31

Spatial Reasoning for Strategic Decision Making

Kevin Dill

31.1 Introduction

In the real world, no military decision can be made without taking into account the characteristics of the space in which the decision is taking place. Individual soldiers will use cover and concealment, so as to maneuver on the enemy without being shot. Squad leaders will split their team, with one element pinning the enemy in place while the second maneuvers to attack from a flank. Platoon leaders emplacing a defensive perimeter will use their largest weapons to cover the most likely avenues of approach, and use resources such as minefields and barbed wire to constrain the enemy's maneuverability—forcing them to attack where expected. Beyond small unit tactics, spatial considerations are taken into account in decision after decision—from the placement of diapers in a toy store (at the back, so that parents have to take their children past the toys) to the placement of a city (in a good location for trade, with ready access to water).

While there are exceptions, game AI often does a poor job of thinking about space in these ways. A common justification is that the developer doesn't want to make the game too hard for the player—but there are any number of ways to balance the game, including simply making the enemies weaker or forcing them to make intentional mistakes that the player can exploit. Further, imagine the benefits of making the AI more spatially competent. As Koster so eloquently explained in *A Theory of Fun for Game Design* [Koster 04], one of the things that makes a game fun is *mastery*. Imagine an enemy that could use good spatial tactics and strategies against you in a difficult but not overwhelming way. If the player can learn from the strategies of their enemy, then they are being *shown* how to take advantage of the environment themselves, and we can gradually build upon that experience, imparting mastery step by step much like the game *Portal* gradually imparts mastery in the use of a teleportation gun.

In this chapter, we present a successful spatial reasoning architecture that was originally developed for the real-time strategy (RTS) game, *Kohan II: Kings of War* ("Kohan II") [Kohan II]. This architecture primarily addressed strategic needs of the AI, that is, the large-scale movements and decisions of groups of units across a map.

Spatial reasoning is, by its very nature, highly game- and decision-specific. The information that needs to be tracked and the techniques that will work best depend greatly on the specific nature of the experience that you're trying to create for your players. Further complicating the issue is the fact that the capabilities of your game engine will often drive you toward specific representations of space and away from others. As the approaches in this chapter were largely drawn from *Kohan II*, a tile-based strategy game, our solutions have that flavor about them. Nevertheless, many of the ideas discussed here are directly applicable in a wide range of domains, and others might spur your own ideas for spatial considerations. Our ultimate hope is to spark your imagination and get you thinking about (1) how space is important to your players' experience and (2) how you can represent and reason about that. From there, it's a small step to finding the right solutions for your own game.

It is worth noting that the *Kohan II* AI was discussed in *AI Game Programming Wisdom 3* [Dill 06]. While it's not necessary to read that chapter in order to understand this one, it will help to clarify the larger picture and in particular how some of the techniques discussed here might be used in the context of a utility-based AI.

31.2 Spatial Partitioning

Before any spatial reasoning can be done we need a *spatial representation*, which is created by breaking the game map up into meaningful *regions*. Regions will form the basis of most of the spatial reasoning techniques we will describe, including techniques for driving exploration, picking attack locations, path planning, and spatial feature detection.

The ideal subdivision is of course going to be game-specific, as you create regions that represent the spatial characteristics important to the decisions you want to make. With that said, here are some general characteristics of good regions as they are defined in Kohan II—just keep in mind that (with the possible exception of homogeneity, depending on your intended use) these are intended as fuzzy guidelines, not hard requirements:

- *Homogeneity*: It's often a good idea to have regions composed of a single "type" of space. For example, in a strategy game, your regions might divide space by terrain type, with each region being composed entirely of land, water, or mountain tiles, perhaps further subdivided to call out features like hilltops and roads that are of particular importance. Similarly, in a shooter with indoor maps, you might divide space into room and corridor regions, perhaps with smaller regions to represent areas of cover and concealment, potential snipe points, and so forth. The important thing is that each region should contain exactly one type of space—so a cover region should contain *only* cover, a hilltop region should contain *only* the hilltop, and so forth. This way, the AI knows that anywhere it goes in that region the expected characteristic will pertain.

- *Not too big*: If the regions are too large, then the reasoning becomes muddy and inaccurate—there isn't enough detail represented to make meaningful decisions. At the ridiculous extreme, we could treat the whole map as a single region—but this wouldn't be helpful to the AI at all.
- *Not too small*: If the regions are too small, then we can get buried in details. This can have profound performance implications, as many terrain reasoning techniques are search based and thus potentially expensive. More to the point, wherever possible (and this can be a difficult balancing act), we would like all of the features that go into making a decision about a region to be contained within that region or at least in a few adjacent regions. For example, if considering attacking a settlement in an RTS game, we'd like that settlement to be contained in a single region. If considering an army, we'd like the entire army to be contained in a single region. This is often not possible—armies typically move around and cross region borders freely, for example—so we will often need to consider characteristics of nearby regions when making decisions. Techniques exist for this, the most obvious being influence maps (which are discussed in detail in Section 31.4), but the fewer regions the army traverses, the better our decisions about it are likely to be.
- *Roughly equilateral/square/hexagonal/round*: Long, skinny regions tend not to be ideal—they can easily be too small in one dimension, and too large in the other. Of course, this is an ideal that depends very much on the situation. Long, skinny regions make perfect sense when representing long skinny things, such as a corridor in a building or the area along the side of a country road where the combination of a ditch and a stone wall may provide improved cover.
- *More-or-less convex*: Convexity is not an absolute requirement, even if you're going to use your regions for high-level path planning—in fact, it is often overrated. Nevertheless, if your spatial partition algorithm creates regions that are badly concave (e.g., a region shaped like a giant letter L or like Pacman with his mouth half open), you may want to consider taking a pass over all the regions and splitting these ones in half, particularly if they're large. As a rough rule of thumb, if the center of a region is not inside the region then the region is too concave.

31.2.1 Region Generation

There is no one right way to generate regions. When possible, it's a good idea to lean on the tools provided by your game engine. Techniques that you might consider include the following:

- *Designer defined*: For many games with predefined maps, designers can hand annotate the maps. This gives the most control over the player's experience and can produce very good results, but isn't possible on large open-world games or games with random maps.

- *Navmesh based*: If your game has a navmesh, then it's possible to simply use that, although this may not result in very good regions—many navmesh generation algorithms create regions that are long and skinny or that are widely varying in size. One partial solution is to automatically combine and/or subdivide the navmesh cells to produce better quality regions, perhaps sacrificing a bit of convexity to produce regions that are closer to round and of appropriate size. Of note, if you decide to head in this direction, consider using a technique such as conforming Delaunay triangulation for your navmesh generation—it will provide triangles that are closer to being equilateral (and thus not long and skinny) than many other approaches [Tozour 02, de Berg 08].
- *Tile based*: If you have a tile-based map (or can temporarily lay tiles over your map), you can use a flood-fill algorithm to create your regions. This is the approach that we used in *Kohan II*, so we will discuss it in detail.

For *Kohan II*, we used a greedy flood-fill-based approach to create rectangles from our grid tiles and then, in certain cases, combined adjacent rectangles to create our regions. It wasn't a perfect approach. It was time consuming (on a 2005 era machine, with a large map, it could take well over 20 seconds, so we ran it at load time only) and would sometimes create regions that were long and skinny or oddly shaped. Nevertheless, it was simple to implement and debug and makes a good basis for the rest of this chapter.

The logic for rectangle creation was as follows:

1. Start a new region that consists of a single tile. This tile should be one that has not yet been placed into a rectangle, and that is the farthest to the bottom and left of the map.
2. Using that tile as the bottom-left corner, find the largest homogenous rectangle that we can. We do this by alternately expanding the rectangle one row to the right and one row up, as long as this expansion results in a rectangle that is homogeneous.
3. If the resulting rectangle is "too big," divide it horizontally and/or vertically, creating two or more rectangles of a more reasonable size—for instance, if our maximum width is 16 tiles, then a 20 × 45 rectangle would be divided into six 10 × 15 pieces.

This algorithm results in rectangles of terrain that are homogeneous, generally roughly square (although this is not guaranteed), convex, and not too big. However, it can create a lot of very small regions along the borders of different types of terrain (e.g., along the edges of rivers or mountain ranges). These regions are not only too small, they also often aren't very interesting (not a lot of importance happens right on the edge of impassable terrain). Consequently, we would prefer to somehow get rid of them.

The first step to addressing this is to take a second pass over all of the rectangles looking for the undesirable ones. For example, we might search for rectangles that are less than two tiles wide in one direction or the other and that contain less than ten total tiles—although the details of how to tune that are of course game-specific. Whenever we find one of these rectangles, we try to attach it to an adjacent rectangle of the same terrain type. We only do this if the resulting region doesn't exceed some maximum height and width and if its center remains inside of its borders. The result is that we

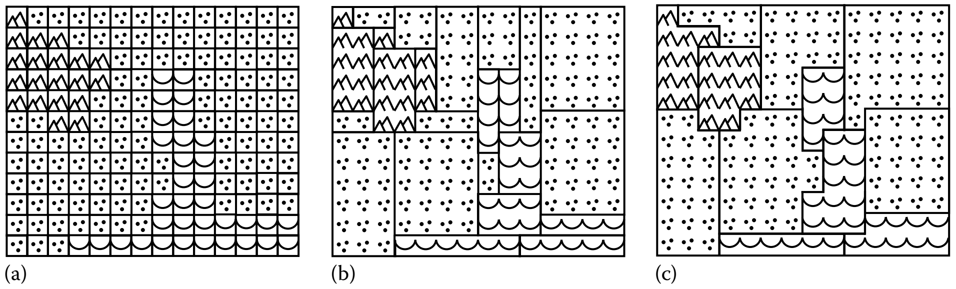

Figure 31.1

A hypothetical tile-based map with land, water, and mountain terrain (a) and the same map divided into rectangles (b) and regions (c).

get a smaller number of more useful, slightly larger, occasionally nonconvex regions. Figure 31.1 shows this entire process step by step.

In *Kohan II*, we retained all three representations. The tiles were useful for low-level path planning and collision avoidance, although we did allow movement within tiles. The rectangles gave a compact representation for the regions (a region consisted of a list of rectangles, and a rectangle simply stored the bottom-left and top-right tiles). Finally, the regions were used for a wide variety of spatial reasoning and decision-making tasks.

31.2.2 Static vs. Dynamic Regions

For many games, it is adequate to use static regions—that is, regions that don't change during gameplay. Static regions are generally baked into the map during development or, if you have random maps, created when the map is generated. Some games, however, have destructible or user-changeable terrain and, as a result, need to be able to update their regions as the map changes.

Dynamic region calculation is a difficult and thorny problem—think hard before deciding to include it in your game! With that said, it's not insoluble. For tile-based maps, a fast localized flood-fill-based solution can work—this is the approach that was taken by *Company of Heroes* [Jurney 07], which was probably the first game to solve this problem. If you're using a triangulation-based navmesh to provide your regions, it is possible to retriangulate the affected area reasonably quickly (e.g., Havok AI does this). Note that neither of these approaches is simple to implement or optimize, and no matter how much you optimize, neither is likely to run in a single frame. Thus, you're going to have to timeslice the calculations and also to figure out how the AI should handle the case where the regions have been invalidated but haven't yet finished recomputing. Regardless, dynamic recalculation of the regions is well beyond the scope of this article (but *Computational Geometry: Algorithms and Applications* might provide at least a starting point [de Berg 08]).

31.3 Working with Regions

So, now we have our regions… what do we do with them? Well, lots of things, but let's start with some examples that take advantage of the granularity and homogeneity of our regions—that is, of the fact that they're not too big, not too small, and made up entirely of a single type of space.

31.3.1 Picking Places to Scout or Explore

As discussed earlier, regions allow us to break the map into larger, less granular, spaces, which can greatly reduce the complexity of decision making. One area where this pays off is in selecting areas to explore or to scout for enemy forces. If we made this decision on a tile-by-tile basis, it would be entirely intractable—there are simply too many tiles to choose from. Instead, we can score the value of exploring a region using information such as the following:

- How much of it is already explored?
- How close is it to our current position?
- Does our current strategy prefer to explore close to the center of our territory (looking for resources) or far away (looking for enemy bases), and how does this region match that preference?

Scouting is similar to exploration, except that the decision is based on how recently that region has been scouted, its proximity to our territory, its proximity to known enemy territory or units, its strategic importance, and so forth.

Of course, once you've picked a region to explore or scout, you need a low-level AI that handles the actual motion over the tiles, perhaps spiraling outward until a new location is picked—but this is generally much more tractable than the high-level decision selecting a general area to explore.

31.3.2 Picking Places to Attack

Regions can also allow us to lump nearby structures or resources together and make decisions based on their aggregate value. For example, deciding where and when to attack is an important part of any strategy game. Lots of factors might go into this decision—the strength of our available units, the enemy strength in the area, the strategic value of the region, etc. Many of these factors are discussed elsewhere in this chapter, but the underlying basis for this evaluation is always going to be the economic and/or military value of the target being attacked.

What we want to avoid is launching multiple separate attacks against targets that are very close together, with each attack individually bringing enough units to defeat the enemy forces in the area. This results in sending far too many units to attack, thus depriving us of the opportunity to use those units in other ways. In order to do this, instead of attacking individual targets, we launch our attacks against *regions*. The value of attacking a region can be considered to be something like the sum of the values of all of the targets in that region, plus 25% of the value of targets in adjacent regions—or zero if there is an adjacent region that has a higher total value.

The result is that if we do launch multiple attacks, then those attacks will be spread out, hitting relatively distant targets. This forces the player to fight a battle on two fronts, which is generally a challenging (and enjoyable) experience.

31.3.3 Picking Unit Positions

In addition to their granularity, the homogeneity of the regions can help drive spatial decision making at the unit level. For example, imagine that our archer units have a bonus to their ranged attack value when they're above their enemies, that all units move slower when travelling uphill, and that spearman units have the ability to hide in tall grass and then spring out and ambush the enemy with a large bonus to their attack.

When picking a place to position a unit, we first need to identify the valid regions. Generally speaking, these are the regions that allow the unit to satisfy its high-level goal. For example, if the unit is participating in an attack, then the region must be close enough to attack the enemy, or if it's standing guard, then the region must be close enough to cover the target it's defending (and ideally located on a likely avenue of approach—more on that in Section 31.5.3). Next, we calculate a score for each of the valid regions based on a number of factors, including the proximity to the unit's current location, the cost of moving there, and the viability of moving there (e.g., it's not a good idea to move your archers through a mass of enemy swordsmen just to get to a good tactical position).

In addition to those factors, however, we can consider the value of the region itself. For example, archers might really like hilltop regions—not only do they get a bonus to their attack but they also have more time to fire at enemies coming up the hill to attack them. Spearmen might have a moderate preference for grassland regions, but only when guarding, since they won't have time to hide during an attack, and only when the region is on a likely avenue of approach, since they can only ambush enemies that go past them.

31.3.4 Path Planning

If our regions are created out of some smaller abstraction, such as tiles, then we can use the regions for hierarchical path planning, which will greatly speed up our path queries. To do this, we first find the shortest path through the regions (which is comparatively fast because the regions are moderately large so there are relatively few of them). Once we have the region path, we find the first region on the path whose region center is not visible from the current position of the unit we're planning a path for. On a tile-based map, we can do this with a series of tile walks, which are also quite fast (linear time on the length of the tile walk). A tile walk is similar to a raycast, except that instead of testing against the collision geometry, it simply determines which tiles are along the line. Finally, we calculate the low-level path (using the tiles) from the unit to that region center. This low-level path should also be very quick to find, because it's more or less straight (it typically has a single turn), which means that A* with a straight-line heuristic will find it in near-linear time (i.e., really, really fast—this is the best case scenario for A*).

One nice thing about this approach is that our regions (i.e., the nodes in our high-level abstraction) are homogeneous, which means that we can weight the high-level path based on the terrain type. For example, we can make a unit that prefers to travel through grassland, or to avoid hilltops, or both, by making regions of those terrain types more or less expensive to travel through. While hierarchical path-planning solutions have been around for some time, they don't typically use a homogenous abstraction, so they lose the ability to weight the path in this way—and thus the ability to choose their paths on the basis of anything other than shortest distance.

In the remainder of this chapter, we'll refer to the connectivity graph through the regions as the *region graph* and to a path through this graph as a *region path* or a *high-level path*.

31.3.5 Distance Estimates

One thing that we frequently want is an estimate of the distance between two distant objects. For example, if we are considering whether to attack an enemy settlement with a particular unit, then we will want to know the distance between that settlement and the unit. If we are considering exploring a particular region, then we will want to know the distance to that region not only from the exploring unit but also from each of our settlements. We could use the full path planner to find these distances, but doing so is likely to be prohibitively expensive, especially if we're going to be doing a lot of these checks (and we will almost certainly be doing a lot of these checks if we want to have good spatial reasoning). The obvious simplification is to use straight-line distance, but if your map is at all complex then straight-line distance often badly underestimates the actual distance.

The solution is to find the region path, which is fairly quick, and then calculate the distance from region center to region center along that path. The result isn't a perfect estimate—it will overestimate the distance to some extent. The good news, however, is that the extent to which it overestimates the distance is bounded by the size of the regions, so unless your regions are enormous, this estimate can be expected to be fairly good.

31.3.6 Region Path Caching

Given the frequency with which we'll be using distance estimates, even finding a region path may not be fast enough. In this case, if you can spare a few megabytes of memory, it's possible to cache the region paths as you discover them. In order to do this, you need to have your regions numbered from $0 \ldots n-1$, where n is the number of regions in your game. You can then create an $n \times n$ lookup array. In this array, you store the first step to get from every region to every other region. So, for example, if the path from region 17 to region 23 is 17 → 12 → 23, then position [17][23] in the array would contain 12 (because the first step to get from 17 to 23 takes us to region 12). Similarly, position [12][23] would contain 23, position [23][17] would contain 12, and position [12][17] would contain 17.

If the region count is low enough, this "next step" lookup array could be precomputed fully at map generation time using an algorithm such as Floyd–Warshall [Floyd 62], which runs in $O(n^3)$ in the number of regions. If this cost is prohibitive, however, the contents of the array could also be computed in a just-in-time fashion. A single high-level path doesn't take long to calculate, so you can always run the path planner the first time you need a path between two particular regions and then store them in the path cache as you go. This works out well, because there are generally fewer units to work with (and thus, less work for the AI to do) early in the game, which gives us more time for path planning. In addition, even though the AI will ask for a lot of distance estimates all at once, many of them will start and end in the same general area. If the path planner uses the path cache to look up shortest partial paths, then groups of queries from similar locations will be very fast because most of the paths are just slight modifications of previous searches.

Clearly, one concern is that this array can get quite big if you have a lot of regions. If you have 2000 regions, and you store the region indices in 2-byte integers (since 1 byte isn't big enough), then the size of the array is $2000 \cdot 2000 \cdot 2 = \sim 8$ MB. That's not a completely unreasonable amount of memory on a modern system—although it's not viable on something like the Nintendo 3DS or a cell phone—but it's enough to catch the attention of the rendering team (who will no doubt want the space for more polygons). It's possible to optimize this a bit by compressing the region indices into less than 16 bits (e.g., we can handle 2000 regions with 11 bit indices), but this makes the code quite a bit more complicated for only an incremental savings.

Another trick is to only store the shortest path from the lower-numbered region to the higher-numbered region. Since the shortest path from region 17 to region 23 is the same as the shortest path from region 23 to region 17 (assuming that we don't have any unidirectional connections, like jump downs or one-way teleports), we can just store the first step from the lower-numbered region to the higher-numbered region. Working out the sequence of queries to rebuild the path is a bit tricky, but doable. The essence of it is that you work from both ends toward the middle, filling in pieces until you have the full path. So, in our example earlier, if we wanted to find the path from 23 to 17, first, we'd look up the first step from 17 to 23 (because we don't store the first step from 23 to 17) and find that it is 12. This tells us that we need to go from 23 → [unknown] → 12 → 17. Next, because 12 is less than 23, we look up the path from 12 to 23 and find that we can go directly there. Thus, our final path, as expected, is 23 → 12 → 17. If implemented properly, there should always be a single [unknown] location as we fill in the path—the trick is to look up the next step from the lower-indexed to the higher-indexed region adjacent to that location.

31.3.7 Recognizing Cul-de-Sacs and Chokepoints

Cul-de-sacs are areas that have only a single point of access (like dead-end streets). *Chokepoints* are narrow places in the terrain that connect two or more larger areas. Previous work has discussed a variety of ways to detect and make use of these features, but these approaches are often complex and computationally expensive [Forbus 02, Obelleiro 08] or require that the region graph be extremely sparse [Dill 04]. Unfortunately, most games have maps that are fairly open and easily traversable, which means that the region map is typically not sparse at all.

If your map has been divided into regions, then cul-de-sacs are dead simple to recognize. A cul-de-sac is simply a region that is passable (i.e., units can move through it), which has only one adjacent passable region. Looking back at the map in Figure 31.1c, there are three cul-de-sacs (though on a less constrained map, they would not be nearly so common): the southeast, southwest, and northwest land regions.

Detecting chokepoints is a bit more complicated. One approach is to look for a region R that separates the adjacent regions into two (or more) groups, which we will call *areas,* that are only connected through R. For example, in Figure 31.2a, region 2 is a chokepoint because you can only get from region 1 to region 3 by passing through it. In Figure 31.2b, it is not (at least, not according to this definition), because there is an alternate route around to the west. We can detect regions like this by doing a breadth-first search from each region adjacent to R and excluding R itself from the search. If the search from any adjacent region fails to find any other adjacent regions, then those two adjacent regions must be in different areas and R is a chokepoint.

There are two major problems with this solution. First, it is expensive to compute (unbounded breadth-first search from every region adjacent to every other region… yick!). Second, it is too strict. For example, in Figure 31.2b, region 2 is not considered a chokepoint, but in reality this region and the region directly to its west between them do a nice job of dividing the map into two areas. More generally, a region can be useful as a chokepoint even if there's an alternate route between the adjacent areas, as long as the alternate route is convenient or costly to take.

Fortunately, the solution to both of these problems is the same. We can simply limit the depth of our breadth-first search. This keeps the cost of the search down and also reflects

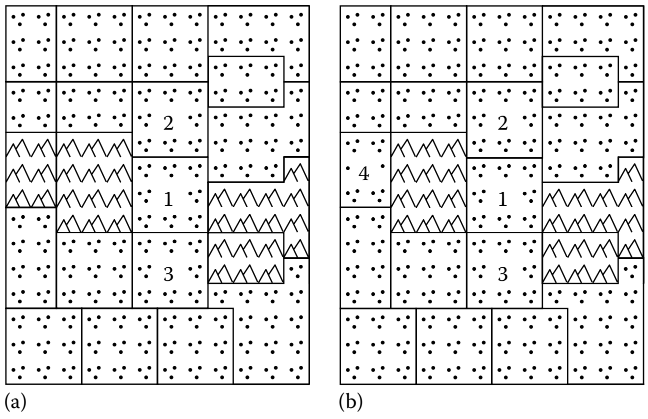

Figure 31.2

(a) Region 1 is a chokepoint because you can't get from region 2 to region 3 without going through it. (b) There are no chokepoints because there is an alternate route around the mountains to the west (through region 4).

the fact that a region can be a chokepoint merely by creating a local division between two areas—it doesn't have to create a global division between them. Finding the right depth for the search is another of those fuzzy, game-specific problems, but in general, a value somewhere between 5 and 10 is probably good for the sorts of maps we've been discussing.

Of note, both cul-de-sacs and chokepoints are generally only interesting if they are reasonably narrow. This is one of the reasons why we want to ensure that our regions aren't too large. As long as the regions have an appropriate maximum width, that width can serve as our definition of "reasonably narrow."

Once you have identified the chokepoints, there are all sorts of ways that you can put them to use. They are often excellent ambush points, for example. One trick is to keep track of how much enemy traffic goes through each chokepoint (or is likely to go through each chokepoint—more on this in Section 31.5.2) and place ambushes on the high-traffic spots. Furthermore, if you can identify a set of chokepoints that separate your territory from the territory of an enemy (i.e., they're along likely avenues of approach—again, more on this is discussed in Section 31.5.3), then they become excellent places to put defensive units and/or fixed defenses (such as forts or walls). This allows you to mass your defenses in fewer locations, giving you a better chance of defeating an incoming attack. At the very least, placing a few forces to stand guard there can warn you that an attack is on the way (in the Army, we called this an LP/OP, or listening post/observation post). Finally, if the chokepoints constrain movement then you might be able to mass your forces at the exit of a chokepoint and defeat the enemy as they come out of it. This tactical advantage may enable a small force to defeat a much stronger enemy—the Battle of Thermopylae, in which 300 Spartans famously held off a massive Persian army (for a time, at least), is perhaps the best known example of this.

On some maps, chokepoints can also be used to plan an attack. If the map is fairly constrained, you may be able to find the chokepoints closest to your target and stage your attacking forces into a multipronged attack, simultaneously striking from two or more different chokepoints. Alternately, you can launch a diversionary attack at one chokepoint and then send a much larger force against the other. As in real life, these sorts of maneuvers can be difficult to time and can go horribly wrong if they're detected in time—but when you're an AI opponent, allowing the player to defeat an overly clever maneuver might be the whole point!

Cul-de-sacs are generally less interesting than chokepoints precisely because they are out-of-the-way spaces that are not normally visited very often. On the other hand, they can be good places to hide economic buildings or other things that we don't want the player to see. We also might be able to optimize some aspects of our AI by excluding the cul-de-sacs from consideration.

One last note on chokepoints and cul-de-sacs: if a chokepoint only has two adjacent passable regions, and one of those regions is a cul-de-sac, then the chokepoint can also be considered part of the cul-de-sac. In other words, if the only place a chokepoint leads to is a dead end, then the chokepoint is really part of the dead end as well. Recognizing this can help to avoid placing units in a region that you think is a chokepoint when in fact nothing interesting is likely to happen there.

31.4 Influence Maps

Influence maps (as they are used in video games) have been around at least since the 1990s [Tozour 01]. They are typically used in the context of tile-based maps [Hansson 10] or on tile-like abstractions such as those proposed by Alex Champandard [Champandard 11]. With a few minor adaptations, however, they can be applied to regions with outstanding results.

31.4.1 Influence Map Basics

The key idea behind an influence map is that each unit has an *influence*, which is *propagated* out from the unit's position. The propagated influence of a unit decreases with distance in accordance with some formula, the simplest of which is simply to decrease the influence by some fixed amount for every tile that you traverse. The overall influence in each tile is the sum of the propagated influences of all of the units on the map.

As an example, consider Figure 31.3. In Figure 31.3a, we see a single lefty unit (whose icon and influence appear on the left side of the tile) with an influence of 5. This influence could represent any property that we wish to track over space, but for the purposes of our example let's assume that it is the combat strength of the unit. Influence propagates outward from the unit, so that adjacent tiles have an influence of 4, and the next tiles beyond those have an influence of 3, and so forth. Thus, on any tile, we can see the influence of this unit—which is to say, how relevant the unit is to combat decisions made with respect to that tile. In the tile where the unit is located, it exerts its full combat strength, but in more distant tiles, which it would have to travel for some time to reach, its influence is decreased.

In Figure 31.3b, we see how the influence of multiple units can be combined to give the overall influence of a player. Thus, instead of a single unit, we have three lefties, each with an influence of 5. The influence in any tile is simply the sum of the influences of the three units and represents the amount of combat strength we can expect that player to be able to bring to bear on a fight that occurs in that tile. There's no particular magic to calculating this—you can simply set all the influence values on the map to 0 and then go

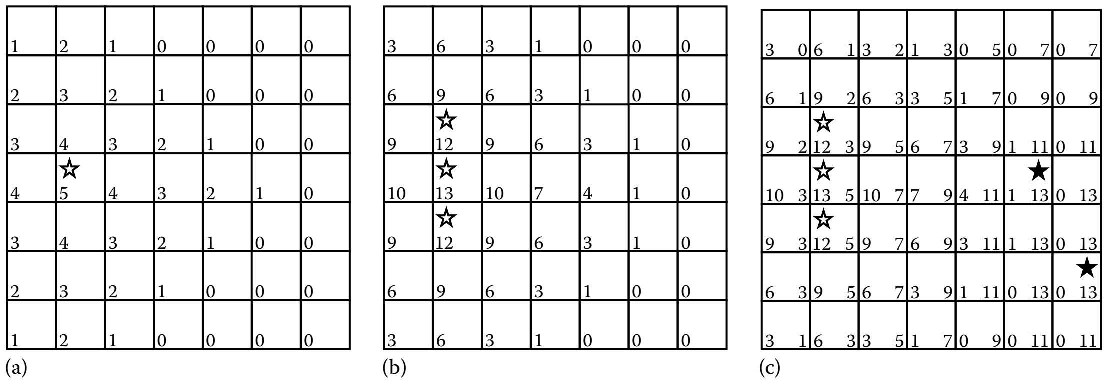

Figure 31.3

An example of an influence map with a single unit (a), three units belonging to the same player (b), and units belonging to two different players (c).

through each unit and add its influence to the map. Addition is commutative, so it doesn't matter what order you add the units in—the end result will be the same. Also of note, in this example we show each unit in its own tile, but the technique works exactly the same if you have multiple units in a tile—simply propagate their influence one at a time and the math will work.

Figure 31.3c is where things start to get interesting. Two righty units have come into the area. These units are a bit more powerful—perhaps they are cavalry units, rather than infantry—and so they each have an influence of 8. Righty influence propagates in the same way as lefty influence, and now we can see how much influence each side has and recognize areas that are contested (i.e., the areas where both sides have high influence, such as the tiles to the immediate right of the three lefty units), areas that are largely unoccupied (i.e., the areas where neither side has much influence, such as the upper and lower left-hand corners), and areas that are cleanly under the control of one side or the other (i.e., the areas where one side has high influence and the other low, such as the entire right side of the map).

Some authors suggest that when calculating the relative influence between two sides, you simply subtract enemy influence from friendly influence and look at the result—so, for example, the lefty influence in the tile with the middle lefty unit would be 8 (because 13 – 5 = 8), and the righty influence would be –8. This approach loses critical information, however. There is a big difference between a hotly contested tile and one that is far from the action, for example, and yet if you use subtraction to combine the scores, then in both cases you'll end up with an influence of 0.

Another point to consider is that if we keep the influences separately, we can recombine them in any way that we want as alliances change or as we consider different aspects of the situation. For instance, when considering my own attack power in a tile I might include 25% of the influence of my allies, since there is a reasonable chance (but not a certainty) that they will jump into the fight to support me, especially if it's a close thing. This single trick was the only thing that we did to encourage teamwork between AI players that were allies in *Kohan II*, and it resulted in really nice coordinated attacks and/or one-two punches (where one enemy would hit you, and then just as you defeated them another enemy would swoop in to finish you off).

31.4.2 Propagation Calculations

One detail that was glossed over in the aforementioned example is the rate at which influence decays while it propagates. In the example, we simply reduced the influence by 1 for each tile it propagated through—so, for example, the righty units (which were more powerful) exerted their influence over a larger area. This approach uses the formula in the following equation:

$$I_D = I_0 - \left(\text{distance} \cdot k\right) \tag{31.1}$$

where

I_D is the influence at some distance
I_0 is the initial influence that the unit will exert in the tile where it's located
Distance is the distance between those two tiles
k is a tuning constant (which can be adjust to make the AI behave appropriately)

It's worth noting that for tile-based influence, we sometimes use Manhattan distance rather than straight-line distance (i.e., we calculate the distance as the sum of the change in x and the change in y, rather than using the Pythagorean theorem to calculate it). This makes sense if our units only move along the axes, but it can also just be easier to compute. This is the approach that was used in Figure 31.3.

We should also note that influences that are less than 0 should be discarded, but for the sake of brevity we have removed the *max*() statement from this and all of the following formulae.

Although the simple distance-based approach given earlier is appropriate for some applications (such as border calculations, which we discuss in Section 31.4.5), we can do better when reasoning about combat strength. Conceptually, the influence in each tile represents the ability of a player's units to get to a fight in a given tile fast enough to have an impact on the outcome of that fight. The farther away the units are, the less impact they'll have, because a significant amount of fighting will already have occurred by the time that they arrive. Thus, the real limiting factor in how far the influence propagates is not simply distance but rather the travel time to arrive at a distant tile. We can model this with a formula like the one in the following equation:

$$I_D = I_0 \cdot \frac{max\_time - travel\_time}{max\_time} \tag{31.2}$$

where

max_time is the maximum travel time at which a unit is considered to exert influence (e.g., 60 seconds)
travel_time is the actual amount of time it will take to travel to the distant tile

The aforementioned formulae are *linear* and *continuous*, which is to say that if you graph them then you will get a single straight line. There are times when we want nonlinear formulae (the line isn't straight), noncontinuous formulae (there is more than one line), or both. For example, an artillery unit that can fire over several tiles might use a discontinuous formula. For tiles within its range of fire, it will exert full influence. Beyond that distance, its influence will depend on the time to pack and unpack the artillery piece, but the travel time will just be the time to get in range. In order to accomplish this, we will need two formulae. Tiles within firing range simply use $I_D = I_0$, while more distant tiles use a formula such as the one given in the following equation:

$$I_D = I_0 \cdot \frac{max\_time - (time\_to\_pack + time\_to\_get\_in\_range + time\_to\_unpack)}{max\_time} \tag{31.3}$$

Thus, the influence is constant and high out to some distance and then it drops to a significantly smaller value (because beyond that distance, the unit needs to pack and unpack as well as travelling before it can enter the fight).

Similarly, a commander unit might have a nonlinear influence that looks like an inverted parabola—that is, the commander remains strong in the vicinity of its troops (where it is highly effective) but then drops off more and more sharply with distance. We could model this with a formula such is the one in the following equation:

$$I_D = I_0 - (\text{distance})^k \tag{31.4}$$

Finally, the designers may want to be able to determine the influence themselves. If they are mathematically inclined, they may be able to help design these sorts of formulae—but if not, allowing them to give you a simple lookup table with the desired influence at any distance may be a good approach. Tables that are created in a spreadsheet and saved as .csv files can be easy for them to create and are also easy to parse in the code.

Ultimately, much of the intelligence in your influence maps comes from the way you choose to propagate the influence. For more detail on constructing AI logic out of mathematical formulae, much of which is directly applicable here, we highly recommend *Behavioral Mathematics for Game AI* [Mark 09].

31.4.3 Force Estimates over Regions

So far, we've talked about mapping influence over tiles, but it's straightforward to map the influence over regions instead. To do this, we first need to ensure that our propagation formula is based on distance (or travel time). All of the aforementioned formulae meet this requirement. Then, when propagating influence, we work our way through the region graph, just as we did through the tiles. In each region, we calculate the distance to the original region along the region graph (which we can track as we go—we don't have to recompute it each time) and use that value to calculate the local influence for that unit.

The only tricky bit in all of this is that we need to ensure that, as we walk the graph, we find the shortest path to each region. In order to do this, as we're walking the graph, when we're deciding which node to expand next, we always expand the node that has

Listing 31.1. The influence propagation algorithm. The regions variable is an array of region pointers, unit is a pointer to the unit whose influence we're propagating, and startRegion is a pointer to the region where unit is located.

```
for (int i = 0; i < numRegions; ++i)
{
    regions[i]->traversed = false;
}

heap<float, int> openList;
openList.push_back(0, startRegion->id);

while (!openList.empty())
{
    pair<float, int> nextRegionEntry = openList.pop();
    int nextRegionID = nextRegionEntry.second;
    Region* nextRegion = regions[nextRegionID];
    nextRegion->traversed = true;

    float distSoFar = nextRegionEntry.first;
    float influence = CalcInfluence(unit, distSoFar);
    if (influence <= 0)
        continue;

    nextRegion->influence += influence;

    Region* child = nextRegion->GetFirstChild();
    for (; child; child = child->GetNextSibling())
    {
        if (!child->IsPassable(unit) || child->traversed)
            continue;

        float dist = distSoFar + GetDist(nextRegion, child);
        openList.push_back(dist, child->id);
    }
}
```

the shortest total path from the starting region. Listing 31.1 shows pseudocode for this algorithm, and Figure 31.4 shows a new map with the influence for the lefty (white) and righty (black) players.

When calculating influence over regions, rather than over tiles, you lose some of the fine-grained precision. On the other hand, the resulting values are much easier to work with. If I have a settlement, three defensive units, and a fort all in close proximity, it's much easier to reason about how they interact if they're all in one region, or at least are in adjacent regions.

Another advantage of this approach is that even though we lose some precision, the resulting values can actually be more accurate. For example, note that the region with the three lefty units has their full influence—that is, the lefty influence in that region is 15. Compare this to Figure 31.3b, where the highest lefty influence was only 13. An influence of 15 is more accurate, since those units are close enough that they are able to—and are very likely to—support one another. Of course, there are border cases as well—for example, the two righty units in adjacent regions still end up with a maximum influence

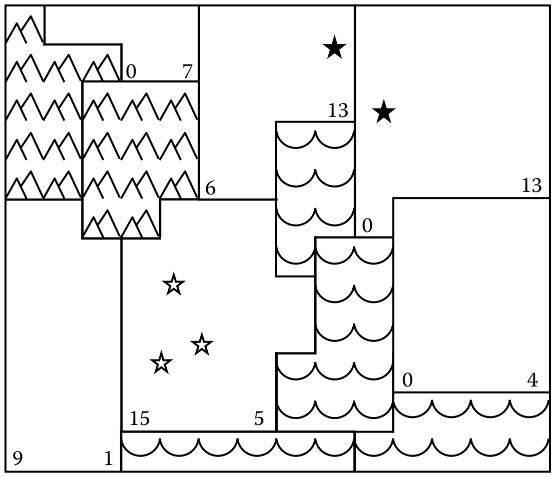

Figure 31.4

Influence propagation over the regions.

of 13, rather than 16. If these cases bother you, it's easy enough to check for them and adjust the influence values accordingly.

31.4.4 Illusion of Intelligence

One possible criticism of influence maps is that they are "cheating," in that they give the AI information that it could not otherwise have. In other words, even though the AI may not know the exact locations of the player's units, it knows generally where they are because it knows about their influence.

One solution is simply not to count influence for units that the AI can't see (or that it hasn't seen recently), but this approach has its own pitfalls. For example, this approach might make it easy to dupe the AI into attacking or defending in places where it shouldn't, which can then be exploited by the player. This will make the AI look stupid and will make the game less fun.

You could argue that the AI should scout better, but building a good scouting algorithm is a hard problem in its own right! Humans intuit how much force they should expect to face based on partial information—if they see two or three enemy units, they can make a guess as to how many they might not have seen and also as to what types of units the enemy is likely to have. Influence maps can provide a nice balance between a truly noncheating AI and an AI that doesn't blatantly cheat but does have a way to compensate for a human's innate intuition and stronger spatial reasoning skills. We don't allow the AI full knowledge, but we allow it enough knowledge to make the sorts of judgments that a human makes naturally.

More to the point, however, it's worth asking ourselves what our true goal is. Is it to build an AI that doesn't cheat or is it to provide a compelling experience for the player? One of the great advantages of basing your attack and defense decisions on influence maps is that it helps to *ensure* that the player's experience will be compelling. It is no fun to battle an enemy who doesn't fight back. It's a huge disappointment to build up your resources, amass an enormous army, and march on the enemy's stronghold—and discover that there are no defenders there, allowing you to win easily. It doesn't matter whether this occurs because the AI is truly stupid or because it made a poor choice and sent its units to the wrong place—it *looks* stupid and will rapidly erode the player's interest in your game.

Because the influence system gives the AI some warning that an attack is approaching—the enemy influence rises—we can avoid this hazard. When the enemy launches their massive attack, the AI will know that it's coming and will be able to ensure that there's something there to meet it. How the AI manages this is of course up to us (the solution used in *Kohan II* can be found in *AI Game Programming Wisdom 3* [Dill 06]), but now we have the information that we need to make a response possible.

Influence maps help to make our attacks appear more intelligent as well. Because the AI will tend to attack where the enemy is weakest, we get an enemy that is "smart" about where it strikes. This forces the player to maintain a broad defense, which adds to the challenge and sense of excitement. Furthermore, if we don't commit all of our forces to an attack immediately, but do require the ones who have actually engaged the enemy to remain where they are, then we get emergent skirmishes and diversionary attacks. The AI will initially attack with just a part of its total force. If the player rushes units from elsewhere to defend against the attack then the next attack may either be held back (because the player now has overwhelming force) or may go in at the new weak spot (which was exposed when the defensive units were pulled away). This result was actually a bit of a surprise for us on the *Kohan II* team—we didn't anticipate it—but it is borne out by the many reviews that said things like "the CPU uses smart battlefield tactics like trying to get the flank and attacking your weak spots" [Abner 04] or that it is "capable of drawing the player away or diverting his attention from a secondary attack." [Butts 04] The AI didn't explicitly reason about these things, they happened emergently as a result of our use of influence maps.

The best of all is that this cheating is hard for players to detect, because it's not blatant, it's not explicit, and it's not perfect knowledge. The AI sometimes does the perfect thing—but sometimes it doesn't. In Kohan II, we allowed players to go back and watch any game from beginning to end, from any viewpoint (i.e., they could turn fog of war off or even watch from the perspective of another player). We never heard complaints of the AI cheating, and indeed, some reviewers described the game as having a "noncheating AI." [Ocampo 04]

It might seem that this will make the AI too hard to beat but honestly, that's a good problem to have. There are a host of ways to tone down a difficult AI, but very few simple ways to increase the challenge when the AI isn't smart enough. What's more, players seem to genuinely enjoy the challenge of beating an AI that maneuvers well. Very nearly without exception, the reviews of *Kohan II* not only mentioned the AI but cited its ability to maneuver intelligently as one of the major strengths of the game.

31.4.5 Border Calculations

So far, we've talked about influence as a way to reason about combat power, but the same techniques can have much broader application. Space being limited, we will give one other example.

It is often beneficial to have a clear sense of what space is inside your borders, what space is clearly under somebody else's control, and what space is contested. This information can have broad-reaching implications. It can affect where you attack, where you position defensive forces, where you choose to expand, how many supporting forces you send to escort noncombat units such as builders or trade caravans, and so forth. On tightly constrained maps it's possible to extract this information directly from the region map [Dill 04], but as we discussed earlier, most strategy games have broader, more loosely constrained maps that don't yield well to this approach.

When a graph-based approach to border calculation is not practical, a less precise but more generalizable approach is to use influence. To do this, simply apply a certain amount of *border influence* to each major structure (e.g., buildings, bases, cities, settlements—whatever is important in your game). When propagating this influence, we want it to spread farther if the initial influence is higher. Thus, we can propagate it using the simple formula given in Equation 31.1.

Figure 31.5 shows an example in which the lefty and righty players each have one settlement. The lefty settlement is fairly small, and so it only has an influence of 15. The righty settlement is a bit larger and has an influence of 25. By examining the influence in each region, we can determine regions that are clearly controlled by the lefties (D and E), regions that are clearly controlled by the righties (C and F), and regions that are contested (A and B). From this, the lefties might identify region B as a good place to position defensive forces (it is contested, is reasonably close to their settlement, and is a chokepoint). Region A is less attractive for defensive forces, since it is a cul-de-sac with nothing in it, but it might be a good position for an ambush force. If the righties were to attack our settlement, forces hidden there could attack from behind and envelop them, blocking their line of retreat. Similarly, the lefties might identify regions D and E as good places to build economic structures, like mines or new settlements, which will need to be defended. Finally, they might pick region C as a good place to build an offensive fort, which will support an attack against the enemy, because it is the one where the lefties have the highest influence.

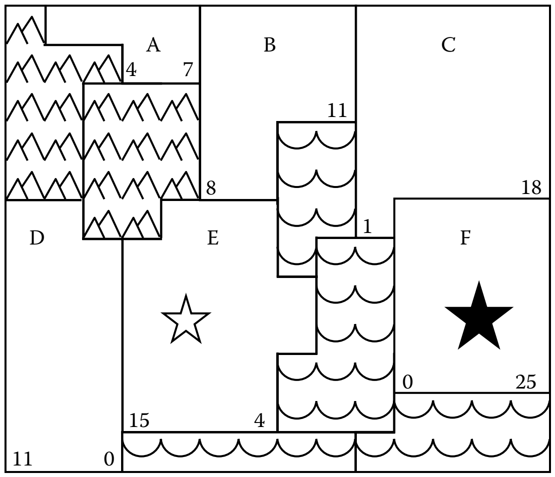

Figure 31.5

An example of how border influence could propagate over regions.

31.5 Spatial Characteristics

Much of the art of making your AI intelligent, whether in terms of spatial reasoning or in any other domain, comes from finding the right way to recognize and capture some distinction that you can then reason about. In this final section we will briefly discuss a number of easy-to-track distinctions that we can use to reason about space.

31.5.1 Scent of Death

One common trick, used in many games, is to keep track of how many things have died in each tile or in each region. We can use this in numerous ways, but two obvious ones are to prefer paths that avoid those regions and to only go into those regions in greater strength. So, for example, if we're considering building a gold mine in a region where we've previously lost a number of units, we might want to send extra defenders along to protect our builder unit (and our new gold mine).

A similar approach can be used to dynamically adjust the strength of enemy units who unexpectedly win battles. If, for example, we lose a battle that we expected to win, then we might increase the influence of the units that defeated us (or even increase the influence of all enemy units) in order to recognize the fact that this player is more wily and tactically competent than we expected. This will help to avoid sending losing attacks against the same target over and over without at least ensuring that each attack is stronger than the last.

31.5.2 High-Traffic Areas

It's often useful to know where other players are *likely* to travel. You can use this information to sneak around them, to ambush them, to place toll posts along their path, and so forth.

One way to gather this information is to simply keep track of the number of units of any given type (military units, caravans, etc.) that have moved through each region. This is guaranteed to be accurate (assuming that you cheat and log units that the AI didn't actually see) but only gives information about what has already happened—it doesn't necessarily predict the future.

Another approach is to plan region paths between all major sites, such as settlements, and keep track of the number of these paths that go through each region. This approach doesn't account for things like the likelihood of travelling between any two sites and can be exploited by players who deliberately choose obscure paths, but it can do a better job of predicting traffic that has not yet occurred.

31.5.3 Avenues of Approach

It is often important to be able to consider not only where the enemy is likely to attack but also *how they are likely to get there*—that is, the likely *avenues of approach*. For example, in real life, when a light infantry platoon is placed in a defensive perimeter, the commander will ensure that somebody is covering every direction, but they will place the heaviest weapons (e.g., the machine guns or grenade launchers) to cover the likely avenues of approach, while placing the lighter weapons (e.g., rifles) to cover the flanks and rear. This isn't foolproof, but it does make it more likely that the enemy will face your heaviest fire when they attack—or else that they'll be forced to work around your sides and attack through more difficult terrain. Likely avenues of approach are also good places to position

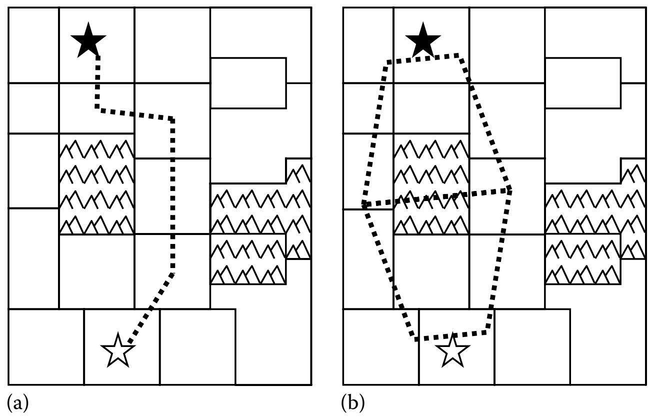

Figure 31.6

The avenues of approach between enemy settlements calculated using the region path (a) or two trapezoids (b).

LP/OPs, screening forces (i.e., defensive forces that will engage the enemy *before* they get to their target), and/or ambush forces (which can be used either to surprise enemies on their way in or to entrap them once they've engaged).

Of course, figuring out where the avenues of approach are is as much art as science, even in the real world. While computers aren't likely to do this as well as humans, there are some simple heuristics that you can use. The most obvious, shown in Figure 31.6a, is simply to calculate the shortest region path to nearby enemy sites or known enemy positions. The problem with this approach is that it doesn't catch alternate routes that might be used if enemy units don't start exactly where expected or deliberately take a longer path, such as the path to the west side of the mountains.

One alternative is to search some number of regions out from the shortest path. For example, if we search two regions out from the path in Figure 31.6a, then we would find the western pass through the mountains—but we would also identify a lot of inappropriate regions as avenues of approach (e.g., the cul-de-sac in the southeast corner of the map). We can add further complexity in order to try to identify and eliminate these regions, but that sort of special-case logic is prone to be brittle and difficult to maintain.

Another alternative, shown in Figure 31.6b, is to superimpose two adjacent trapezoids onto the map such that they stretch between the region you're defending and the nearby enemy position. These trapezoids should be narrower at the origin and destination and wider where they connect. Any region underneath these trapezoids can be considered an avenue of approach. This solution works well in this particular example, but one can easily imagine maps on which the only path goes well to the side of a straight line or even wraps around behind your defensive position, and so the actual avenue of approach is quite different from what the trapezoids suggest.

The best solution might be to combine the two techniques, for example, by only using the trapezoids if they cover every region on the shortest path or by warping the trapezoids to expand to the side of the shortest path, but this is certainly an area in which more work is merited.

It's also worth noting that while these examples have, for simplicity, only shown a single enemy location from which avenues of approach can originate, in reality, there are often multiple sources for enemy forces. In that case you may need to consider the approaches from all of them.

31.5.4 Flanking Attacks

In the infantry one learns numerous techniques for attacking an enemy position. Without question, when it is possible to achieve, the preferred approach is to first have one element of your force fix the enemy in place with suppressive fire (i.e., fire at the enemy position, keeping their heads down and preventing them from moving), while another element maneuvers around to their flank and then cuts across from the side to finish them off.

This tactic is rarely seen in video games and indeed runs a significant risk of making the enemy "too good" (especially since they typically outnumber the player as well). With that said, it certainly could add a significant element of both realism and stress to the game. It is actually not difficult to achieve. Divide the AI force into two groups. One group lays down suppressive fire, pinning the player in place. The other group calculates an area similar to the double trapezoid in Figure 31.6b. They then plan a path to the enemy that excludes the area inside of the trapezoids, which will bring them around the side of the enemy and across their flank. This approach was used by the human AI in *The Last of Us* and is described in more detail in Chapter 34, which discusses that game's AI.

31.5.5 Attackable Obstacles

Many fantasy strategy games have maps that are littered with lairs, lost temples, magic portals, mana springs, and other areas that your units can attack and exploit. At the same time, these games usually also feature enemy players who you need to overcome.

When assigning units to attack a distant target such as an enemy settlement, we don't want our units to get bogged down fighting smaller targets. At the same time, if a unit is going to have to path well out of the way of a spider lair, for example, then it might not be the most appropriate unit to use on that attack—or at the very least, we should probably deal with the spider lair first, so that our line of retreat will be clear.

In *Kohan II*, we solved this problem by not allowing a unit to be assigned to an attack if the region path between it and the target was not clear of enemies. This led to a new problem, however, which was that the player could prevent the AI from attacking a settlement simply by placing smaller, low-value forts in front of it.

We solved this second problem by adding a portion of the priority for attacking the distant target onto the intervening target any time that we found an intervening target that blocked one of our units from joining in an attack. As an example of this solution, consider Figure 31.7. The lefties have a priority of 100 to attack the righty settlement located in region F with their infantry units in region E, but there is a lost temple located in region C that will block their advance. As a result, we don't allow the AI to use those units in an attack on the settlement, but we do transfer 25% of the priority to attack the settlement to a goal to attack the lost temple. There was already a priority of 10 to attack the lost temple, so that priority is now increased to 35.

The end result is that if there is a good place to attack a high-priority target, such as an enemy settlement, then the AI will find it and perform the attack. If the high-priority

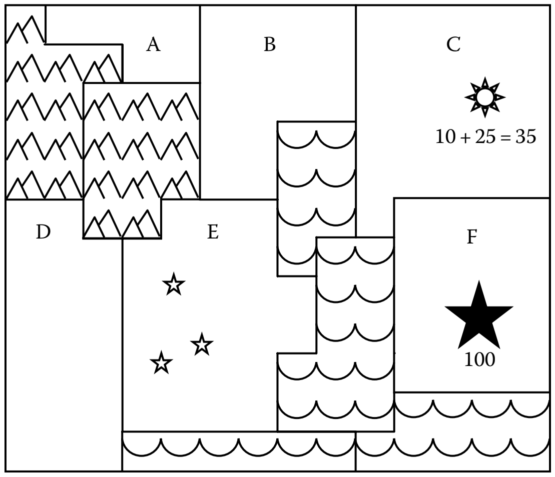

Figure 31.7

The lefties want to attack the enemy settlement in region F, but can't because of the lost temple in region C.

targets are all screened, then the AI will transfer the attack priority for hitting those targets to the screening obstacles and they will be attacked instead.

31.5.6 Counterattacks and Consolidation

In the infantry, one is trained that whenever you take an enemy position, the first thing you do is get into defensive positions and prepare for a counterattack. Only after some time has passed without counterattack do you move on to other tasks.

In strategy games, this same concept can serve us well. We can expect players to counterattack when we take territory away from them, if only because they may have defensive units that are late to arrive, and we can also enhance the player's experience by counterattacking when they take territory away from us. In the words of one reviewer, "[In] most RTS single player games… your armies run into predetermined clumps of foes and they duke it out. Once subdued, you can rest assured that the region is secured and pacified. Not in Kohan II. Fight over a town and take it; you'll find yourself beating a hasty retreat as a counter attack will soon follow unless you can consolidate your advance and secure your line of supply" [WorthPlaying 04].

Accomplishing this is relatively simple. Any time the enemy captures one of our settlements, we increase the priority for attacking the region that the settlement was in for the next several minutes. If we have enough remaining units to launch an attack (which we determine using the enemy influence in the region), then a counterattack will result. If we don't have the strength for a counterattack, as is often the case, then we'll preserve our forces for other goals—but if the enemy influence should drop too quickly (i.e., if the player pulls his units out of the region right after capturing it) then the counterattack goal will still be there for consideration.

We can brace against player counterattacks and follow-up attacks in a similar way. Any time that we win a fight against the enemy (whether offensive or defensive), we boost the priority for defending in the corresponding region. The influence map will tell us how much strength we need to bring to that defense, but boosting the priority lets the AI know that defending that specific area is particularly important because of the increased likelihood of further attack.

31.6 Conclusion and Future Thoughts

Spatial reasoning is a vast topic and this chapter has done little more than scratch the surface. We have concentrated most heavily on strategy game AI and particularly on techniques drawn from the RTS *Kohan II*, although ideas for other types of games are sprinkled throughout. For example, small unit tactics (such as those found in most first-person shooters) rely heavily on concepts such as cover and concealment, fire and maneuver, aggressive action, suppressive fire, lanes of fire, high ground, restricted terrain, kill zones, and so forth—and we have at best handwaved at those. Although true military tactics may be overkill for most games, they are a good place to begin—and they are full of spatial reasoning problems. The U.S. Army's FM 3-21.8, *The Infantry Rifle Platoon and Squad* [US Army 07], is publically available and is an excellent starting point for learning about how real-world forces operate.

Of course, many games are neither strategy games nor first-person shooters. Spatial reasoning solutions are as varied as the (often game-specific) problems that require them and the types of player experience that the game designers want to create. As with any aspect of game AI, it is best to start by thinking over the problem space. What would a human do? How would the human decide, and how can those decisions be simulated in code? What would those decisions look like to the enemy? What portion of that would make the game more compelling for our players and what would make it less? What would be fun?

In this chapter, we discussed some of the tools that can be used to help answer those questions, including techniques for creating a spatial partition, characteristics of a good spatial partition, a number of example uses of that spatial partition, influence maps and example uses of those, and a smattering of additional spatial characteristics that can be used to drive AI decisions. Although some of these concepts are directly applicable to broad classes of games, most require modification and customization to be truly useful. Our hope here is not to have provided a complete solution that can be dropped into any new game, but rather to have given a few ideas, starting points, hints, and most of all to have inspired you, the reader, to take space into account in more meaningful and compelling ways in your next game.

References

[Abner 04] Abner, W. 2004. Reviews: Kohan II: Kings of War. GameSpy.com. http://pc.gamespy.com/pc/kohan-ii-kings-of-war/549000p2.html (accessed May 18, 2014).

[Butts 04] Butts, S. 2004. Kohan II: Kings of War. IGN.com. http://www.ign.com/articles/2004/09/21/kohan-ii-kings-of-war?page=2 (accessed May 18, 2014).

[Champandard 11] Champandard, A.J. 2011. The mechanics of influence mapping: Representation, algorithm and parameters. http://aigamedev.com/open/tutorial/influence-map-mechanics/ (accessed May 18, 2014).

[de Berg 08] de Berg, M., O. Cheong, M. van Kreveld, and M. Overmars. 2008. *Computational Geometry: Algorithms and Applications*, 3rd edn. New York: Springer-Verlag.

[Dill 04] Dill, K. 2004. Performing qualitative terrain analysis in master of orion 3. In *AI Game Programming Wisdom 2*, ed. S. Rabin, pp. 391–398. Hingham, MA: Charles River Media.

[Dill 06] Dill, K. 2006. Prioritizing actions in a goal-based RTS AI. In *AI Game Programming Wisdom 3*, ed. S. Rabin, pp. 321–330. Boston, MA: Charles River Media.

[Floyd 62] Floyd, R.W. 1962. Algorithm 97. *Communications of the ACM* 5–6, 345.

[Forbus 02] Forbus, K.D., J.V. Mahoney, and K. Dill. 2002. How quality spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems* 17(4), 25–30. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.1100&rep=rep1&type=pdf (accessed May 18, 2014).

[Hansson 10] Hansson, N. 2010. Influence maps I. http://gameschoolgems.blogspot.com/2009/12/influence-maps-i.html (accessed May 18, 2014).

[Jurney 07] Jurney, C. and S. Hubick. 2007. Dealing with Destruction: AI from the trenches of COMPANY OF HEROES. In *Game Developer's Conference,* San Francisco, CA. http://www.gdcvault.com/play/765/Dealing-with-Destruction-AI-From (accessed May 18, 2014).

[Kohan II] Kohan II: Kings of war. [PC]. TimeGate Studios, 2004.

[Koster 04] Koster, R. 2004. *A Theory of Fun for Game Design*. Phoenix, AZ: Paraglyph Press.

[Mark 09] Mark, D. 2009. *Behavioral Mathematics for Game AI*. Boston, MA: Cengage Learning.

[Obelleiro 08] Obelleiro, J., R. Sampedro, and D. H. Cerpa. 2008. RTS terrain analysis: An image-processing approach. In *AI Game Programming Wisdom 4*, ed. S. Rabin, pp. 361–372. Boston, MA: Course Technology.

[Ocampo 04] Ocampo, J. 2004. Kohan II: Kings of war updated hands-on impressions. GameSpot.com. http://www.gamespot.com/articles/kohan-ii-kings-of-war-updated-hands-on-impressions/1100-6104697/ (accessed May 18, 2014).

[Tozour 01] Tozour, P. 2001. Influence mapping. In *Game Programming Gems 2*, ed. M. DeLoura. Hingham, MA: Charles River Media.

[Tozour 02] Tozour, P. 2002. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 171–185. Hingham, MA: Charles River Media.

[US Army 07] US Army. 2007. *FM 3-21.8: The Infantry Rifle Platoon and Squad*. Washington, DC: Department of the Army. http://armypubs.army.mil/doctrine/DR_pubs/dr_a/pdf/fm3_21×8.pdf (accessed September 7, 2014).

[WorthPlaying 04] WorthPlaying.com. 2004. Kohan II: Kings of War. http://worthplaying.com/article/2004/11/26/reviews/20850/ (accessed May 18, 2014).