstructed. If not, the point is placed as normal. If so, the point is not placed at all. Once the set of valid points is established, each one is given a fraction of the original influence source's energy, based on the number of points that were ultimately placed.

This allows influence propagation to flow smoothly through narrow corridors, for example. The result closely resembles something from a fluid dynamics simulation, where tighter constraints cause influence to move faster and farther, as if under higher pressure. By splitting influence points explicitly into new sources, it is possible to guide this "flow" through unobstructed regions of geometry, rather than uniformly outward, as would occur if the falloff function were simply replaced with a Gaussian distribution or similar function.

Another way to visualize this procedure is to consider an influence source as a cellular automaton. In the propagation step, the automaton distributes the energy it carries into a

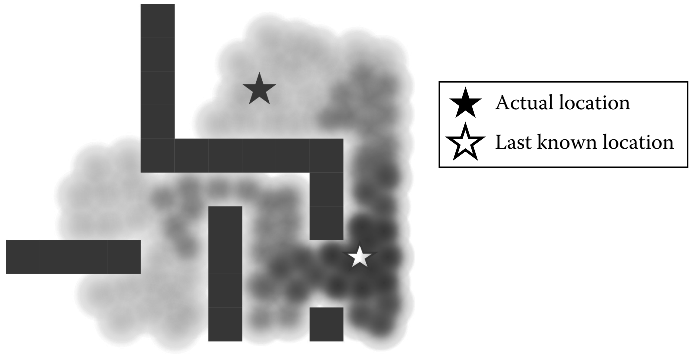

Figure 29.4

This map illustrates grid free, obstacle-aware influence propagation.

number of descendants in the surrounding (nonoccluded) space. During the simplification step, automata are resampled so that those sharing a sufficient amount of space in the world are collapsed into a single "cell." Additionally, automata that have lost too much energy due to decay or splitting are considered to have gone extinct.

If this cellular automata concept is plotted onto a grid, the result looks remarkably similar to that illustrated earlier in Figure 29.1. Allowing the influence sources to move freely rather than being truly cell based yields the effect shown in Figure 29.4, which offers a snapshot view of how infinite-resolution influence mapping handles obstacles during propagation.

Each influence source in this image is propagating a short, linear distance. However, *in totality*, the influence values will range smoothly and fluidly around obstacles, as the figure shows. By carefully controlling the placement of propagated influence nodes, it is possible to represent any spatial characteristics desired.

The methods leading to these results are directly inspired by similar techniques used in the computer graphics world, most notably *photon mapping* [Jensen 01]. In turn, more generalized statistical techniques such as *kernel density estimation* [Rosenblatt 56] are important ancestors of infinite-resolution influence mapping. These approaches provide both the foundations of the methods described in this article as well as ample opportunities for exploring the technique further.

Although it can initially seem limiting to use simple (even linear) falloff functions for representing influence sources, the vast body of prior research in the aforementioned areas should offer some reassurance that nontrivial problem spaces can be very effectively handled using such straightforward mathematical processes.

29.8 Optimization Queries

Influence mapping is valuable for far more than just acting as a mapping of spatial coordinates to arbitrary numbers. Besides just testing the value at a point, there are several other types of query that are commonly employed. In mathematical terms, these are all reducible to *local optimization queries*.

For instance, it might be useful to know the location (and value) of the point of lowest (or highest) influence, within a given area. Looking for such a lowest value is known as *local minimization*, and looking for the highest value is similarly referred to as *local maximization*. More generally, optimization within a restricted input domain is known as *constrained optimization*.

A tremendous volume of research has gone into optimization problems of various kinds. Fortunately, if the influence falloff function is carefully selected (i.e., continuously differentiable), the infinite-resolution influence map optimization problem becomes fairly straightforward.

For some types of optimization problems, there are known techniques for analytically finding an exact solution. Some require processes that are fairly difficult to automate, however. In practice, it is usually adequate to rely on *iterative* approximations to optimization queries, which are much easier to implement in the general case and typically more than fast enough for real-time use.

Specifically, the *gradient descent* optimization method [Cauchy 47] can yield local minima or maxima in a relatively short number of iterations, provided that the function being optimized is continuously differentiable. In essence, gradient descent is performed by looking at a random starting point and computing the slope of the function at that point. Based on that slope, the algorithm moves down (or up) the gradient defined by the function's curve or surface. This is repeated until further steps fail to yield improvements, indicating that a local optimum has been found.

There is a considerable tuning caveat to this technique, which is selecting the distance up (or down) the gradient to travel between steps. Different applications might find very different optimal values for this stepping. It is advisable to calibrate this value using a realistic sample set of influence sources. Optionally, the distance can be "guessed at" by examining the magnitude of the function at the current sample point and attempting to dynamically increase or decrease the step size accordingly.

Gradient descent in two dimensions simply requires partial differentiation of the candidate function and can also be generalized to three dimensions easily enough. The question is, what is the candidate function being optimized? In this case, the influence contribution function is defined again in the following equation:

$$g(x,y)=\sum_i \max\left(s_i\left(1-\frac{\sqrt{(x-x_i)^2+(y-y_i)^2}}{r_i}\right),0\right) \tag{29.5}$$

Recall that this boils down to adding up several individual functions. Conveniently, the derivative of such a summation is equal to the sum of the derivative of each function in turn. This can be succinctly expressed as in the following equation:

$$g'(x,y)=\sum_i f_i'(x,y) \tag{29.6}$$

So all that is necessary to compute the derivative of the influence map function itself is to sum up the derivatives of the influence functions for each contributing influence source.

The partial derivative of the influence function with respect to x is given by Equation 29.7. Similarly, the partial derivative with respect to y is given by Equation 29.8. The vector describing the slope of the influence function is then equal to Equation 29.9:

$$\frac{d}{dx} f_i(x,y) = \frac{-s_i(x-x_i)}{r_i\sqrt{(x-x_i)^2+(y-y_i)^2}} \tag{29.7}$$

$$\frac{d}{dy} f_i(x,y) = \frac{-s_i(y-y_i)}{r_i\sqrt{(x-x_i)^2+(y-y_i)^2}} \tag{29.8}$$

$$\left(\frac{d}{dx} f_i(x,y), \quad \frac{d}{dy} f_i(x,y)\right) \tag{29.9}$$

Note that the influence function is defined to fall off to 0 outside the effective radius of each influence source. This logic must also be included when computing the values of the partial derivatives. In this case, it is sufficient to check the distance from the test point to the influence source beforehand and simply skip the source if its contribution would be zero.

Once this slope at a point can be computed, the gradient descent algorithm can be used to find nearby local optima. Queries for local minima or maxima can be accomplished by performing gradient descent and constraining the distance from the query point to a given radius (or other area). The algorithm for minimization is demonstrated in Listing 29.2. Note that this is an approximation only and will not necessarily find the

Listing 29.2. A pseudocode implementation of gradient descent for a local minimum.

```
maxRadius = searchRadius + NODE_MAX_RADIUS
nodeList = FindNodesInRadius(searchPoint, maxRadius)
minimum = InfluenceAtPoint(searchPoint, nodeList)

for (STEP_LIMIT) {
     if (minimum <= 0.0)
         break;

     gradientXY = GradientAtPoint(searchPoint, nodeList)
     newSearchPoint = searchPoint - (gradientXY * SCALE)
     newSum = InfluenceAtPoint(newSearchPoint, nodeList)

     if (distance(newSearchPoint, originalPoint) > maxRadius)
         break;

     if (newSum < minimum) {
         searchPoint = newSearchPoint;
         minimum = newSum;
     }
}
minimumPoint = searchPoint;
```

true optimal point in a given range without some additional modifications. However, in practice, it works well enough for general use.

Modifying this code for maximization is straightforward: each search step should check for *increasing* values of influence, and the new search point should be found by adding the scaled gradient vector instead of subtracting it.

Of course, the provided equations work only for the linear falloff function defined earlier; again, though, any falloff function can be used provided that it is continuously differentiable. If doing a lot of manual calculus is not appealing, any suitable mathematics package (including several online web sites) can be used to find the partial derivatives of a substitute falloff function [Scherfgen 14]. It is worth noting that more complex falloff functions may be substantially more expensive to compute and differentiate than the linear model.

29.9 Traveling to the Third Dimension

Thus far, influence mapping has been considered only in the context of flat, 2D geometry. While influence maps can be applied to more complex terrain, they often struggle to cope efficiently with overlapping vertical areas, such as multiple floors in a building. Potential solutions for this problem include manually marking up horizontal regions of a map, thereby creating smaller influence maps on each floor of the building, for example.

It is not difficult to find "pathological" cases where 3D geometry simply does not lend itself well to grid-based influence mapping. For games utilizing a navigation mesh, the basic technique can be loosely applied to navmesh polygons instead of squares in a grid. However, this generally equates to a loss of resolution, and a measurably more expensive propagation phase, since mesh adjacency information must be used to spread out influence instead of trivial grid cell lookups. Moreover, if the environment does not lend itself to the use of a navmesh, influence mapping in a cube-based grid is even more problematic in terms of memory and computation time.

For the 2D case, removing grids and location-based networks has a number of advantages. In the 3D case, avoiding voxelization of the world space makes infinite-resolution influence mapping an intriguing possibility. However, it is important to note that the propagation techniques discussed earlier will become accordingly more expensive in three dimensions, due to the increased number of influence sources generated during each propagation step.

Getting the infinite-resolution technique to work in three dimensions is pretty simple. The *k*-d tree data structure generalizes to any number of dimensions—in fact, the name stands for "*k*-dimensional tree." Adding a third dimension is simply a matter of adding another splitting axis option when partitioning points into the tree.

Point-based influence is also simple to generalize to three dimensions. The linear, distance-based falloff function is adjusted to compute distance based on all three coordinates instead of only two. Summation of contributing influence sources works exactly as in two dimensions.

Temporal propagation can be appreciably more expensive due to the extra points needed to effectively spread influence out over time. However, aggressive simplification and careful tuning can mitigate this expense. For very geometrically sparse environments, one alternate option is to increase the radius of an influence source as it spreads, rather than adding new points. Note that this might complicate *k*-d tree construction and traversal, however, due to the increased maximum effective radius of any given influence source.

Last but not least, optimization queries can also be carried forward into the third dimension with relative ease. The partial derivative of the falloff function with respect to *z* follows the same pattern as the first two derivatives, and as with computing the influence value itself, computing the slope (or *gradient*) of the influence function is just a matter of summing up the partial derivatives for each axis.

29.10 Example Implementation

A complete implementation of the 2D form of infinite-resolution influence mapping can be found on this book's website (http://www.gameaipro.com/). The demo is written in C# for simplicity. While it should not be considered a performance benchmark, the speed is appreciable and illustrates the potential of the technique to provide a powerful alternative to grid-based approaches.

The demo is divided into two sections. One displays a raw influence map with a variable number of influence sources placed in a configurable pattern (or randomly). This view includes the ability to perform optimization queries (specifically minimization) against the generated influence map and can show the gradient descent algorithm in action. The second section of the demo shows how temporal propagation works, including moving influence around a simple set of obstacle regions.

All code in the demo is thoroughly commented and includes some tricks not covered in detail here. It is worth repeating that this example is written for clarity rather than performance, so an optimized C++ version may look substantially different.

While the influence mapping techniques illustrated are not (to the author's knowledge) presently in use in games, the foundational mathematics is certainly highly proven in other spheres, again notably including computer graphics in the form of photon mapping. Examination of the demo code is strongly encouraged, as an animated visual demonstration is far more illustrative than simple descriptions in prose.

29.11 Suitability Considerations

Infinite-resolution influence mapping is not a drop-in improvement for standard grid-based techniques. Before applying it to a particular problem space, it is important to consider several factors that may heavily affect performance and suitability. Although there are no hard and fast rules for these considerations, careful planning up front can be invaluable.

29.11.1 Influence Source Density

Due to the use of a *k*-d tree to store influence source points, the density of sources can be a major factor affecting performance. While having a few dozen points in close proximity is no issue, scaling to thousands or tens of thousands of overlapping influence sources would pose a significant challenge. Note that many thousands of sources are easily handled if they are spatially distributed such that most sources do not overlap; this allows the *k*-d tree to ignore a larger proportion of nodes that cannot possibly be relevant to a given query.

29.11.2 Query Point Density

If the locations of queries against the influence map are relatively well distributed and sparse, infinite-resolution techniques may be very well suited. However, as queries tend to cluster or repeatedly sample very small areas, the cost of recalculating the influence contributions (and, even more so, of performing optimization queries) can become prohibitive. Use cases that need to issue large numbers of queries in dense proximity are likely better served using a grid-based approach.

29.11.3 Update Frequency

One of the most expensive steps of the infinite-resolution method is rebuilding the *k*-d tree. When updates to the influence map can be batched and performed in a single pass, the cost of propagation, simplification, and *k*-d tree reconstruction can be significantly amortized. However, if updates need to occur very frequently, or in a series of many small, independent changes, the costs of keeping the data structures up to date may be come problematic.

29.11.4 Need for Precise or Extremely Accurate Results

Fundamentally speaking, infinite-resolution influence mapping works by discarding information and relying on statistical approximations. These approximations tend to lead to subtly less precise and less accurate results. For cases where the production of exact influence values is important, grid-based methods are almost certainly preferable.

29.12 Conclusion

Influence mapping is a remarkably effective tool for knowledge representation in game AI. Traditionally, it has been confined to 2D grids and relatively simple connected graphs, such as navigation meshes. Representing large, fine-grained, and/or 3D spaces is typically very difficult if not outright impractical for influence maps. Escaping these limitations is deeply appealing.

Truly 3D, freeform influence mapping can be a remarkably powerful tool. This power, as is often the case, comes at a cost. While the infinite-resolution method is certainly very efficient in terms of memory storage, it can become highly costly to query and update influence maps created in this fashion.

As with any such performance trade-off, there will be situations where infinite-resolution influence mapping is useful and other cases where it is not applicable at all. Care should be taken to measure the performance of both techniques within the context of a specific use case. Like all game AI techniques, this one has its limitations. Given the right circumstances, though, it can be a very powerful tool in any AI developer's arsenal.

References

[Bentley 75] Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18(9): 509.

[Cauchy 47] Cauchy, A. 1847. Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu des Séances de L'Académie des Sciences XXV, Vol. Série A* 25: 536–538.

[Fortune 86] Fortune, S. 1986. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, Yorktown Heights, NY, pp. 313–322.

[Jensen 01] Jensen, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. Natick, MA: A.K. Peters.

[Lee 77] Lee, D. T. and Wong, C. K. 1977. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica* 9(1): 23–29.

[Rosenblatt 56] Rosenblatt, M. 1956. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics* 27(3): 832.

[Scherfgen 14] Scherfgen, D. 2014. Online derivative calculator. http://www.derivative-calculator.net/ (accessed January 17, 2015).