Knowledge is Power, an Overview of AI Knowledge Representation in Games

Daniel Brewer

1 Introduction

Artificial Intelligence systems process data to make decisions. The algorithms used to make these decisions are often the focus of intense discussion. However, the behavior of an agent is far more dependent on the data provided to the agent, than on the algorithm used to make its decisions. It is therefore important to analyze the requirements of the design and provide the agents with the required information to perform the role the designer intends [Carlisle 13]. This article will present an overview of commonly used data representations in games and illustrate how to improve agent behavior by providing the right information.

2 Categories of Knowledge

Every game has different requirements for its AI systems. The data to make decisions can be split into three broad categories:

- Static Environment Data
- Dynamic Spatial Data
- Entity Specific Data

Static Environment Data, as the name suggests, involves representing the game world itself to the AI systems. This is data that does not change at runtime, or changes very rarely. It represents the physical structure of the level and is what our agents use to find their way around, or select positions to move towards.

Dynamic Spatial Data is used for representing spatial relationships that are affected by the environment and change over time. Typically this includes territory and influence over the world that shift as the game progresses.

Any data used to make decisions about a particular ‘thing’ in the game falls under Entity Specific Data. It is potentially the broadest category and is the most game-play specific. This information will vary dramatically based on the requirements of the design.

3 Static Environment Data

Static Environment Data represents the structure of the game world. The most common form is navigation data for path finding. In an action or strategy game, it is important to represent tactically relevant information about the level, such as the region connectivity and chokepoints, visibility approximation, and areas of operation for agents. This data also represents physical objects in the world that the agents are required to interact with, such as doors, vending machines, switches, etc.

3.1 Navigation

Navigation and path finding is one of Game AI’s eternal problem spaces. As environments and the movement requirements of agents grow in complexity, so too does the representation of the navigable space. Agents are no longer simply required to find the shortest path, but instead should select an appropriate path for their behavior. For example, a rogue character might prefer to remain in the shadows and move across soft carpets instead of stone floors where his footsteps might be heard.

Finding a route through a level is traditionally handled by a search algorithm, such as A*. These algorithms treat the world as a connected graph. The nodes of the graph represent locations in the game world. Traversable routes between nodes are represented by links connecting the nodes in the graph. The more features you need the agents to respond to, the more detailed and complex the graph. A more complex graph will use more memory and the longer the search algorithm might take to find the desired route. The trick is to represent the least amount of detail required for the agents to behave the way the designer wants.

There are many possible navigation representations [Tozour 04]. At the basic level, you need to represent the valid, navigable areas and which places are blocked by obstacles. If an agent requires extra information, such as illumination levels or surface type for our rogue above, this data needs to be annotated in the navigation data. For our rogue character in the example above, we would need to mark the surface type of each node - which are carpet and which are stone. We would also need to annotate the illumination level of each node in order to identify which nodes are in shadow and which are in the light.

Regular grids are a popular and well-studied representation. The resolution of the grid is uniform. You are either representing a big large open area with many tiny cells, or you’re grouping everything in a large cell together and may miss obstacles or block off

narrow passages, as Figure 1 illustrates. Any region annotations are per grid cell. It is important to select a grid resolution detailed enough to represent the different regions.

Navigation meshes (NavMeshes) are very popular for continuous worlds. They represent the simplified walkable surface. The advantage is that they adapt to the underlying world. Large open areas can be represented by few large polygons, whereas tight, dense areas will be represented by many smaller polygons. The edges of the polygons do not need to be axis aligned either, which allows them to conform to the floor better around irregular obstacles. Region annotations are per polygon. However, since we are not restricted to a uniform polygon size, we can tessellate the polygons to add more detail as required.

It is possible to store the path finding cost for different regions directly in the navigation data. This is too restrictive though, as it means all agents will use the same costs. We may want the palace guards to not care about the illumination and floor surface, while the rogue does care about this data. It is better to use flags or tags for annotation and then allow the individual agents to have different heuristics for interpreting the cost of the regions. This way the world representation doesn't change, but the agents' interpretation of that representation can change and can result in different behavior depending on the situation. The rogue may prefer to stay in the shadows while sneaking around, but once he is found and decides to flee, he can change his heuristic to simply find the fastest route away.

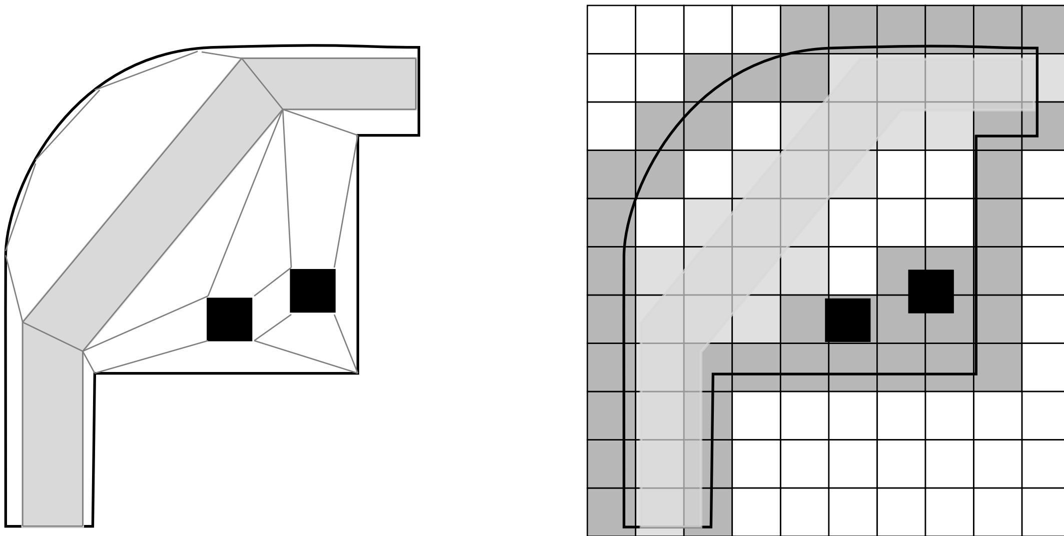

Figure 1 A NavMesh is used to represent the map on the left, while a regular grid is used for the same map on the right. Note that in this case, the grid is too low a resolution to capture all the navigable space behind the pillars in the corner.

3.1.1 Dynamic Navigation Changes

So far we've handled the static, unchanging environment. If the level can change dynamically at runtime, we need a way to update our navigation representation. Some of these changes may be predetermined - e.g. a canal that can be emptied or flooded, or a drawbridge that can be raised or lowered. Sometimes the changes can be completely

dynamic. Perhaps the player can throw a Molotov cocktail and create an area of fire anywhere in the level.

Regular grids have the advantage of being very efficient to modify at runtime. The affected cells can be determined quickly and the appropriate flags changed. Modifying NavMeshes at runtime is trickier. For predetermined changes, it often simpler to demarcate the areas that will change at design time and then toggle the flags appropriately when the world changes at run-time. However, for the completely dynamic changes, you either need to perform cuts and merges on the polygon data to patch in changes, or be able to quickly rebuild the NavMesh. Rebuilding the entire level can be time consuming and wasteful, so a common solution is to build the NavMesh out of many smaller tiles. This is almost a hybrid large regular grid + NavMesh approach. Each tile might be 10m x 10m or 20m x 20m and then if any dynamic changes are required, only the tiles containing changes need to be rebuilt and patched into the NavMesh structure.

3.1.2 Other Representations

Waypoint graphs are a nice, efficient representation for sparse navigation data. If you are working on an interstellar space game with fixed hyperspace routes between systems, then waypoint graphs are your best representation. Each system would be a node and each link would be a hyperspace route between systems.

Waypoint graphs are also useful for a high level representation of the map. In a competitive arena shooter, each room of the map can be represented by a node and the corridors and doorways between rooms can be represented as links between nodes. In a grand strategy game, countries can be represented as nodes and the borders and travel routes are the links. Corridor maps and road networks are other commonly used representations and are effectively waypoint graphs where the widths of the connections are annotated in the representation.

Grids and NavMeshes are typically used for agents that have to move along the ground. Agents that can move freely in 3D, face new representation challenges. 3D regular grids are simple, but extremely wasteful and dense. Waypoint graphs get complicated when the obstacle geometry is dense.

If your agents are restricted to flying over terrain, then height-fields can be a good representation. You are effectively using a 2D grid with the height of the tallest obstacle stored in each cell. The agents can then determine whether it is quicker to fly over or around an obstacle [Josemans 17].

However if your agents are required to navigate through geometry, such as flying through the wreckage of a destroyed spacecraft, or if there is no ground surface, then a sparse voxel octree representation may be more appropriate [Brewer 17]. This would have many small nodes around dense obstacle volumes and few large nodes around open spaces.

3.2 Off-nav links

As game environments get richer, more detailed and more realistic, we expect our game AI agents to traverse these environments in interesting ways. Vaulting a short wall or railing, jumping across a narrow gap, clambering up a wall onto ledge, and sliding down a zip-line are all commonplace actions we expect agents to perform. We need to have some way to

represent these actions to our agents' path finding search so they know that they can vault a railing from a walkway to land below instead of taking the long way around via the stairs.

Off-nav links are a common way to represent these actions. These are links between nodes not represented in the native navigation structure. These nodes do not need to be previously connected, nor do their relative positions matter. What matters is that the agents now know they can go from one area to another by using this new link.

Not all agents may be able to use all these links. The path finding search should only follow links that are open to the agent. An agile agent should take advantage of jumps, vaults and zip-lines, while a heavy, lumbering agent should not take these short-cuts and instead take an alternate route. This means for each of these off-nav links, the pathfinder needs to test if the link is suitable for the agent.

If the number of different types links is prolific, or the requirements for the links overly complicated, the off-nav link can provide an evaluation function to return whether the link is suitable or not, or to provide an extra cost value for using this link. However this is not the most performant method. The types of links can often be reduced to a reasonable set of core capabilities, in which case a simple bit-mask can be used to represent the requirements for the link and the capabilities of the agent. Comparing these bitmasks is a very efficient way to filter out invalid links.

Off-nav links are often represented by smart objects. This allows all the complexity of performing the action to be encapsulated in the smart-object and essentially hidden from the path finding search. The path finding code doesn't need to know how to use a ladder; it just needs a quick way to test if the agent can use ladders, and to know that a ladder provides a route between the ground and the roof.

3.3 Smart Objects

Smart objects are one of the unsung heroes of game AI knowledge representation. At their essence, a smart object is an abstraction that encapsulates some arbitrarily complex game logic behind a simple interface and ties this logic to a particular location or entity in the world. This is a way to embed knowledge in the world, instead of encoding it in an agent. The agent only needs to evaluate and start the smart-object executing and wait until it is complete. The Sims is the poster child for how smart objects can be used to extend the capabilities of agents and make them seem more aware of their environment [Bourse 12].

A simple example from an action game is an alarm switch. An agent who notices the player may have a behavior that checks for useful smart objects. If the alarm has not yet sounded, this search may return a nearby alarm switch smart object. This smart object would be responsible for providing the appropriate destination position and facing for the agent to move towards and once the agent starts executing the smart object, it is the smart object, not the agent's behaviors, that determines the animation to play to press the button and would also start the alarm sounds, flashing lights and start the reinforcement spawning system to summon more agents to the area.

A more complex example might be a ladder. Smart objects are often used to represent off-nav links as they just have to advertise the connection between areas and then handle all the mechanics of traversing the link internally. An agent might path through a ladder smart object to get to a position on the roof. The ladder smart object would have its

own internal state machine that would cause the agent to play a get on ladder animation and then loop a climbing animation while moving the agent up the ladder and finally play a get off animation at the top before completing and allowing the agent to continue to his destination.

The uses of smart objects are prolific: switches, doors, off-nav links, gun-emplacements, treasure chests, cover points, patrol points, queuing at the bar, positions around a campfire, or door-breach points for a swat entrance and so on. Using smart objects requires the AI system to only decide which smart object to use, without needing to know all the details of the object. A lot of complicated decision logic can be encapsulated in a simple, designer-placed smart object.

3.4 Tactical Information

A lot of decisions in action and strategy games revolve around deciding where an agent should move and how he should get there. Agents seem smarter when they show some self-preservation instincts and take up firing positions in cover, or try to break line of sight with their target. It can also be beneficial for agents to have an understanding of the overall flow and structure of the level, such as choke points and good vantage points. It is important for agents to be able to identify these features, which requires we represent them to the agents.

3.4.1 Cover

Most simply, cover can be stored as a collection of discrete cover points. Each location would need to store a position, a direction from which cover is provided and some flags to select the appropriate animations to use. The cover selection algorithm can simply collect the cover points, score them to determine the most useful ones and select the point with the best score. While this provides good information about where to move, it doesn't help provide any information about what is obscured by the cover.

Instead of discrete points, cover can be represented by line segments. An agent can take up position anywhere along this line and be in cover. Each segment needs to provide the direction and height of the cover provided. This extra information is common to all possible points along the segment. For position selection, one can simply generate a number of points along the segment to be evaluated. By connecting segments together into chains, agents have additional information about how they can move while staying in cover or can reason about where a target might move from a cover location.

Another advantage of using line segments for cover is that it becomes possible to use this information to help identify areas of open space that are obscured. Since the cover line segment effectively reduces the collision geometry down to a rectangle, a frustum can be constructed from the silhouette, as shown in Figure 2. This frustum can be used to quickly determine line of fire between positions without performing ray-casts against the collision geometry. It can also be used to weight path finding queries to get agents to take more defensive routes through the environment [Brewer 13].

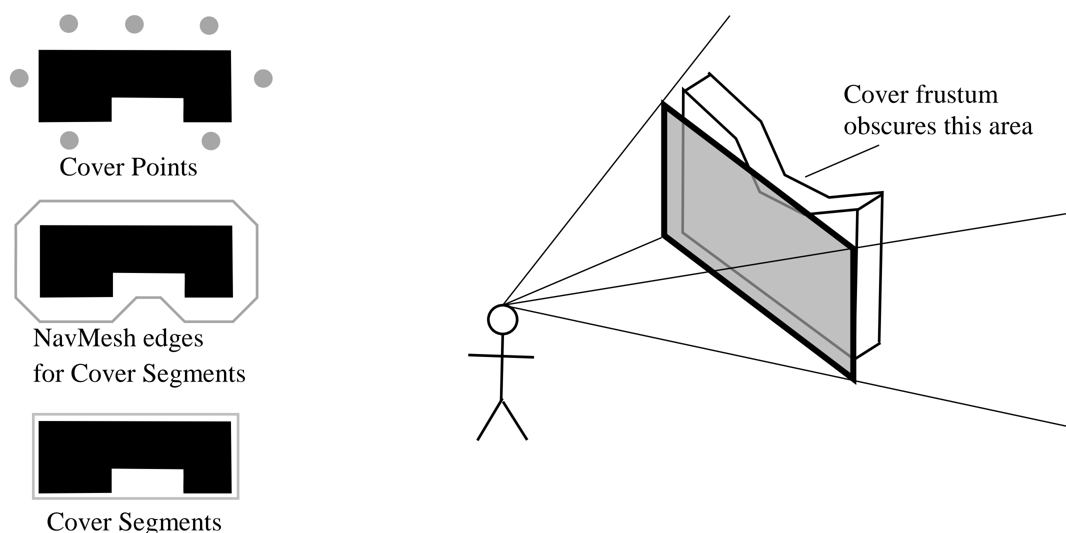

Figure 2 Different types of cover representations are shown on the left. The right shows how a simple cover segment can be used to create a frustum approximating the cover provided by an obstacle.

It is possible to use the edges of the NavMesh directly in place of independent cover segments. All that is required is to add extra annotation to the NavMesh edges to identify whether they are against a wall or obstacle that can provide cover and how tall that obstacle is. This does not require a separate data-structure for the cover and allows the cover query to take advantage of the floor layout to provide cover within a determined movement distance instead of straight line distance. However often the NavMesh will have extra edges to round corners and may have more detailed edges where a single line would be more appropriate.

Cover markup can either be placed manually by designers, or automatically generated. An automatic system could take the edges of the NavMesh as input, and then post-process them to construct simpler cover segments.

3.4.2 Areas of Operation

Designers often want to choreograph how a particular encounter will unfold. Detailed scripting of every agent's actions is not only tedious, but highly error prone as each player is likely to tackle the encounter in a different way. Instead of assigning specific positions for agents, it is often more useful to provide agents with a volume in which to operate. The agent's systemic behaviors and position selection systems are still in control; however, the positions considered are filtered by the area of operation.

This can be handled explicitly with scripts assigning specific agents to specific areas of operation. It can also be controlled indirectly by designers adjusting minimum and maximum agent counts in areas and having agents assigned systemically [Isla 08] [Gallant 17]. Adding connectivity mark-up between areas of operations, simplifies the direction of agents. NPCs that find their area of operation invalidated can automatically follow the link to the next fallback area of operation. This can allow designers to stage complex fighting withdrawals and ambushes, such as the maneuver shown in Figure 3.

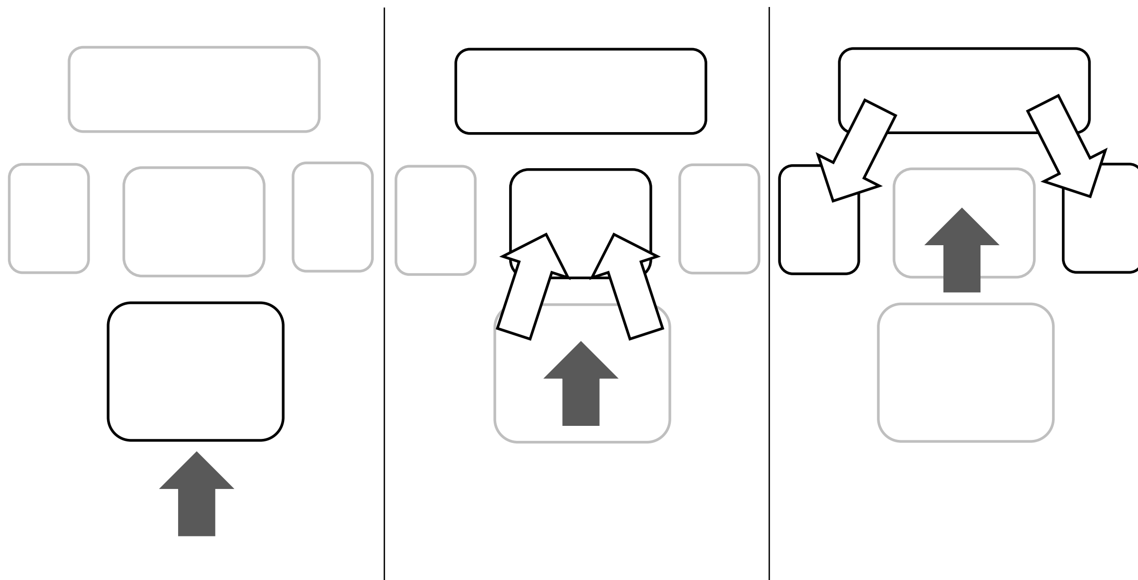

Figure 3 An example encounter setup to retreat ahead of the player, drawing him into an envelopment counter-strike.

3.4.3 Tactical Graph

When making tactical decisions, it is important to reason about the structure and flow of the map. The navigation representation can certainly help, but it is often too detailed. A NavMesh will allow you to identify the free space between some trees, but will not easily be able to tell you about the forest. It is often useful to have a higher level representation of the flow of the map. What are the main regions of the map? How are these regions connected? Where are there choke-points?

A structure similar to a rough corridor map can be useful to represent this information. It should not be a detailed representation of the navigation space however. It can ignore most small obstacles and merge together areas into semantically distinct regions.

If there are only two routes into a region of the map, then those are the areas that need to be defended from attack. If one of those routes is very narrow and restrictive, then fewer defenses will be needed to cover that route, while a wide, open route would require more significant defenses. This is the type of information that can be stored in a tactical map, such as that illustrated in Figure 4.

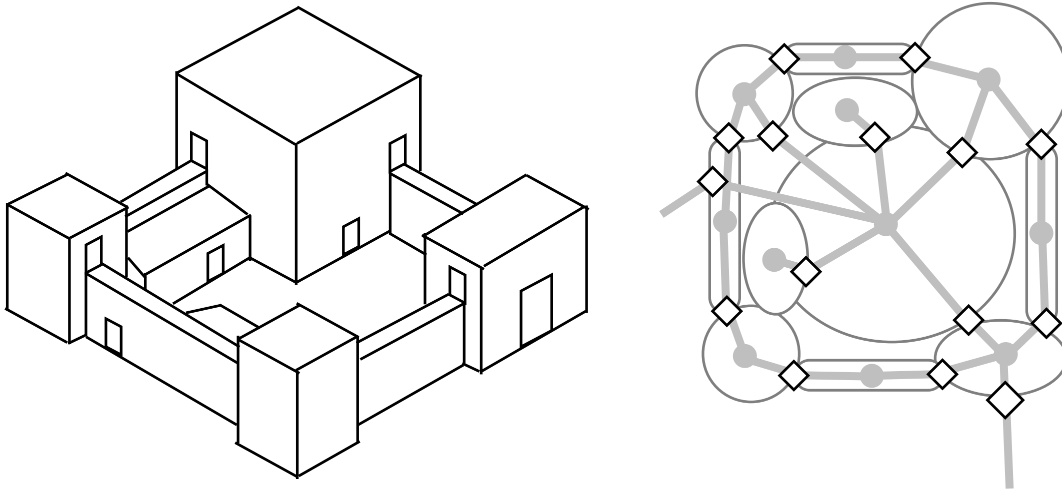

Figure 4 An example tactical map for a small castle. Note that the detailed navigation representation is not required, only the high level structure and flow.

Note that this structure is similar to a high-level in a hierarchical navigation representation. It is useful to represent tactically significant connections between areas. For example there may not be a direct route to move from a courtyard to an overlooking balcony; however, it may be possible to see the one area from the other. This information would be useful for a position selection algorithm. The visibility information is only an approximation. It should not be used to determine if a target can be seen, but only if it might be possible to see a target from a location. The distinction is subtle. This approximation should be used for an early filtering pass when evaluating possible positions.

4 Dynamic Spatial Data

Even though the physical structure of a map may be static, the positioning of units can result in many dynamic situations. It is useful to be able to predict and react to these changing situations. This is where it is useful to have a map of values representing changing information in a way that reacts to and responds to the environment.

Influence maps are often represented as an array of values correlated to a grid or graph [Mark 15]. Values accumulate in cells in a grid, or nodes of a graph, and the values spread out and dissipate through neighboring cells in the grid, or connections in the graph. Since the grid or graph is usually correlated to the structure of the map, these values tend to flow around obstacles and through channels in the map.

It is often simplest to represent an influence map as an array parallel to a navigation representation. This allows you to reuse the connectivity information of the navigation representation and means your influence map is only an array of values, instead of a more complex structure. However the larger and more detailed the navigation representation, the larger the influence map, which can result in significant overhead processing the influence map. It is therefore useful to use a coarser graph structure, such as a tactical graph, instead

of the most detailed navigation representation.

Influence maps are commonly used in strategy games to identify areas where the opponent is strong or where they are weak [Dill 15]. However, there are other uses. If you spread influence from the player's position and consider the difference in influence over time, areas ahead of the player will have rising influence while areas behind them will have falling influence. Figure 5 shows how this can be used to predict where the player is heading.

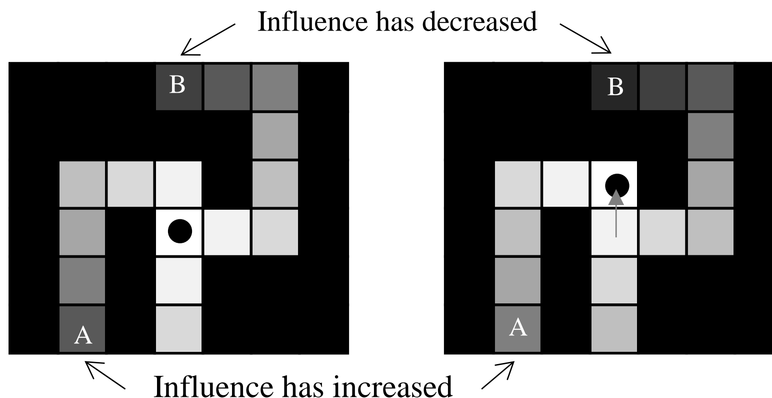

Figure 5 When the unit moves north, the influence at B is actually decreasing due to the structure of the map, while influence at A is increasing. The unit is therefore moving towards A and away from B.

Adding influence to areas that include agents, but not spreading this influence to neighboring areas, results in agents leaving a trail of influence behind them. This can be used to get agents to spread out and explore the map. Having the value decay slowly will allow agents to circle back again.

Occupancy maps are simply influence maps with a modified influence propagation function [Isla 09]. Influence originates at the target's position, but is eliminated in areas visible to the agents and accumulates in areas hidden from the agents. This then represents the probability that a target might be hiding in an area and allows agents to infer the possible routes the target might have taken through the map.

It is often useful to know the territory owned by rival factions. A simple distance volume, such as 10 miles from the city, does not take into account how the terrain may impact the territory. Figure 6 shows that using an influence map will allow for this territory to conform to natural boundaries such as rivers and mountain ranges.

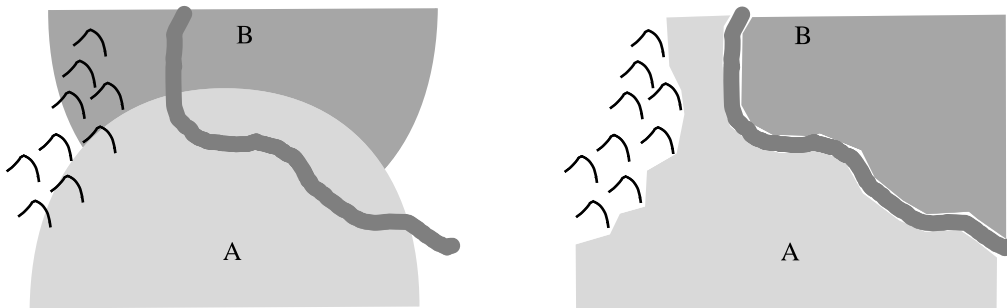

Figure 6 Simple distances (left) is not as accurate as an influence map (right) for determining how natural barriers affect territory.

It is possible to use influence maps to simulate the spread of a dynamic hazard, such as a fire, or an oxygen leak in a spacecraft. Add influence at the source of the hazard and spread it through the connections in the graph. It is useful to modify the spreading function to not spread through closed doors or barriers.

Influence maps are not just for strategy and actions games. They can be used in social systems too. Each character or faction can be represented as a node in a graph and the connections between these nodes can be weighted based on the relationships between the characters or factions. This can then be used to determine the impact an action will have on the social landscape.

5 Entity Specific Data

These are the things your agents need to know about the other active entities in the game. It is the most varied type of knowledge you need to handle in your AI systems and is highly dependent on the type of game you are making. In an action game, this would include information about targets. In a racing game, this would be information about the other cars in the race. In a high strategy game, this may also be information about the enemy civilization, what technology they've developed and resources they have to trade.

Instead of having the behavior layer query the world directly for possible targets, it is often useful to split off these queries into a separate perception system. The perception system is then responsible for checking which relevant entities can be seen or heard for each agent [Welsh 13]. Each agent can then be provided with its own list of entities it is aware of, and what information it knows about these entities.

5.1 How to store information:

There are many different ways of managing and storing entity specific data. The game design requirements will usually dictate which architecture is most suitable. We discuss a few common options below.

5.1.1 Target Information Struct

Action games typically do not require a lot of target knowledge. At a minimum, enemy agents will need to know which targets they are aware of, if they can currently see a specific

target and where they last saw that target. A simple *struct* can be used to store known information about a target, for example:

```
struct TargetInfo
{
   targetEntity
   isTargetVisible
   lastKnownPosition
}
```

An array of these *structs* allows an agent to track multiple targets. The benefit of this method is that it can be very compact in memory and allows fast random access. The downside is that the structure is rigid and all targets will have the exact same members. If additional information is required, extra data members have to be added to the *struct* and the code recompiled. Now all targets will have this new data, even if they do not require this specific information for a specific target. This is still a good method to use if the number of different properties we care to track is small and uniform across most targets.

5.1.2 Stats and Tags

Some games may require more flexible and nebulous information about entities. Tags are simply an optional label we can give a character. For example, we may want to know if a character is *Brave*, or *Happy*, or *Foolish*, and so on. A character can have an array of tags and we can configure these offline at design time, or add and remove tags at runtime to track dynamic changes. If the number of tags is limited, we can use a simple *enum* for each. Or we could use a *Symbol* or *Class* to represent them. [Graham 18].

Stats are almost just like tags, except they have a numerical value associated with them. So instead of tracking if a character is *Brave* or not, we can have a *Bravery* stat and a character can have *Bravery 70%*, or *Bravery 40%*. These fuzzy values allow for more nuanced behavior from our characters than would be possible with binary tags. The values of stats can be fixed and specified by designers, but can also change over time or in response to actions or events in the game. In The Sims, a character with a high *Hunger* stat, will find a smart object that can satiate her *Hunger* and after performing the action, her *Hunger* will be reduced [Bourse 12].

If the information needed is a simple boolean, e.g. hungry or not, a tag is most appropriate. However, if a more fuzzy value is desired, e.g. how hungry is the character, then we required a stat. Tags and stats can be stored either on the specific entity to which they belong, or as Entity-Tag / Entity-Stat key-pairs in a table.

5.1.3 Event History

For some games, it is useful to keep a history of perceived events. It may not be enough to know if an agent can see the player in a restricted area, but also to know if the agent has seen the player enter the restricted area multiple times [Vehkala 13]. Depending on the requirements of the game, this history can be stored in a few different ways. We can add the tag *PreviouslySeenInRestrictedArea* to the player once an agent spots her. Or we can increase a *NumTimesSeenInRestrictedArea* stat on the player each time she is spotted. For an even more detailed history, we can use a database with a time-stamped log of all the perceived events. Table 1 shows an example of a possible history of an agent spotting the

player, then losing sight of her briefly and then hearing and seeing her again a few seconds later.

Table 1 An example of a history of perceived events

Time	Event Type	Entity	Perception Type	Position
135.6	Enemy Target	Player	Visible	[20, 0, 15]
135.8	Enemy Target	Player	Visible	[18, 0, 16]
142.2	Combat Sound	Player	Heard	[16, 0, 23]
143.4	Enemy Target	Player	Visible	[18, 0, 23]
143.5	Combat Sound	Player	Heard	[18, 0, 23]

This history can quickly grow in size and memory if left unchecked. We can keep it manageable by truncating the list to a set number of entries, or discarding entries older than some max time limit. If we need to remember events over a longer term, we can record the more salient information in a more compact, long-term database that can persist for the entire game.

5.2 Public vs Private

Some information is public knowledge that all agents should know, e.g. in a social game, everyone should know that Sam is dating Fred. However, some information might only be known by the specific agents who have perceived the event, e.g. Guard 1 can see the player, but around the corner, Guard 2 cannot. Some games may require all private knowledge. Some may be able to work with all public information. Others may require a mix of both private and public knowledge.

Private information that should only be known by specific agents should be kept local to each agent. For example, each agent should have their own Target Information array to contain the information about the targets they are aware of. Each agent manages their own knowledge and each agent can have differing information about the world.

Public information is common to all agents and should be stored in a central place. All agents can reference this common information and any state or changes will be accessible and known to all agents. A good example is Prom Week, which uses a common social facts knowledge base to track the social interactions between all the characters [Mateas 13].

5.3 What information do we need?

The information to track per entity is highly dependent on the design of the game. A social interaction game, like Prom Week, will require very different information than a stealth action game like Hitman, or an action shooter game like Halo. We cannot cover all the options in a single article. The key is to have access to the minimum amount of information to allow the agents to make decisions according to their design in the game. The more entity specific data knowledge we track, the more complex and believable we can make our agents behavior. We will go into some simple examples for an action game to illustrate the point.

The most commonly used data about targets are visibility and last known position. It

can be useful to store the last seen time, or time since seen, rather than just a boolean flag for 'is visible'. Agents can then still consider targets visible for a short time after losing line of sight. This adds robustness when a target is only momentarily obscured.

Keeping track of the target's last known position, instead of current position, allows your players to trick and manipulate the agents instead of them having omniscient knowledge of the player's whereabouts. Note the use of the term 'last known position' instead of 'last seen position'. An agent may have other senses than sight and can update their knowledge of a target's position if they hear a sound he makes, or if a teammate relays the target's position.

If you do have agents sharing target knowledge, it is useful to have a flag to indicate whether the agent has personally witnessed the target or only has secondhand information. An example where this can lead to better gameplay is for a behavior to trigger the alarm. Consider a player entering the L-shaped corridor shown in Figure 7. He sees one agent immediately in front of him; however, another agent is waiting around the corner, where there is no line of sight to the player. The first, visible agent sees the player and informs his teammate. From the player's point of view, it would be unfair for the hidden agent to activate the alarm based on secondhand information. It would be better for the first agent, who can actually see the player first hand, to run for the alarm. Not only does this make more sense to the player, it provides him the opportunity to prevent the alarm from being triggered.

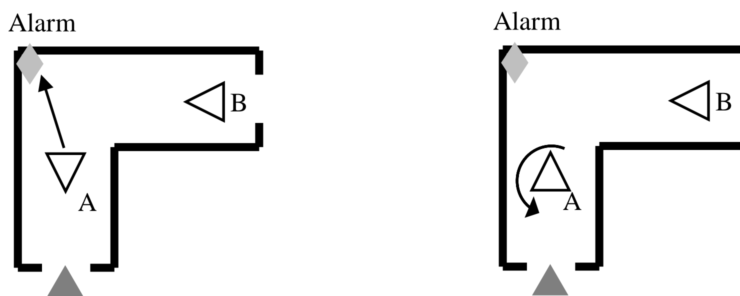

Figure 7 On the left, agent A has witnessed the target first hand and should go for the alarm, while agent B only has second hand information. On the right, agent A has heard the target, but not yet seen him. Agent A should turn around to first see the target before running for the alarm.

Additionally, more meaningful behavior can be crafted by keeping track of the types of information perceived. Tracking whether an agent has seen the target versus only heard the target can result in more believable behavior. In a similar situation to the one above, if the first agent had his back to the player and just heard a noise, it would make more sense for the agent to turn around to see the source of the noise, instead of running directly for the alarm switch.

Another useful property is to store whether the target is reachable. A failed path finding query can take significant processor time. It is better to avoid continuously performing path finding queries that you know will fail. If you fail to find a path to the target, you can mark that target with an unreachable flag. This way you know not to try path

to that target again, until the situation has changed. Once the target has moved significantly away from this unreachable position, or the level has changed (such as a previously locked door is now open), you can clear this unreachable flag to allow another attempt at finding a path to the target.

Knowing that a target is unreachable is also useful for constructing fallback behaviors. A melee focused agent could select a different target, one he can reach. Alternatively, the agent could switch to a range weapon and attack from a distance. Having this information allows your agent to make these decisions intentionally.

Interesting behavior can be crafted by knowing what weapon the target is wielding and where he is aiming. A trivial example is having the agent play a sideways dodge animation if he is being aimed at, however it may be unnecessary to dodge if the opponent only has a melee weapon and no ranged weapon. A more complex behavior might be for a swashbuckling game. If the player is armed with a single-shot flintlock pistol, it would make sense for the agent currently threatened by the pistol to remain still while other agents, who are not being aimed at, can cautiously circle around and approach from a flanking direction. If the player only has his sword equipped, the agents can all approach more aggressively.

These examples are all very action game focused; however, the same principle applies to other genres. By having each agent determine this information based on whether he can perceive that information, makes the agents more believable and allows for the player to apply more interesting strategies to engage with the agents.

6 Conclusion

Whenever we make game AI systems, we are translating parts of the game design into something the computer can execute. Agents require data to make decisions and exhibit the desired behavior. Richer behaviors can be crafted by providing richer information to agents. Information can be embedded in the world, or gathered via messages and sensory systems. When constructing behaviors for an agent, we should take the time to plan what data will be required for the agent to exhibit the behavior, rather than focus on which behavior selection algorithm to use.

7 References

[Carlisle 13] Carlisle, P. A Simple and Robust Knowledge Representation System. In Game AI Pro. ed S. Rabin. Boca Raton, FL: CRC Press, pp. 433-440

[Tozour 04] Tozour, P. 2004. Search space representations. In AI Game Programming Wisdom 2, ed. S. Rabin. Hingham, MA: Charles River Media, pp. 85-102.

[Josemans 17] Wouter Josemans. Putting the AI back into Air: Navigating the Air Space of Horizon Zero Dawn. Game AI North 2017. Available online (https://www.guerrilla-games.com/read/putting-the-ai-back-into-air)

[Brewer 17] Brewer, D. 2017. 3D Flight Navigation Using Sparse Voxel Octrees. In Game AI Pro 3, ed S. Rabin. Boca Raton, FL: CRC Press, pp. 265-274

[Bourse 12] Bourse, Y. 2012. Artificial Intelligence in The Sims series.

http://www.yoannbourse.com/ressources/docs/ens/sims-rapport.pdf (accessed November 11, 2020).
[Brewer 13] Brewer, D. 2013. Tactical Pathfinding on a NavMesh. In Game AI Pro, ed S. Rabin. Boca Raton, FL: CRC Press, pp. 361-368
[Mark 15] Mark, D. 2015. Modular Tactical Influence Maps. In Game AI Pro 2, ed S. Rabin. Boca Raton, FL: CRC Press, pp. 343-364
[Dill 15] Dill, K. 2015. Spatial Reasoning for Strategic Decision Making. In Game AI Pro 2, ed S. Rabin. Boca Raton, FL: CRC Press, pp. 365-388
[Isla 08] Isla, D. Building a Better Battle: HALO 3 AI Objectives. GDC 2008. Available online(http://www.gdcvault.com/play/497/Building-a-Better-Battle-HALO https://web.cs.wpi.edu/~rich/courses/imgd4000-b12/lectures/halo3.pdf)
[Gallant 17] Gallant, M. Authored vs. Systemic: Finding a Balance for Combat AI in 'Uncharted 4'. GDC 2017. Available online (http://www.gdcvault.com/play/1023949/Authored-vs-Systemic-Finding-a)
[Isla 09] Isla, D and Gorniak, P. Beyond Behavior: An Introduction to Knowledge Representation. GDC 2009. Available online (http://www.gdcvault.com/play/1267/(307)-Beyond-Behavior-An-Introduction)
[Welsh 13] Welsh, R. 2013. Crytek's Target Tracks Perception System. In Game AI Pro. ed S. Rabin. Boca Raton, FL: CRC Press, pp. 403-411
[Graham 18] Graham, R. and Brewer, D Knowledge is Power: An Overview of Knowledge Representation in Game AI. GDC 2018. Available online (https://www.gdcvault.com/play/1025172/Knowledge-is-Power-An-Overview)
[Vehkala 13] Vehkala, M and De Pascale, M. Creating the AI for the Living, Breathing World of Hitman: Absolution. GDC 2013. Available online (https://www.gdcvault.com/play/1017802/Creating-the-AI-for-the)
[Mateas 13] Mateas, M and McCoy, J.An Architecture for Character-Rich Social Simulation. In Game AI Pro. ed S. Rabin. Boca Raton, FL: CRC Press, pp. 515-530.

10 Biography

Daniel Brewer graduated in 2000 with a BScEng in Electronic Engineering, focusing on Artificial Intelligence, Control Systems and Data Communications. Since then he has become a veteran game developer. As lead AI programmer at Digital Extremes, he has been instrumental in bringing the co-op, online, multiplayer action shooter, Warframe(2013) to life, as well as continuing to further develop, maintain and upgrade the game AI systems in the game. Other notable titles include Halo 4 multiplayer DLC packages(2012), Darkness II (2012), BioShock 2 multiplayer (2010) and Dark Sector (2008). Over the years, he has presented numerous talks about various facets of AI at the Game Developers Conference.