Flooding the Influence Map for Chase in *Dishonored 2*

Laurent Couvidou

1 Introduction

Some scientists speculate that as animals we developed brains for the singular purpose of controlling movement (Wolpert 2011). Thus, when it comes to simulating organic movement for non-player characters in a video game, how and where they move is fundamental to a good experience.

Effectively reasoning about NPC movement, however, is complex and usually involves a fair amount of *spatial reasoning*, the process of modeling a space to draw conclusions about how to act within it. More specifically, *influence mapping* is a common technique for reasoning about a space (Rabin 2004). At its core, an influence map is a discrete set of cells, each having a position and shape in space as well as connections to neighboring cells (either explicit or implicit). Methods used for discretization vary, from a 2D or 3D grids to various forms of graphs or even using point-based influence for infinite resolution (Lewis 2015).

The canonical algorithm focuses on using an influence map to describe the amount of control a player has over an area given the position and type of units on the board, in other words the *current* game state (Hodson 1995). Subsequent work has explored how this idea can be adapted to guess where a player or enemy might go, or the *prospective* game state (Isla 2009, Champandard 2011, Isla 2013). Building on that work, this chapter delves into a simple algorithm that leverages an influence map to explore that possibility space and guess where the player may have gone when pursued by an NPC.

2 Where Is The Player?

For context, the combat AI of *Dishonored 2* supports many different states. For the sake of this chapter we'll consider only two of them:

- *Engage*: NPCs with direct perception of their target that try to attack them.
- *Chase*: NPCs that have lost their target and run after them.

This discussion will focus on how we initiate the *chase* state. From the player's perspective, a "good chase" is one in which the NPC appears to be operating on a reasonable belief of where the player may be and ceases pursuit at an appropriate time. A pursuing NPC therefore needs to determine an appropriate point to run to with the hope that the player is found along the way, triggering a re-engagement. If on reaching this point the NPC fails to find the player, they will give up combat with an explicit transition (looking around and barking their disappointment) and switch to a slow-paced search behavior where they explore the level thoroughly. *Dishonored 2* also features levels with a lot of verticality and powers that allow and even encourage players to quickly escape (e.g. *Blink, Far Reach*), which we have to keep in mind when determining what is sensible for NPC behavior.

When pursuing a player, there are many ways to find an appropriate destination. One approach is to have the fleeing players place *breadcrumbs* along their path that NPCs then seek and consume. This was used in the original *Dishonored* and gave reasonably

convincing results. However, this provides NPCs with essentially omniscience regarding players' whereabouts, which can feel unfair and frustrating to the player when it is noticed.

For *Dishonored 2*, we switched to a different engine and had to rewrite most of the AI from scratch, giving us an opportunity to try another approach. As a temporary solution, we first implemented a straightforward (literally) form of dead reckoning. NPCs would perform a linear extrapolation using the last-known target position and direction, with a navmesh line-of-sight check to stay within navigable space.

Surprisingly, this version survived a long time in production and almost made it to shipping. At some point, though, play testers reported that the chase experience was disappointing. In many cases, the extrapolation was hitting nearby navmesh boundaries (e.g. a wall or a cliff) causing NPCs to give up too early when there was still an obvious path to follow, as shown in Figure 1.

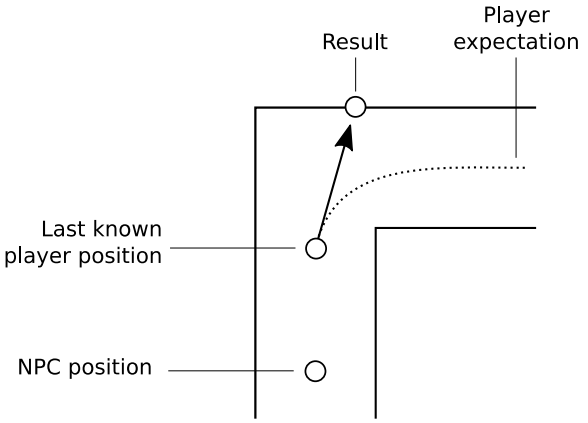

Figure 1 NPCs often fail to maintain chase long enough when using dead reckoning.

Our conclusion was that both breadcrumbing and dead reckoning were unsatisfying. We wanted something that would not feel like the AI was cheating yet remain robust given our intricate level topologies. Thus, we decided to use an influence map.

3 Algorithm

Dishonored 2 uses a sparse 3D matrix generated from the navmesh as a canvas for influence mapping that we dubbed the Dynamic Space Manager (DSM). It is not a traditional three-dimensional array, but a collection of one-dimensional arrays mapped to world coordinates with a dictionary of keys for better memory performance (Sadoulet 2017).

We used this DSM to develop an influence propagation algorithm for an NPC chasing a player that tries to get away during combat. Since this is fast-paced gameplay we need to quickly commit to a single destination. So, we ran a fixed number of propagation steps per frame to make the resolution fast enough, without having a performance hit on a single frame.

Also note that for the sake of this chapter, we will limit our examples to a 2D grid; adding a dimension is left as an exercise to the reader.

Let us assume a cell data structure composed of the following:

- A position.

- A list of neighboring cells.
- An integer "temperature" value, representing the likelihood of finding the player there.

The algorithm then proceeds as follows:

Step A. Once the player's location is lost by an NPC, we seed the map using their last known position and direction. The twist compared to other approaches is that we perform two kinds of seeding (Figure 2a):

1. *Heat generation*: We first *heat* up the cell under the last known player *position*, by setting their value to an arbitrary *hot value*, **H** (9 in the example).
2. *Barrier generation*: We also create a *barrier* seed with all the neighbors of the heated-up cell that are in the opposite half-plane compared to the last-known player *direction* (delimited by the dotted diagonal in the example). Our implementation sets the cell value to -1 to signal that it is part of the barrier.

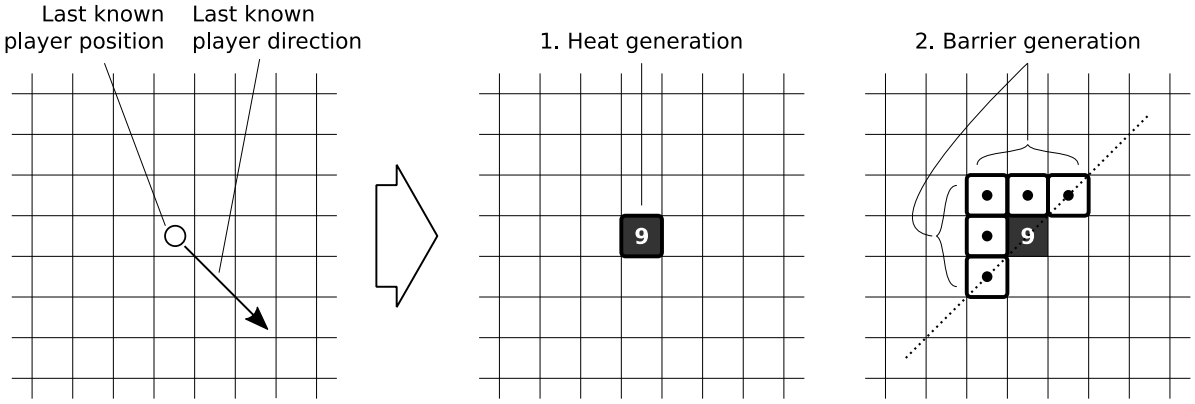

Figure 2a Seeding the influence map.

Step B. The algorithm then runs a finite number of influence propagation steps. Just like we have two kinds of seeding, we also have two phases of propagation (Figure 2b):

1. *Heat propagation*: In the first phase, the heat now propagates from all the cells heated up by the previous step. Neighbors in all directions that were not already touched (either already warm, or part of the barrier) are set to the same, fixed hot value. Cells that are untouched this round but were already warm now cool down, decrementing their value by one after each step until they reach zero. The result is effectively a heat wave that progressively floods the map.
2. *Barrier propagation*: In the second phase, the barrier now propagates based on all the cells added to it by the previous step, in the same half-plane that was described above (in local space compared to the source barrier cell). This prevents heat from flooding backwards compared to the prospective player movement; we want NPCs to reason that their target keeps its momentum for a while and chase them accordingly, otherwise they could turn their back and run in the opposite direction.

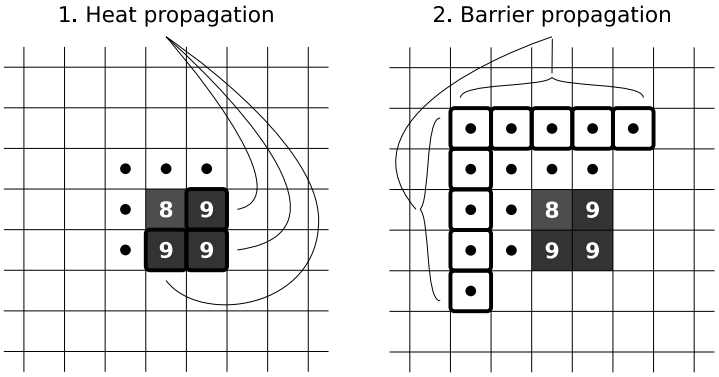

Figure 2b Booting up the propagation after seeding.

We can see the evolution of the algorithm after a few more steps in Figure 3, with the benefit of some surrounding walls (thick lines) to show how it impacts propagation. This is a simple example, but it should give you a grasp of how the flood can behave in a real game level.

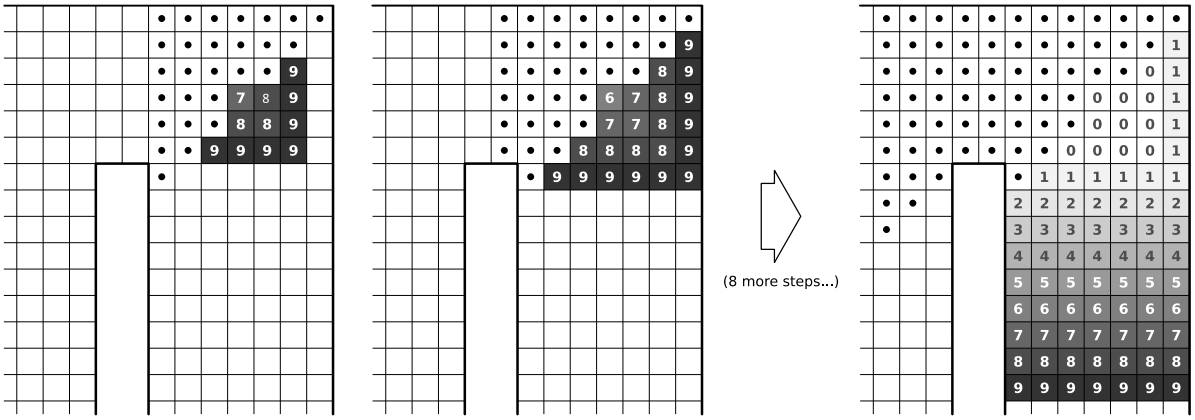

Figure 3 More propagation steps flood the influence map.

Without stop conditions, this algorithm can potentially run for a very long time and flood large parts of the map, depending on the level topology. For that reason, we stop the propagation when one of the following conditions is satisfied:

- There are no further cells to heat up.
- There are *too many* cells heating up at the same step (more than $\mathbf{M_H}$).
- The maximum number of propagation steps has been reached ($\mathbf{M_S}$).

The resulting chase destination is then determined by computing the mean position of the warm cells in the map (their centroid), weighted by their values. If that point is located outside the warm cells, we pick the nearest one. This gives the NPC a goal to chase that follows the level topology instead of hitting the nearest wall, unlike the basic dead reckoning described previously. We compare the two approaches in Figure 4.

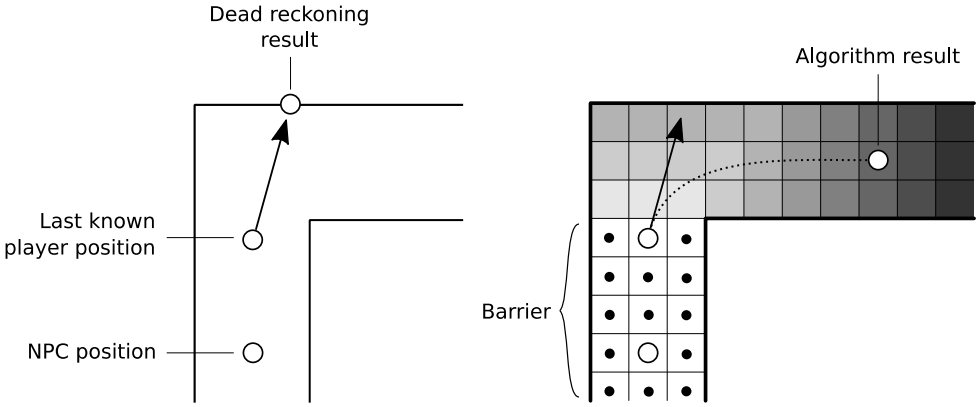

Figure 4 Flooding the influence map allows to follow the level topology.

4 Results

This approach has many nice properties. First, as shown in Figure 5, it naturally leads NPCs to the areas that are the most suited for a player's escape. On the left-hand side, the player might have run in two different directions as both ways are equally plausible and the algorithm result is right in the middle. Thus, the NPC reasonably gives up combat since there is no way to make a better decision and switches to a slow-paced search behavior. On the right-hand example, however, one path is short corridor with a nearby dead end and the other path is a long corridor that offers more space for exploration. Here our algorithm will correctly guess that the longest corridor is the most plausible player escape route.

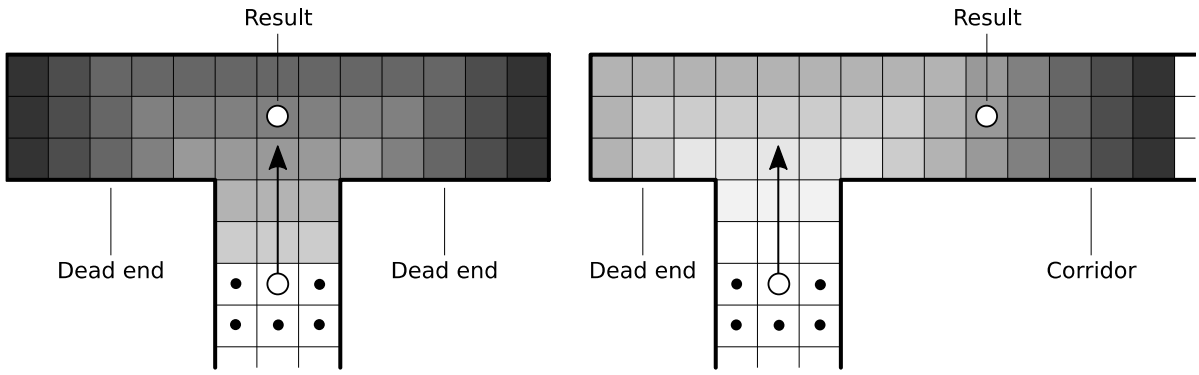

Figure 5 The result tilts towards the player's most likely position.

Another nice property is that the algorithm stops quickly when it encounters a wide area (due to the maximum number of heated up cells per step, $\mathbf{M_H}$), but runs longer in narrow corridors (as long as the maximum propagation step count, $\mathbf{M_S}$, is not reached), as shown in Figure 6. This again makes sense in the context of a chase. A wide area means that the player can hide anywhere, and it is better for NPCs to switch to slow-paced search. Whereas in a narrow corridor, it is better to keep running for a while since the player is most likely to be found at the other end.

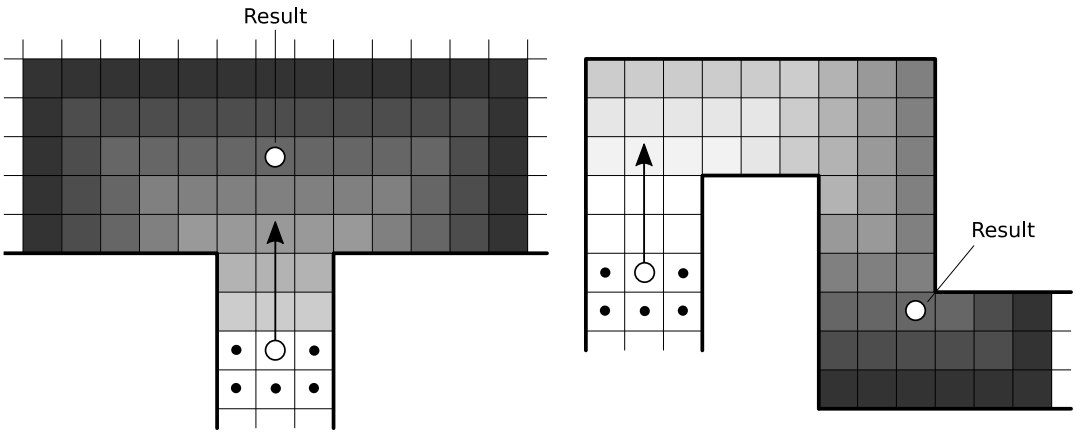

Figure 6 The chase stops early in wide areas but keeps going in long corridors.

These are only 2D examples, but the algorithm performs well in 3D spaces, too. When cells have connections up and down (staircases, traversals, slopes), the flood has more options to expand vertically but the same principles still apply.

Finally, the algorithm can be easily tweaked with only a handful of constants, to make the chase more or less forgiving, as shown in Table 1.

Table 1 Tweakable constants

Constant	Effect	Value in *Dishonored 2*
Hot value (**H**)	Sets how prone NPCs are to explore long corridors. The lower the value, the faster the cells in a dead end cool down to zero, so the more the result is skewed towards corridors.	20
Maximum heated cells per step ($\mathbf{M_H}$)	Drives how easily the propagation stops in wide spaces and therefore how aggressively the NPCs will venture into wider spaces.	25
Maximum steps ($\mathbf{M_S}$)	Drives how far the propagation can run. This value is the one that strictly limits how far NPCs will run when they chase a player, so it was used as a difficulty level variable in *Dishonored 2*.	Level 1: 20 Level 2: 30 Level 3: 40

5 Caveats

One downside of this approach is that it does not adapt to new information. Believable NPCs running towards the given destination should not ignore new stimuli leading in another direction, such as gun sounds or explosions. In *Dishonored 2*, we reset and re-run the algorithm when a stimulus of high enough priority is received. The exception is direct sight or contact, which brings the NPC back to engage mode. While theoretically NPCs could chase forever if they continue to receive new information, in practice this was not a problem we observed. If you were concerned about this in your game, you might consider

ways to prevent the algorithm from resetting indefinitely, for example, by accepting only a finite number of resets.

Another downside is that although the barrier is designed to prevent propagation "leaks" (which can put the result directly behind an NPC, making them turn around and run in the opposite direction compared to the last-known player position), it also prevents them from exploring potentially interesting spaces, such as the exit visible on the right in Figure 7.

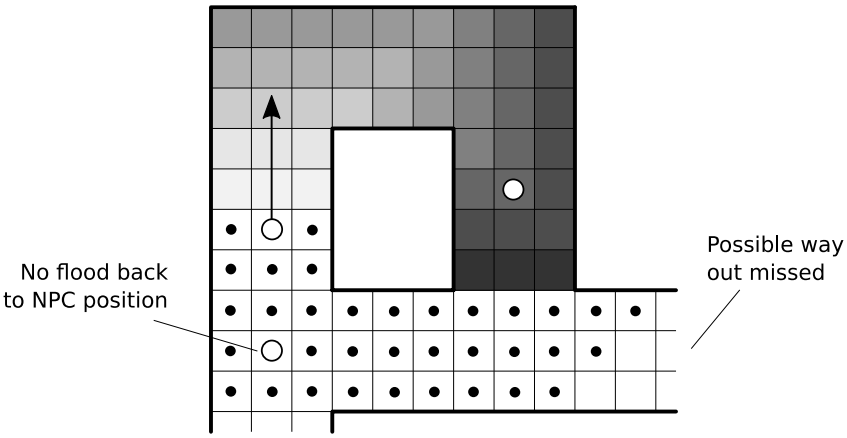

Figure 7 The barrier sometimes prevents NPCs from exploring an area.

Though we considered a few different workarounds for this issue, ultimately, we were unable to find anything satisfactory that justified the additional complexity. Every attempt at constraining the barrier (maximum distance, smaller angle, bounding shape, et cetera) forced us to set an arbitrary limit that always reintroduced some unwanted leaks in one level topology or another. Moreover, a 3D space lessens the problem since the heat flood has more options to explore, and thus is less likely to be artificially blocked by the barrier.

6 Possible Improvements

One variation that we explored was running the algorithm over time by propagating the information using player run speed. We found, however, that this caused problems for the locomotion animation. Oftentimes the updates would cause abrupt destination changes as the flood updated, causing the NPC to turn frequently and unnaturally after repathing to the new destination. Thus, in our case we fell back to running the full algorithm over just a few frames. Nonetheless, this could give interesting results for games where abrupt changes in direction are reasonable and do not appear too erratic with the chosen animation style.

Additionally, the algorithm does not prevent an NPC from deciding to stop chasing right before a corner, which could feel unnatural. It also favors large areas over small ones, as the centroid calculation will accumulate more weight in wide areas. Logically, you could argue that a narrow corridor makes for an equivalent if not better escape route than a wide one. Both points might be solved by introducing more detailed spatial reasoning. In our case, they were not reported as issues during playtests, so we didn't investigate them any further.

Finally, one could spice up this simple flooding algorithm with the addition of more traditional strategic analysis performed on the influence map, such as seeking cover or

resources, changing what locations are desirable or logical (Tozour 2001). For example, a dead end may nonetheless have great cover selection for a player seeking to remain out of sight.

7 Conclusion

Our algorithm was intentionally kept simple and lightweight in terms of computing power requirements. With careful memory handling we ran without trouble on the target hardware (PC, PS3, Xbox One) for *Dishonored 2*. Moreover, its outcomes are predictable and proved to be very robust in all kinds of level topology, no matter the intricacy. Given the complexity of believable movement having comprehensible tools such as this was very valuable for our team.

Finally, the results are surprisingly "human" and enjoyable; NPCs continue to chase or give up when it makes sense. Players also intuitively understand this behavior, leading to some emergent gameplay, such as tricking enemies. We found players learned to place traps in the path of the NPC and then hide, much as you might lure a human pursuer. Unsuspecting NPCs would then run past their target and to their demise, allowing for some nice and memorable "ha-ha" moments.

All in all, we feel that this is a robust and flexible approach for reasoning about NPC movements that would be useful in many games.

8 References

Wolpert, D. 2011. The real reason for brains. TED talk, https://www.ted.com/talks/daniel_wolpert_the_real_reason_for_brains?language=en.

Rabin, S. 2004. Common Game AI Techniques. In *AI Game Programming Wisdom 2*, ed. S. Rabin, 3-14. Charles River Media.

Lewis, M. 2015. Escaping the Grid: Infinite-Resolution Influence Mapping. In *Game AI Pro 2*, ed. S. Rabin, 327-342. CRC Press.

Hodsdon, S. et al. 1995. The "Influence Mapping" Thread. Usenet thread. https://web.archive.org/web/20090203043159/http://gameai.com:80/influ.thread.html.

Isla, Damian et al. 2009. Beyond Behavior: An Introduction to Knowledge Representation. *GDC 2009*, San Francisco, https://www.gdcvault.com/play/1267/(307)-Beyond-Behavior-An-Introduction

Champandard, A.J. 2011. The Mechanics of Influence Mapping: Representation, Algorithm & Parameters. *AIGameDev.com* https://web.archive.org/web/20190717210940/http://aigamedev.com/open/tutorial/influence-map-mechanics/.

Isla, Damian et al. 2013. From the Behavior Up: When the AI Is the Design. *GDC 2013*, San Francisco https://www.gdcvault.com/play/1018057/From-the-Behavior-Up-When (30:30~).

Sadoulet, X., and Couvidou, L. 2017. Taking Back What's Ours: The AI of 'Dishonored 2'. *GDC 2017*, San Francisco. https://www.gdcvault.com/play/1024159/Taking-Back-What-s-Ours/.

Tozour, P. 2001. Influence mapping. In Game Programming Gems 2, ed. M. DeLoura, 287–297. Charles River Media.