AI for Autonomous Agents in Games and Simulations

Student Name: James King

Supervisor Name: Magnus Bordewich

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

January 19, 2014

Abstract — **Context:** Designing the core autonomous behaviour for characters in a video game where those characters are expected by the player to perform tasks for themselves poses many interesting challenges. This paper explores the tasks faced when developing a capable and somewhat convincing set of Artificial Intelligence routines within a simulated zombie epidemic real-time strategy game with player allocated goals. **Aims:** As the game player doesn't have direct control over the characters, but instead can assign tasks, algorithms to allow the characters to plan and implement those tasks in a cooperative and efficient manor will need to be designed. These algorithms must not be computationally expensive to allow for many agents to act in real-time, and also express superficially convincing human-like behaviours. **Method:** At least two conceptually distinct designs for character behaviour will be designed, and additionally several slight variations of each. These will all be compared in terms of the ability for characters to achieve assigned tasks, avoid threats, and system resource usage. A hybrid between the main designs may be explored if one is not universally better than the other. **Proposed Solution:** A Subsumptive architecture will initially be explored, followed by a Belief-Desires-Intentions model. Following this, each approach will be adapted to experiment with different character behaviours and strategies including weighting self defence against fleeing, and group formation approaches.

Keywords — AI, BDI, cooperative, multi-agent, subsumptive, task planning, video game

I INTRODUCTION

A Context

Video game worlds are often populated with Non-Player Characters (or NPCS), the behaviour of which can make or break the game. A great deal of effort is required when designing the routines used by these characters if they are to satisfy some major requirements; they should behave with approximate rationality, they should provide an appropriate challenge for the player, and if the characters represent humans (or at least animals) they should act in a believable way. For Real-Time Strategy (or RTS) games the difficulty is further accentuated by the necessity of the game to support perhaps hundreds or even thousands of these characters in an environment simultaneously. This combines the previous three requirements with a fourth: efficiency. Designing solutions for the first three key requirements is often more of a creative task rather than a purely methodical one, and entails an element of subjectivity. The fourth requirement is easier to test, but the difficulty

in maintaining a minimum performance quality heavily depends on the complexity of the solutions for the first three goals.

B Problem Domain

This project aims to explore possible implementations of an NPC behaviour system designed to realise the central gameplay of a work-in-progress RTS game. The components of the game that existed prior to the beginning of this project form a zombie epidemic simulation. A city environment is procedurally generated, producing buildings separated by streets constructed on a tile-based grid. When this process is complete, a number of human characters are dispersed around the environment, a fraction of which initialise as a zombie. The simulation then begins, with the zombies navigating towards the nearest visible human, and humans attempting to run away from danger if any is present (and otherwise just wandering randomly). If a zombie catches up to a human it may attack it, reducing a stored health value for that human and infecting that human with some probability. If a human loses all their health and is infected, they become a zombie too.

The conflict is clearly one-sided, not least because the zombies are smarter than the humans. The humans have little regard for their surroundings other than the locations of visible enemies, and so often run directly into the inner corners of walls. This would obviously not a sufficient behaviour; looking at the core requirements for a decent NPC it fails at being rational, at providing an appropriate challenge (the humans don't stand a chance), and they certainly don't act like humans. Their only redeeming quality is that their simple AI isn't too computationally expensive, so thousands of them can be supported simultaneously.

C Project Aims

At the very least the artificial intelligence routines explored in this project should improve each human agent's survival ability. This may mean attempting to escape when cornered, deciding when it is rational to attack in self defence, and implementing strategies to hide from danger. Later on in the project a player will be able to assign actions for the humans to complete, such as instructing a group to navigate to a specific location, or to construct barricades out of material found in buildings. These commands may conflict with an agent's necessity for self preservation, so the processes developed should intelligently decide when it is rational to belay an order.

On the topic of player-specified tasks, an efficient path finding technique will need to be implemented that balances time required to processes with optimality of the path found. This algorithm will be essential for when agents are directed to a specified location by the player, but also useful for attempting to navigate when performing other tasks or to detect if a path exists to bypass some barricades (establishing whether they are safe). Some tasks may not be assigned to a specific agent but will rather be goals to be achieved by the collective group. For example, the player will be able to instruct that a barricade should be built in a specific location. In this instance the human agents should automatically distribute sub-tasks between themselves in order to achieve the common goal efficiently.

The two core approaches to be compared are a Subsumptive architecture and a Belief-Desires-Intentions model. A subsumptive architecture features a stack of behaviour layers

where each may in turn choose to either act or subsume control to the next layer, starting with the top of the stack. The first layer in the sequence to specify an action is heeded, and the rest are ignored. This design usually relies on complex behaviour emerging from complimentary layers. A Belief-Desires-Intentions model is a radically different approach, where each agent records a formal representation of the world from which a set of attainable goals are found, a subset of which is acted upon according to estimated rationality.

D Deliverables

These following core elements will be created before the end of the project.

1. **Expanded Core Game** — The central mechanics that the AI system will interact with must be implemented. These include the creation of game objects that represent a resource to be used when building barricades, the system to support the construction of the barricades themselves, and an expanded user interface which allows a player to select groups of agents and assign tasks. Additionally facilities to be used by the two proposed architectures should be provided, such as a path-finding implementation.

2. **Subsumptive Prototype** — An agent design using a subsumptive architecture will be developed, implementing the core requirements of the human NPCs. This prototype will be constructed by breaking up the human's functionality into a stack of behaviours, from which a successful strategy is designed to emerge. Relies on deliverable 1.

3. **BDI Prototype** — A distinct agent design using a BDI architecture will also be developed, meeting the same requirements as the subsumptive approach. This prototype will use a personal abstracted representation of the environment for each human agent, from which a set of achievable goals are found using a goal evaluation algorithm. These goals are then further reduced into a set that is calculated to yield the most utility, and are simultaneously possible. Relies on deliverable 1.

4. **Analysis & Final Report** — Both completed prototypes will be analysed and compared in terms of agent survival rate (without player intervention), efficiency at which goals specified by a player are achieved, prevalence of unnatural behaviour, and system resource usage. The results of this comparison will be compiled into a report, along with the identification of the prototype that is most suited for the problem and an explanation for why that prototype was chosen. The report will also cover possible areas of additional research exposed by this project. Relies on deliverables 2 and 3.

II DESIGN

A Requirements

The main deliverable can be broken down into a set of core functional and non-functional requirements, each with a relative priority rating. These requirements have been categorised

into the Core Requirements that the existing game will be extended to fulfil, and the AI Requirements that the two agent design prototypes must independently satisfy. Functional requirements are prefixed with 'FR', while non-functional requirements are prefixed with 'NFR'.

i Core Requirements

FR 1.1	The number of initial agents and the world size must be configurable.	Med
FR 1.2	NPCs must be able to navigate between any two positions in the game environment if a path exists.	High
FR 1.3	Game objects representing a resource used when constructing barricades must exist within the game environment.	Med
FR 1.4	NPCs must have the ability to construct barricades, ideally using resource objects (FR 1.3).	High
FR 1.5	A player must be able to select a group of human agents through the user interface.	Med
FR 1.6	A player must be able to instruct selected agents (FR 1.5) to move to a chosen location.	Med
FR 1.7	A player must be able to instruct selected agents (FR 1.5) to block off a specified area with a barricade.	Med
FR 1.8	An analysis of the scalability of the game environment must be produced, exploring how the environment size and number of agents affects performance.	High
NFR 1.9	The simulation must be able to maintain an average frame time of at most 16.67ms with a world size of 128x128 tiles and 250 active agents.	Med
NFR 1.10	Moving objects within the game environment must not pass through solid geometry outside of exceptional circumstances that would not occur during standard gameplay.	Low
NFR 1.11	NPCs must demonstrate an ability to recognise whether a specified object is visible, and not react to things they should not be aware of.	Low

ii AI Requirements

FR 2.1	Human agents must exhibit a strategy for avoiding or eliminating nearby visible threats.	High
FR 2.2	Human agents must be able to navigate to a position as instructed by a player (using FR 1.8), unless this conflicts with FR 2.1.	High
FR 2.3	Human agents must possess the ability to autonomously construct barricades to block off areas without player direction.	Med
FR 2.4	Human agents must possess the ability to construct barricades to block off areas specified by a player (using FR 1.9).	High
FR 2.5	An analysis of the survival rate and efficiency at which player assigned tasks are achieved must be produced.	High
NFR 2.6	Agents should exhibit erratic or unnatural behaviour only rarely, if at all.	Low

B Implementation

Several key elements of this project require a non-trivial amount of planning and design, the details of which will be described in this section.

i Path Finding & Navigation

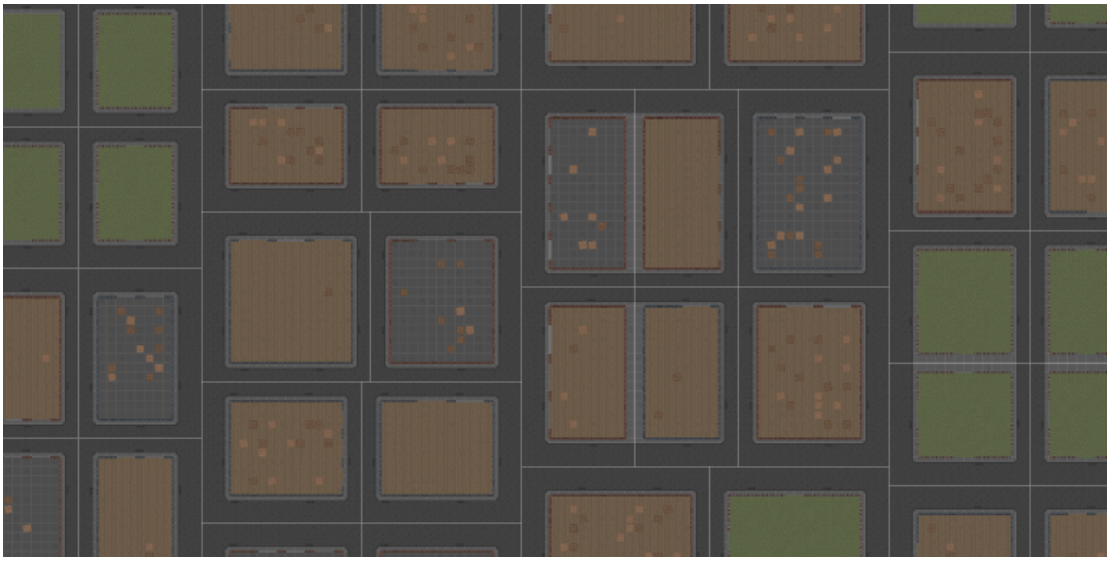

Figure 1: A diagram to show the structure of a procedurally generated city environment, featuring blocks of buildings separated by roads. Each block is bounded by a white rectangle.

The problem of finding efficient paths through virtual environments is often faced when developing AI for games, and so has been explored extensively. Generally the A* algorithm (or one of the many variants) is used, although while it guarantees a high quality path it may require a significant amount of time to find it (depending on the complexity of the environment). When observing the structure of the worlds produced by the procedural city generation algorithm (See figure 1) it is easy to spot aspects that could potentially be exploited to produce an A* adaptation which required less time to execute.

When imagining how we would solve the problem of navigating between two disparate positions in a city in the real world, we don't think on a metre-by-metre basis. We plan our overall route street-by-street, and only worry about how to navigate down each street when we reach it. A similar approach can be implemented here, where an agent first applies the A* algorithm on a graph representing the road network of the city, and then applies a more refined A* instance along each street when it reaches it. A refined path is also found from the agent's initial position to the nearest street, and then from the final street to the destination. For cases where the street-based route takes a corner, the refined A* instance should look for paths between the start of the street preceding corner to the end of the street following it, in case the block that they both neighbour can be cut across.

This algorithm should produce results very close in length as to when applying a single instance of refined A* across the entire route, with a far shorter execution time for all but the most trivial instances. If the implementation still leads to a performance bottleneck then it could potentially be improved by caching the paths found along individual streets, which can then be looked up instead of calculating the same paths multiple times. This will particularly be beneficial when a large group of nearby agents are travelling to the same destination.

ii Vision & Collision Testing

Visibility testing and collision detection are similar problems, and can be largely solved with the same algorithm. For collision detection the program is testing if an object (simplified as its bounding rectangle) can be 'swept' along a vector representing the movement it wishes to make during the current simulation step. If the swept rectangle intersects with solid world geometry, the object's movement is pulled back to the furthest it could go without such an intersection. Similarly, when testing the visibility of an object you are essentially testing whether it can be swept towards the observer without being blocked by any solid geometry, thereby detecting if its image would be blocked by a wall.

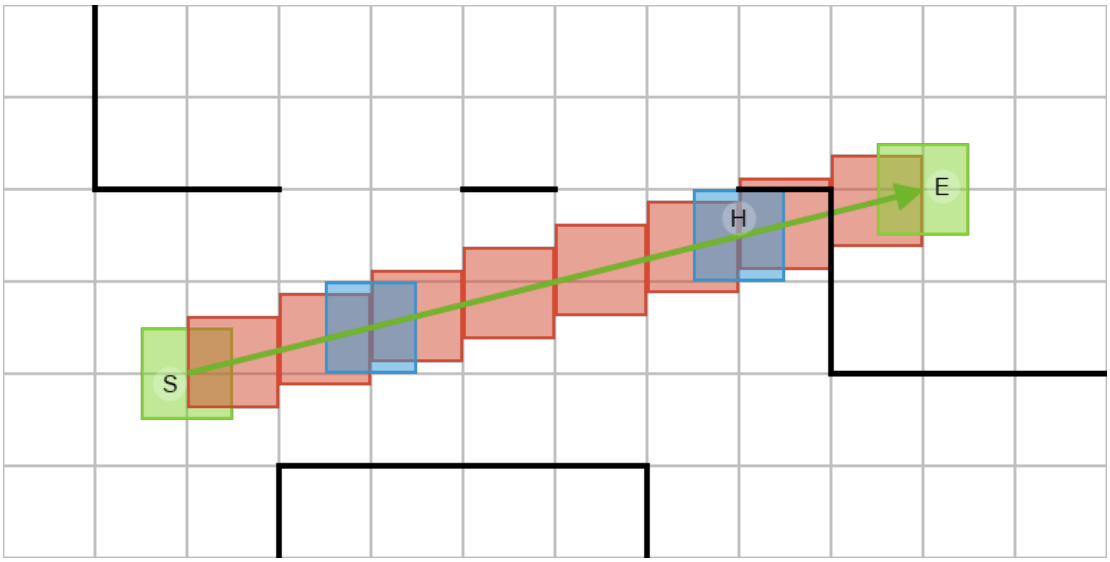

Figure 2: A diagram to show the world geometry intersection checks performed when sweeping an object's bounding rectangle along a trace line. The object starts at position S, and its destination is position E. Black lines represent solid walls. The red boxes show iteration steps where a horizontal collision test is performed against possible obstructions to the right, and the blue boxes are iteration steps where vertical collision tests are performed upwards. Position H is the projected box location where a collision is first detected, and so will be the position the object is moved to.

This sweeping action is the costly part of the technique, an optimal solution will limit the number of intersections tested with this swept area to the absolute smallest number required. As in this scenario only walls can intersect the object, and walls are found along the edges of tiles, the distance along the movement vector that the object is swept can be

incremented by amounts that exactly move it to the next tile boundary that would be met. This method is illustrated by Figure 2.

This technique compares well against the usual naïve approach of sweeping a certain fixed distance along the movement vector each iteration. It is guaranteed to find an intersection if one exists, and calculate the exact distance the object would have to travel to reach it rather than just the previous iteration location when using a constant step distance. It also requires fewer iterations, assuming a constant step distance approach uses a small iteration delta (which it would need to for accurate results).

iii Subsumptive Prototype

Central to a subsumptive agent architecture is a stack structure upon which behaviour layers can be pushed. This is relatively simple to implement using an Object Oriented approach, using an abstracted behavioural layer class with a method to be overridden called when that layer should decide to act. This method will return *true* if an action is performed by that layer, otherwise $false$. The first layer in the stack will be invoked first, and if that layer returned $false$ the next layer will be invoked. This continues until either a layer returns *true*, or the last layer has been invoked. The proposed layers to be implemented are listed below, with the top-most layer first.

- **DropResource** — Drop a carried resource if in danger or at a barricade under construction
- **SelfDefence** — Attack a pursuer if they are too close
- **Mob** — If there are many more humans than zombies nearby, head towards a zombie
- **VacateBlock** — If carrying a resource or in danger while inside a building, run outside
- **Flee** — Run away from nearby danger
- **FollowRoute** — Follow a route assigned to this agent
- **PickupResource** — If standing on a resource and not in danger, pick it up
- **FindResource** — If not in danger and not holding anything, move towards a resource
- **HarvestResource** — Attack a nearby object that yields resources when broken
- **SeekRefuge** — When outside and neighbouring a safe-looking building, run inside it
- **Wander** — Walk around randomly

This particular configuration of behaviours should satisfy each requirement of the AI prototype, using complimentary sets of layers to allow for more complex compound behaviours to emerge. The survival strategy is implemented with the $SelfDefence$, Mob, $VacateBlock$, $Flee$ and $SeekRefuge$ layers, $FollowRoute$ for travelling to player designated locations, and the remainder (along with $VacateBlock$ and $SeekRefuge$ again) for constructing barricades autonomously.

The survival aspect functions with several emergent sub-strategies. The $SelfDefence$ behaviour will attack a zombie that is extremely close to the agent. This is expected to occur when the zombie catches up to the agent due to a movement speed advantage, or if the agent has been chased into a corner. At that point attacking the pursuer is the only rational thing left to do. The Mob behaviour actively directs an agent towards a nearby zombie if that agent predicts that a decent number of other agents will perform the same action, enough to overwhelm the zombie. When the agents are close enough, the $SelfDefence$ behaviour will take control and cause the agents to attack the surrounded zombie. This strategy should lead to zombies being eliminated in safe situations where it is unlikely for a human agent to take much damage. $VacateBlock$ and $Flee$ will help avoid situations where being in close proximity to a zombie is unnecessary, and $SeekRefuge$ ensures agents aren't exposed outside.

Barricade construction relies on a similar interaction between several behaviours. The process usually starts with an agent wandering around outside, carrying no resource object. The $SeekRefuge$ behaviour will lead it to navigate inside a building, within which it may find objects that yield resource items when damaged when obeying the $HarvestResource$ behaviour. When the object breaks into its component resources, the $PickupResource$ behaviour is followed and the agent now holds a resource item. The $VacateBlock$ layer instructs the agent to leave the building if it is holding a resource, and as soon as the agent leaves it drops the item as instructed by the $DropResource$ layer. The agent then attempts to $SeekRefuge$ again, and the cycle repeats. Resource items are continually dropped onto tiles neighbouring an exit to the building, which pile up to form barricades. When the barricade becomes large enough, the tile it belongs to becomes solid and the agent automatically finds a different path to leave / re-enter the building, thereby dropping items on a different tile.

This solution is quite trivial to implement due to the simplicity of each individual layer. Some behaviours, such as Mob and $Flee$, provide some scope for experimentation by testing different thresholds for when to perform that action. Additionally, it may be interesting to test whether the agents perform best if they all posses the Mob layer, or if there is some ratio of agents that refuse to actively attack zombies that leads to a higher overall survival rate.

iv BDI Prototype

For a BDI agent architecture we must design each of the three core components (Beliefs model, Desires identification, and Intentions filtering) as individual systems. After this, for each type of intention a planning mechanism must be implemented.

The Beliefs component involves designing a system for representing aspects of an agent's environment that could influence its actions, updating that internal representation as the agent perceives stimuli. The easiest structure for this model of the environment would be to store every detail exactly as it is perceived; but this would be difficult to parse, would hold unnecessary detail, and could result in too much memory being consumed. A better approach would be an abstracted model, perhaps considering two levels of abstraction as with the path-finding solution (see section i). The highest level abstraction will be at a city block scale, considering observed attributes of the contents of each block. Each agent will track a few key properties about each block; the last time they observed it, whether that

block contains a building, the number of human agents seen within that block, the number of zombie agents, and the number of barricade resources seen. Each agent will also track the locations of observed agents and resources for their local environment, information that will be discarded when they travel a specified distance from those locations to save memory. Agents will also record whether they are within a building or outside, their health value, and whether they are holding an item.

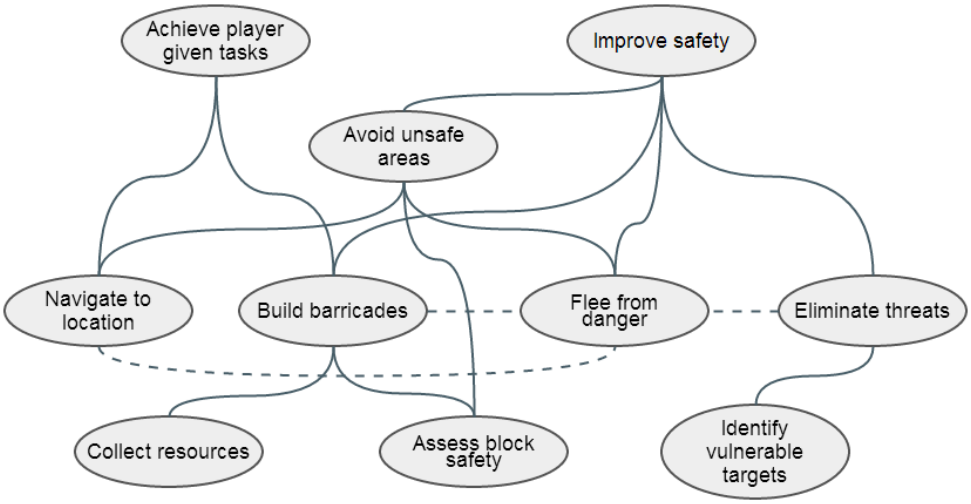

Figure 3: A diagram to show a breakdown of the main goals for a human agent into sub-goals, with possible conflicts signified by dashed lines.

Desire identification is the next component requiring planning, which involves designing a mechanism to parse the beliefs model to find a set of achievable goals that align with the higher aspirations of the agent. For this specific scenario, these aspirations will be to improve safety and achieve player-given tasks. These high level goals can be broken down into sub-goals as described by Figure 3.

v Development Environment

The original game that this project is extending is written in C#, so that will be the primary programming language used. Microsoft's Visual Studio 2013 will be used as an integrated development environment for its C# compiler, debugging and profiling tools, and Sublime Text 2 for any peripheral text editing. Development will mainly be on a Windows desktop PC (Intel Core i5, AMD 7850), using a Windows laptop (Intel Core i7, nVidia GeForce GT 540M) to test performance on a device with an nVidia GPU. The OpenTK library is used to expose OpenGL bindings for .NET, and is the only non-standard library used.

vi Analysis Strategy