Determining the Requirements of Creating Realistic Artificially Intelligent Game Agents

by

Ryan R Panuski

B.S. General Science, Pennsylvania State University, 2006

Submitted to the Graduate Faculty of

University of Kutztown in partial fulfillment

of the Requirements for the Degree

Master of Science

University of Kutztown

2010

Approved:

(date)	(Advisor)
(date)	(Chair, Department of Computer Science)
(date)	(Dean, Graduate Studies)

Abstract

Artificially intelligence agents when used within a game environment help to provide structure and immersion. In order to have the greatest amount of enjoyment while playing a game, the immersion has to be complete. To achieve this, the agents themselves have to become highly realistic and convincing enough that the player does not know the difference between the agent and another player.

This thesis works to discover what exactly is needed as far as a technological implementation to perceptually achieve this in a game environment. Working with different artificial intelligence methodologies, the objective can be extrapolated and expounded upon into various additional disciplines.

Acknowledgements

I wish to express my appreciation and thanks to all of those who have helped to make this thesis possible. This includes my family without whose support and understanding I would not have gone back for my Masters.

To Roger Grayson, a loyal and supportive friend who helped suggest the topic and was a great sounding board.

To my adviser Dr Oskars Rieksts who's assistance was invaluable in the completion of this thesis.

To Dr Dale Parson who took the time to read and share his ideas and wisdom; as well as helping to formulate the structure on which the thesis is based.

Thank all of you again for helping and supporting me in this endeavor.

Table of Contents

Figure Index

Chapter 1: Introduction and Artificial Intelligence in Game Applications

Creating realistic artificial intelligent agents is a worthwhile and challenging goal. Not only does it create a strategically challenging opponent in a game, it also allows for a more realistic immersion within the application itself. The more challenging and realistic an opponent within a game application, the more fun is to be had while playing. Another reason for having intelligent agents within a game environment is because game environments are created in an effort to have a great degree of realism and mimicry of the outside world. As such games would be an ideal test bed for robotics or artificial intelligence in which to be simulated and developed. Obtaining a greater degree of realism when creating agents for games, increases the realism to be achieved when the information learned is applied to other areas.

The problem to be solved within this research program is *In a game environment what does a realistic artificially intelligent agent require?* In order to create a realistically intelligent agent one must assess certain aspects of how the agents acquire and represent knowledge. The way in which an agent acquires and represents knowledge is paramount in the creation of a realistic agent. Once the knowledge is obtained, then it must be used to plan a way to solve the problems given to the agent. Each of these points are key in the creation of an intelligent agent which can interact within a given environment.

1.1 General Game Information

Historically game developers have been focused on making a game fun and enjoyable to play as opposed to creating an agent within the game that will be a challenging opponent. When creating a game, developers would always eliminate features that made the game challenging or realistic if it was seen to possibly have an adverse effect upon the fun factor.

There are cases in which the fun factor would not be impacted negatively if the game was exceedingly challenging. An example of this would be computer games that were highly structured and the required objectives were dependent upon very specific rules. Such games could include but not be limited to chess, checkers and multiple card games. The generalized categories of these genres would be games of skill and chance.

Game developers use varying methods to accomplish the acquisition of knowledge. The most commonly used technique when acquiring knowledge used by game developers is structured omniscience. Structured omniscience is described as though the game application itself has access to all information contained within it; however, game developers limit how much of that information the agent itself is allowed to access at any given moment. Figure 1.1 shows an agent within a game which has been given complete knowledge of its environment within a subset around itself, designated by a red circle. The agent shown, is able to access all information within the subset of the information of the application which is relevant to the agent. As the agent is allowed to acquire knowledge without any restrictions within the circle presented this creates an agent that is not very realistic and will pursue the player without ever having seen the player.

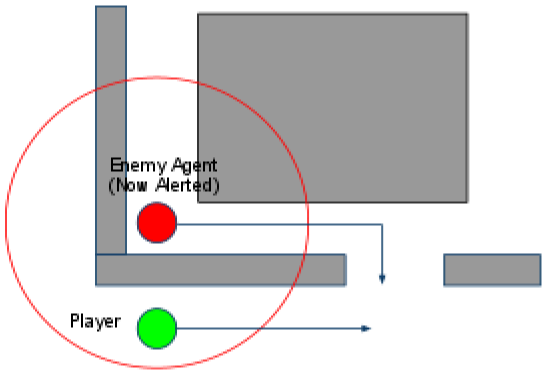


Figure 1.1: Agent knows everything within its circle.

The problem shown in Figure 1.1, is a profound example of a loss of realism in behavior. Game developers have worked to fix the issue caused in that situation. One of the ways that this issue is resolved is by limiting the knowledge reacted upon by the agent until after a threshold is crossed. After the threshold is crossed, the agent will actively pursue the player. Figure 1.2 illustrates this. The agent will not react to the player until it sees the player in its field of view. As with Figure 1.1 the red circle illustrates the subset of knowledge that the agent possesses. However the light green area represents the knowledge the agent will not react to until the player is visible.

This approach also has realism issues, as illustrated in Figure 1.2. In part C of Figure 1.2, the enemy agent knows exactly which direction to turn to pursue the player even though it would be unable to know which direction the player turned in the hallway once the player had left the agent's field of view. Usually realism issues of this type are completely ignored in modern games.

An example of this type of realism being ignored is seen in a game which was released in 2008 called *Crysis*. The agents in the game know exactly where the player is at all times once the player has been seen within the game environment. The agents in *Crysis* are always facing the direction to which the player is located, even if there isn't any realistic way they could know where the player is. For example if the player ducks down behind an object as cover and moves along hidden behind the object, when the player moves out of cover in a flanking maneuver the agents will be facing them and begin to attack, even if there was no previous way of knowing where the player would move.

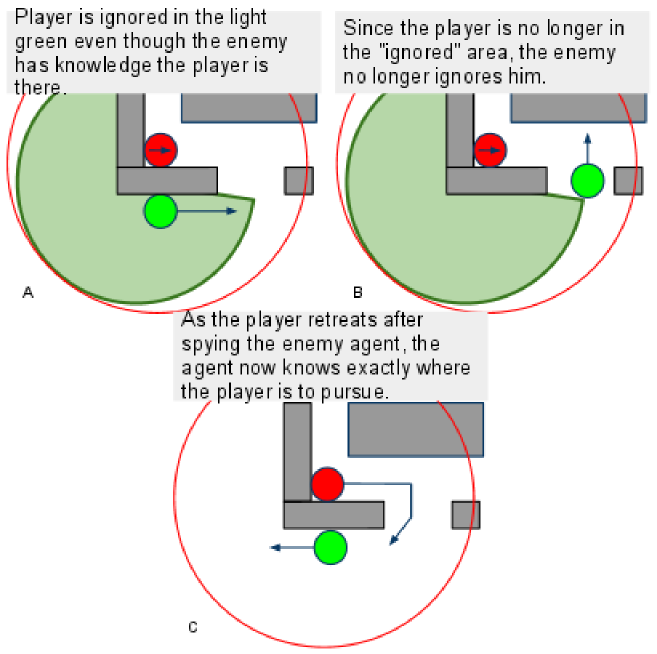

Player is ignored in the light green even though the enemy has knowledge the player is there.
Since the player is no longer in the "ignored" area, the enemy no longer ignores him.
A
B
As the player retreats after spying the enemy agent, the agent now knows exactly where the player is to pursue.
C

Figure 1.2: Knowledge acquired is not acted upon until the player is seen.

Due to such issues in modern games, a large portion of the realism in the agent's portrayed behavior is lost and the illusion of artificial intelligence suffers as a result. "Artificial intelligence" in this case refers to the agent's actions and behaviors being realistically perceived as intelligent when viewed from a humans perspective.

As recently as 2006 one of the chief methodologies used in the planning phase for agents in games was Finite State Machines (FSM's). (Orkin, 2006) Although FSM's are not a way with which to plan how to resolve an encountered dilemma, they are a means to structure a specified reaction to a defined input type. A FSM is a system of states which have transitions between them. Each of these states can be structured to be similar to a mode of behavior. The transitions between them are governed by inputs which influence the state changes. Figure 1.3 is a very simple FSM with two states and the transitions between them.

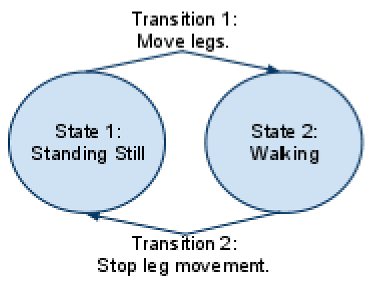


Figure 1.3: Finite State Machine

Game developers will choose states to create inside the FSM for their agents within games that relate to their specific setting or application through which they are to be applied. In other words the states of the FSM are constructed using the actions or behaviors the agent in the game system displays as guides. The states could be simple as shown in Figure 1.3 where they are defining an action or they could be fairly complex defining an entire behavior. In addition sometimes the transitions in games will be governed by fuzzy logic, which can help the agents to be less predictable.

In addition to FSM's game developers have used methods such as goal oriented action planning (GOAP) which was used in the 2005 release *F.E.A.R.* GOAP is a decision making architecture that allows characters to decide not only what to do, but how to do it. (Orkin, 2003). Another method that has been used for a planner in games is a Hierarchical Task Network (HTN) which was used in the recently released *Killzone 2* in 2009.[1] Both of these methods will be elaborated upon further in the paper.

The problem of greatest significance within the game development community is obstacle avoidance. This is due to the understanding that agents will have to move appropriately between two different points in their environment. With this understanding there is a great emphasis put on path-finding for modern games. During development many game developers will have agents determine a path for traveling between two points, this is done by plotting a course using a chosen algorithm. The path found is saved and in most cases is not able to be adjusted dynamically once the game has been released. When game developers create these paths they use numerous methodologies. The two most prominent are way-point graphs and navigation meshes.

A way-point graph is a series of points that have been placed in a game world environment. Each of these points represents a node within the graph. The nodes will be placed in the game world usually by a map designer. Additional nodes can be placed by the game engine itself, if it finds a need for additional nodes to be added to the graphs being created. A game engine is the core functions of the game application, it usually handles various tasks such as animation, physics, game-play and path-finding. Nodes created by the game engine are usually extensions to currently existing graphs which had been created by the map designer. Extending the graphs by adding additional nodes is done during development and not during run time. Game engines also usually will have a debug or development mode to them in which case they can have more options and abilities to afford the developers of the game numerous ways with which to display needed information. This information can be used to display information within application variables, or to have possible paths calculated in real time as opposed to using precomputed values, in addition to many other features which can be added as needed.

During run time the game engine usually does not calculate the paths required for the agents to traverse as the computation time would be to critically intensive when the processor cycles could be better utilized for rendering graphics, physics calculations, or numerous other calculations required by the game application. The only time when the game engine might calculate agent traversal paths is during a level transition, if possibly an agent was in a non precomputed starting position. Then the calculation of the path for moving the agent to the required position could be handled either during the level transition or possibly during run time as the information was needed.

Figure 1.4, shows an example of a way-point graph of a simple environment. The light blue box represents a river and therefore impassable terrain. Grey boxes represent buildings or obstacles. A yellow dot represents the goal on the right side of the environment. Arrows represent individual paths from one node to another. The path between the start position and the goal position begins in the top left and terminates in the middle right. Pink arrows represent invalid connections between points, due to the river being impassable, and thus are not represented in the final graph. A specific path is not shown to be traversed; instead multiple options are given. This is because when choosing a path there are many factors to be taken into account, in addition to the connections shown in Figure 1.4. This will be explained later in the paper.

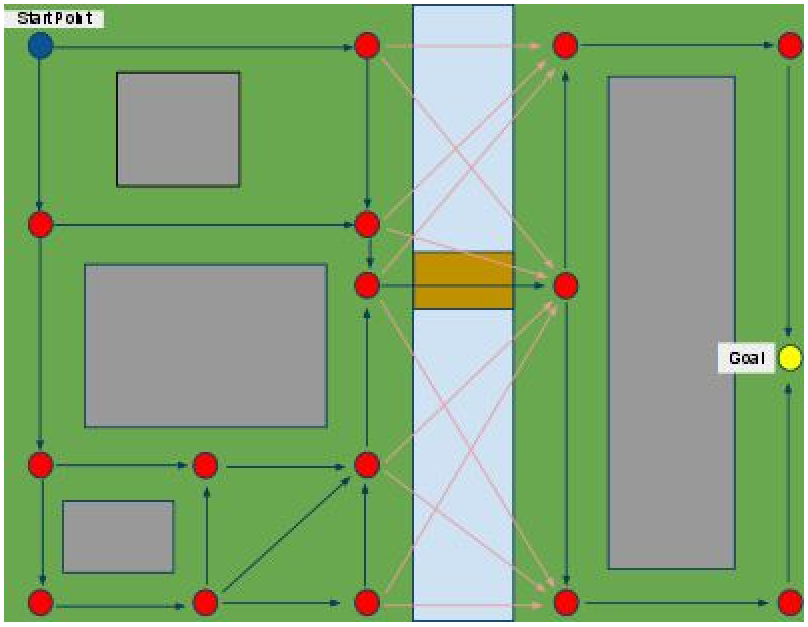


Figure 1.4: Example of a simplistic way-point graph.

As mentioned above a second method used to find a path for an agent within an environment is called a navigation mesh. Navigation meshes are a different way to model the environment. Usually navigation meshes are automatically annotated into the environment by the game engine itself, where-as way-points are generally placed by the designers when created. These annotations are handled during development of the game when the levels are created initially. Additionally way-points and navigation meshes differ because, instead of singular points being in open spaces as is the case in Figure 1.4, entire open areas are marked as polygons. These polygons have points associated with them either in the center itself or along the edges. A navigation mesh can be thought of as a photographic negative of a way-point graph. The reasoning for this is because in a photographic negative, the light areas appear dark and the dark areas appear light in what is called a tonal inversion. Figure 1.5 depicts a section of an environment broken down and represented both as a way-point graph and as a navigation mesh. The graph and mesh are inverses; where there are nodes in the way-point graph there are no nodes in the navigation mesh and vice-versa. In the navigation mesh shown in Figure 1.5 one may wonder why there is not a node in the top left corner centered within that polygon and why the start position, represented by the blue dot, points to the two sides of the polygon. The reason is that in the mesh, or graph that is formed in these situations, nodes that are too close together are usually pruned out. Therefore the central polygon node in the top left polygon is removed for efficiency.

In Figure 1.5 with the differences shown in how both a way-point graph and a navigation mesh interact within an environment, it is easy to see that either one of them can be used in different situations depending upon the complexity of the world that is being inhabited. In a complex environment usually navigation meshes will be used.

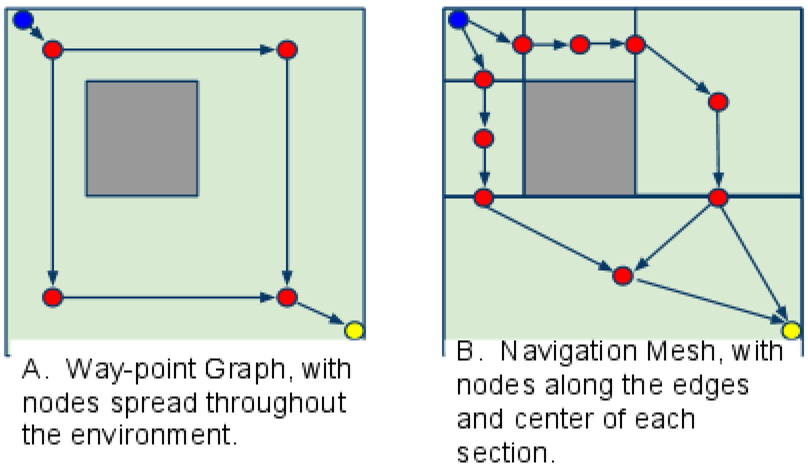

A. Way-point Graph, with nodes spread throughout the environment.
B. Navigation Mesh, with nodes along the edges and center of each section.

Figure 1.5: Way-point Graph and Navigation Mesh.

1.2 Pac-Man

To accurately answer the question posed above *In a game environment what does a realistic artificially intelligent agent require?* a game had to be chosen to work with. The game world had to be sufficiently engaging to require more than a simplistic approach but also allow for ease of demonstrability of the required results. The game chosen to be emulated in the application side of the thesis was Pac-Man. Pac-Man was originally created by Namco in 1980, designed around the simple goal of obtaining the highest score possible. To that end the character of Pac was controlled by a human player and its objective was to eat all of the food pellets and power-up pellets on each level, while avoiding the ghosts wandering the maze. An additional way for the player to increase his or her score was to eat the ghosts while under the influence of the power up pellet. This would also help the player in avoiding the ghosts because, once eaten, a ghost would be out of play for a moment, allowing the player to move around the maze unmolested. During the course of the game the ghosts would increase in speed to make the game more difficult and challenging.

1.3 Original Pac-Man Artificial Intelligence

In the original Pac-Man game the ghosts were the agents whose movement was predetermined and static, whereas in the application for this thesis both Pac and the ghosts are agents controlled by the computer simulation. To that end, in the original Pac-Man game the actions of the ghosts were predictable and deterministic patterns were shown to exist and could be followed to always achieve a victory with the highest score possible. A realistically artificially intelligent agent does not use deterministic patterns in a repetitive fashion as the ghosts originally did, unless specified by a task or behavior such as patrolling a particular zone.

The following will describe the original inadequate level of artificial intelligence found in Pac-Man and describe briefly why it is not realistic. In Pac-Man there were four different colored ghosts and each one had a specified way to interact with the player. That interaction could be loosely defined as a behavior and each ghost had a Japanese name which described that behavior.

There were some basic rules that all of the enemy agents, i.e., the ghosts, within Pac-Man had to observe. While chasing the player, they were not able to move backwards. Within certain areas of the map the ghosts could only move in a designated direction. These rules were sometimes changed when the player had eaten the power-up pellet as is explained below. Additionally during the beginning of the current level after designated time intervals the ghosts would retreat to their home zones. This was done in an effort to give the player some breathing room. More details of this interaction will be mentioned later.

The red ghost was called Oikake which means to chase down or pursue. Renamed Blinky in English, the strategy used by his method of AI was to always follow the player. To achieve this he would compute a path which would decrease the horizontal or the vertical distance between himself and the player. When deciding where to move along that path he would always choose to decrease the greater of the two distances. I.e., if the distance between himself and the player was four vertical blocks and three horizontal blocks, the ghost would choose to decrease the four vertical blocks to three before taking a horizontal path that would decrease the horizontal distance to two. This occasionally would make the ghost seem to make stupid decisions. Figure 1.6 illustrates Blinky's target position as being Pac's position on the map. The number of squares between Blinky and Pac is a difference of two horizontal blocks and three vertical blocks. In this case, using the above stated logic, Blinky would attempt to move downward to decrease the vertical distance as that was the greater of the two distances. However as he would encounter a wall, Blinky would move closer in the horizontal direction.

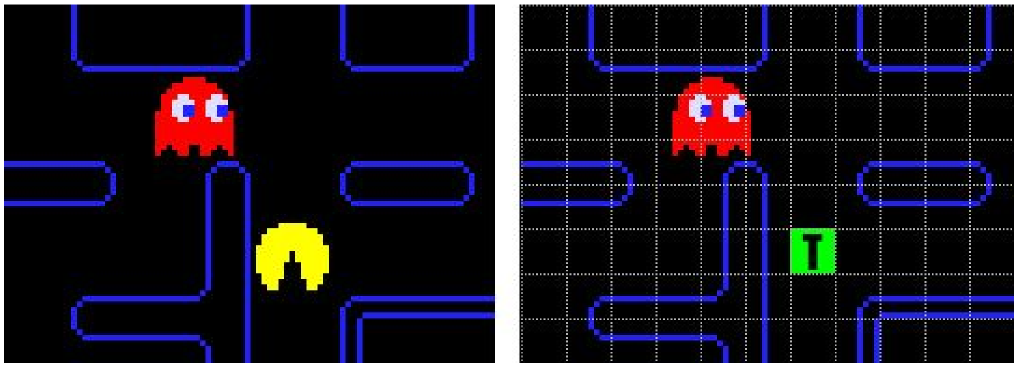


Figure 1.6: Blinky's target position.[2]

Machibuse was the pink ghost in Pac-Man and renamed Pinky in English. His name means to ambush. In keeping with that personality type, he would always have a target position of four blocks ahead of the player. By having a target position of four blocks ahead of the player, Pinky will never actively attack the player by moving to the player's position. Pinky will only be used in a manner such that he has to be avoided but does not move onto the player's position. Figures 1.7 and 1.8 display Pinky's target position as a green square, with each of these target positions being four blocks ahead of the player. Using the same logic as Blinky, the red ghost, he would decrease the greater of the vertical or horizontal distances towards the player. By doing this he would effectively cut off the player in many situations. However this behavior could be easily exploited by the player as well. If the player were to turn towards an obstacle or wall within the maze, then the target position for Pinky would be on the other side of the wall and so, temporarily, he would not be a threat to the player. Fooling Pinky in this manner can be shown within Figure 1.7. In the bottom left corner of the four part diagram, Pinky's target position is shown as being on the other side of a wall from Pac and he is hence fooled into moving incorrectly to attack Pac. This strategy is only applicable when Pinky is already in a position to be fooled, such as shown in Figure 1.7, where he would not actively go around the obstacle to seek Pac.

In addition to being fooled Pinky could be forced out of the player's way. Figure 1.8 shows Pinky being forced out of the player's way, because when the player changes his direction to look at Pinky that effectively moves the target position behind the ghost. One may wonder why Pinky did not just turn around in that situation and move towards the target position. The reasoning for this is because in the game of Pac-Man as mentioned above, the ghosts were not able to reverse direction except under special circumstances, as discussed later. Under this limitation the ghost could be forced out of the player's way in a predictable manner.

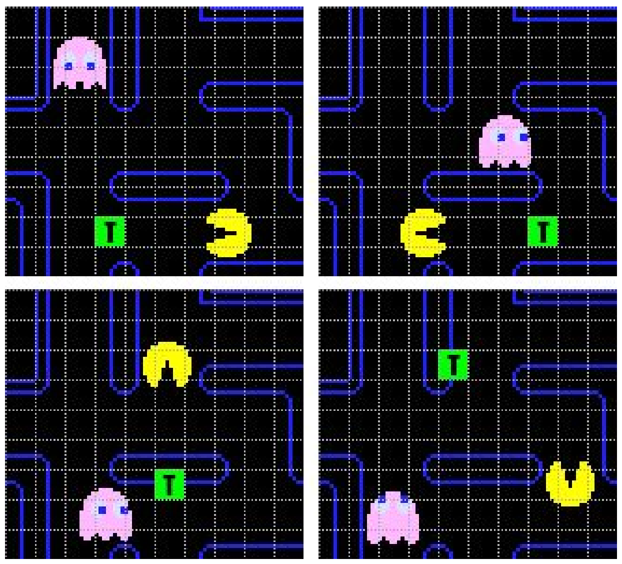


Figure 1.7: Pinky's target positions.

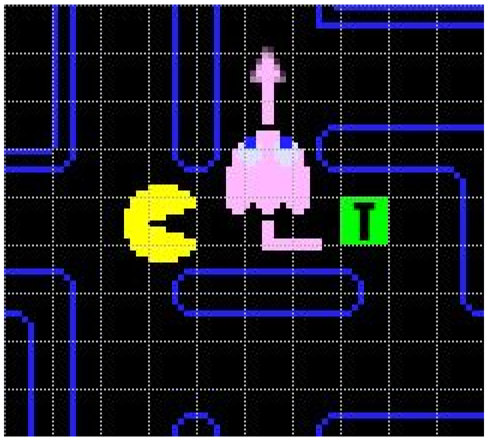


Figure 1.8: Pinky being forced out of the way.

The blue ghost called Kimagure or Inky in English was named so because his behavior was shown to be fickle or moody. As such he would use multiple inputs in order to set the target position which he was to move towards. Inky would take into account the position of the red ghost as well as the position two units in front of Pac. Taking those two points it would draw an imaginary line between them and then extend that line again in the same direction. This new position became Inky's target position. Figure 1.9 shows this accurately. The two points considered there are the red ghost Blinky's position and the point two units in front of the player. These two points create a line. The line is then doubled in length. The end point of the line becomes Inky's target position. This appears to give a more accurate simulation of intelligent behavior the closer Blinky the red ghost is in relation to the player. For example, when Blinky is relatively close to the player in Figure 1.9, Inky will move down towards his target position which is close to where Pac will be emerging and therefore help to trap him.

Conversely, Figure 1.10 displays how his behavior is described as fickle or moody as he moves away from the player because his target position has been set as a distant point. This allows the player to escape.

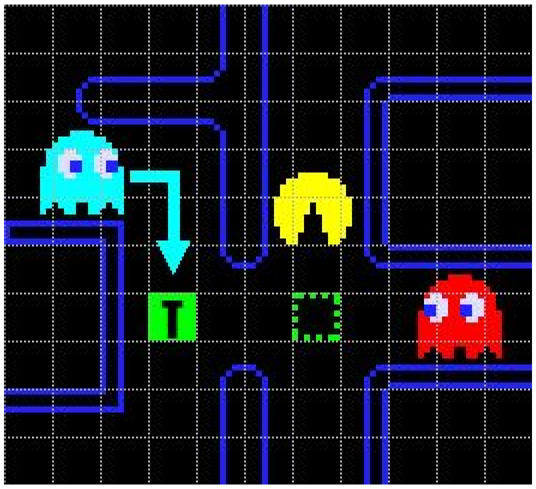


Figure 1.9: Inky's target position.

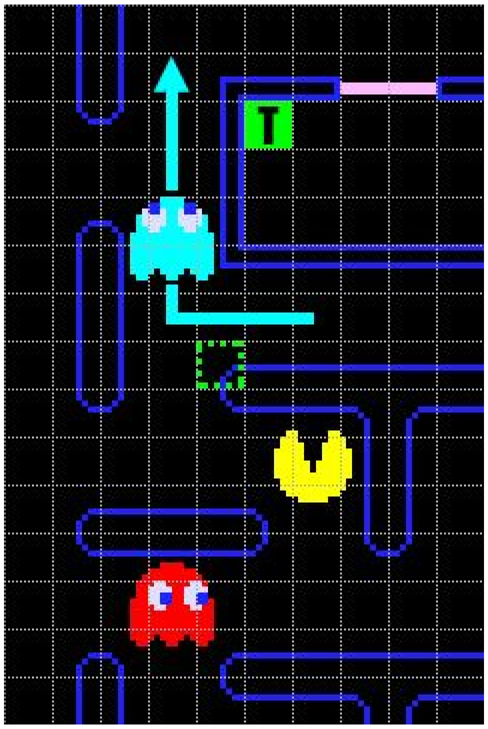


Figure 1.10: Inky making an unintelligent decision.

The final ghost in Pac-Man was orange in coloration and named Otoboke in Japanese or Clyde in English. Meaning 'pretending ignorance' his method of tracking the player was very erratic, having two different modes depending on the distance he was from the player. Being within eight blocks of the player he would react differently than if he was farther away from the player. We will refer to these as the farther mode and the nearer mode. In the farther mode he would move towards the player using the same logic as Blinky, i.e., decreasing the greater of the vertical or horizontal distances. However once Clyde moved to within eight blocks of the player his actions changed and he would immediately change his direction and move to the lower left corner of the map which was considered his "home zone". This would result in the situation shown in Figure 1.11, where Clyde is circling a middle island, attempting to move towards and then away from the player.

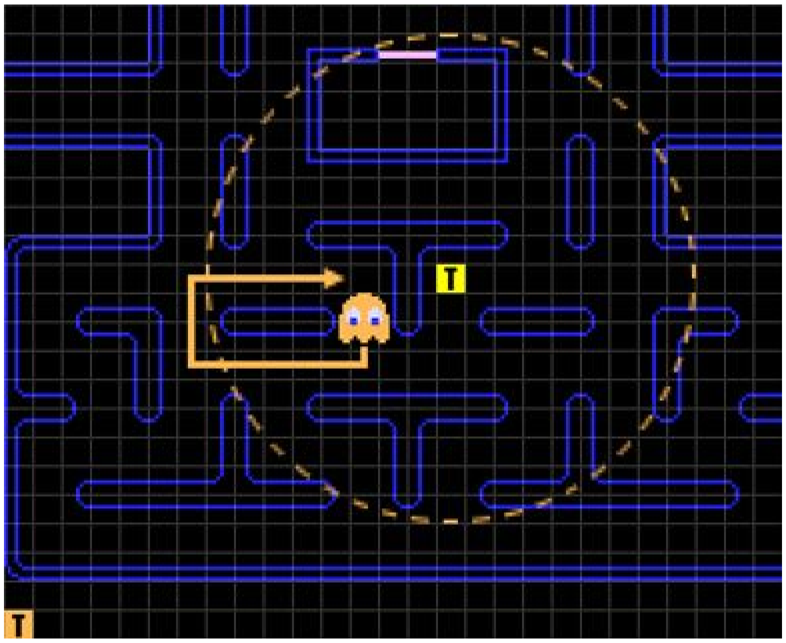


Figure 1.11: Clyde circling his middle island.

As mentioned above these behaviors remained fairly constant throughout the course of the game, the only changes occurring if the player had consumed a power pellet or if in the earlier levels the ghosts were to retreat to their home zones. If the ghosts were to retreat to their home zones at any time, they would move to their designated corner for a moment and then return to chasing the player. As the game level progressed the retreating time was decreased and finally eliminated. As shown in Figure 1.11 the orange T tile is where Clyde is moving towards when going towards his home zone. The T or target position is located in the bottom left corner of the picture and is beneath the game area itself. When moving towards their home zones each of the other ghosts would be moving to a different corner targeted in the same fashion as Clyde is above. This would only happen for the other ghosts, when the ghosts were designated to return to their home zone, which as mentioned above, happened on regularly structured intervals during the beginning of each game level.

The change that occurred when the player ate a power-up pellet was that the ghosts would be allowed to reverse direction for a short period of time. This does not mean they always reversed direction; only that they could do so. Whether or not they did change direction depended on their logic of hunting the player as well as a randomized possibility. In addition to the possibility of reversing direction, the ghosts also moved in a predetermined direction at intersections during this time. When a level was changed or when the player died, a random number was chosen. Upon the consumption of a power pellet by the player that random number was translated to a direction in which the ghost would always move when entering an intersection. If that direction would encounter a wall, then the next direction in sequence was chosen in which the ghosts would move. The sequential order from which the chosen direction was determined was up, left, down, right. This only occurred while the player had consumed a power-up pellet and was able to eat the ghosts.

The techniques briefly mentioned previously in how modern games handle artificially intelligent agents will be discussed in more detail in the following chapter. We will elaborate upon what they are and how they are used in current applications.

Chapter 2: Artificial Intelligence

The term Artificial Intelligence was coined in 1955 by John McCarthy who defines it as: *The science and engineering of making intelligent machines, especially intelligent computer programs.* [3] Merriam Webster defines artificial intelligence as; *The capability of a machine to imitate intelligent human behavior.* According to Meriam-Webster's definition of artificial intelligence, the machine or agent imitating intelligent human behavior is required to have clearly defined attributes in order to succeed within an environment. It should have a way to acquire knowledge of its surroundings, a way to represent that knowledge in order to assess its meaning and gain an understanding about the information, and the ability to plan a course of action based upon the representation of the acquired information. And, finally, it should have a way to achieve the goal or solve the problem placed before it.

2.1 Knowledge Acquisition

Knowledge acquisition in artificial intelligence as it relates to games is handled through multiple techniques. To quote an interview response from an industry veteran and academic[4], Phil Carlisle, when asked about the different types of sensors used for knowledge acquisition, responded, "You query for what you need when you see you need it."[5] Unfortunately that response and perspective seems to falter when faced with constructing a realistic artificially intelligent agent. When the goal is a realistic agent the question should be how to gather accurately the information needed from what was obtained and not simply querying for what was required as that is not always possible. An elaboration of the methods used in the current application and how they are realistically applied will be discussed later. First we look at some of the knowledge acquisition techniques currently in use in the gaming industry. When gathering information some of the techniques used are ray casting, positional annotation, sphere of responsibility, and fields of view.

2.1.1 Ray Casting

Ray casting is a type of knowledge acquisition through which the sensor type used is a ray. The sensor extends outward and returns information about anything that it is required to report to the agent, so that an accurate understanding can be obtained about the environment. A ray cast sensor can be extended in any direction at any angle from the chosen initial start position. The start position is usually the artificially intelligent agent, though it is not required to be. There are numerous benefits and pitfalls associated with such a system. One of the pitfalls is shown in Figure 2.1. Four sensors have been extended from the red colored agent. The top sensor has encountered an obstacle and registers such as denoted by the orange line. The obstacle in the bottom right is not seen due to its position.

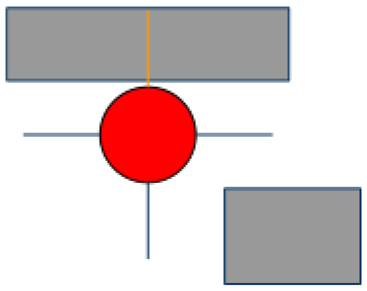


Figure 2.1: Ray Casting for knowledge acquisition.

The pitfall shown in Figure 2.1 can be resolved by creating additional sensors at additional angles from the start position. However the limitation of linear sensors will still cause problems as there will be empty spaces which the rays will not be able to cover and so the sensors will not be able to gather complete and accurate knowledge about the environment. On the other hand, there are benefits to such a system. If the environment was structured so that the agent would only be able to move in the specified directions in which the rays were available then the limitations of the sensors would not be a hindrance, and the reduction of memory used would be a benefit.

2.1.2 Positional Annotation

Positional annotation is a very structured way to collect information. That is because positional annotation is the act of specifically deciding what information to give an agent dependent upon its current position. The developer or designer chooses what information is valid to relay to the agent. Positions are annotated using various methods. Annotations can be implemented by someone annotating the environment in which the agent is working or annotations can be created automatically by the agent with the knowledge it has obtained to this point. If the agent annotates a position in the environment, it could either annotate either the game environment or the agents own individual copy of the environment. The following example will elaborate with greater clarity. In Figure 2.2, there is an intersection which has been annotated by a small blue circle. The circle when encountered by the agent traversing the environment will convey to the agent the information designated. The information could be as simple as *'entered intersection'* or it could be very elaborate in the information given, such as: *'entered intersection; exits right, left, down; goal left'*.

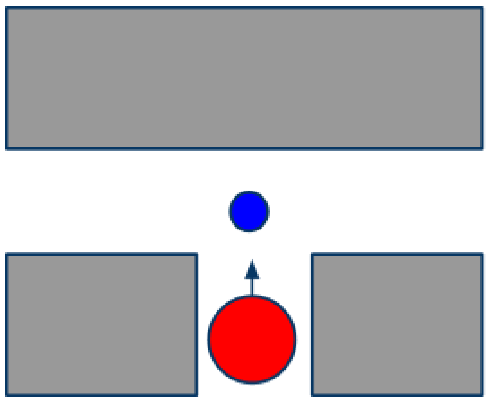


Figure 2.2: Positional Annotation of an intersection.

All types of information gathering have costs and benefits associated with them. In positional annotation the information gathered is strictly limited to what has been annotated. There are no additional methods through which to gain additional information once the position has been marked, unless the position's information is annotated differently at a later time. Unfortunately this method of knowledge acquisition requires the most effort and is the least dynamic of the many different types. It does allow for a rigidity of structure that is sometimes appropriate for the situation or problem addressed. An example of this is in a situation where there will never be any change to the environment. If there are positions within an environment such as an air duct system which will never be changed in size or configuration, then it may be applicable to use this type of knowledge acquisition. Due to its limitations, however it may not be recommended in areas where the structure of the environment would commonly be reconfigured drastically.

2.1.3 Sphere of Responsibility

A third type of knowledge acquisition is sphere of responsibility. This is similar to structure omniscience, described above, as only a subset of all available game information is made available to the agent. When gathering information using this methodology, the agent has a sphere or circle around itself, in which it gathers knowledge about its surroundings. This sphere controls which subset of all information is within the ability of the agent to acquire, giving the agent knowledge of its surroundings, as shown in Figure 1.1. There are no inherent limitations with this type of knowledge acquisition as the agent knows everything in the area surrounding it and can make accurate decisions based upon that information.

2.1.4 Fields of View

The fourth way in which an agent acquires knowledge is by using a subset of the sphere of responsibility called a field of view. A field can be any size which is a subset of the whole. Also called view cones, the fields are angular subsections which point in specified directions. As shown in Figure 2.3, the fields of view do not have to be contiguous.

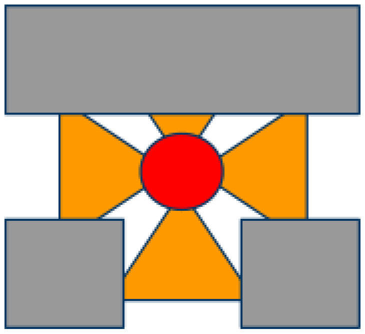


Figure 2.3: Fields of view.

As with ray casting, knowledge acquisition via fields of view will leave gaps in which information will not be gathered. This leads to problems due to not all information being gathered and therefore accurate decisions cannot be computed.

2.2 Knowledge Representation

When knowledge is acquired from sensors it has to be stored so that it can be easily recalled. The manner in which knowledge is structured, stored and accessed is called knowledge representation. Common forms of knowledge representation within the field of artificial intelligence as applied to gaming and simulation are graphs and potential fields.

2.2.1 Graphs

A graph is one of the common ways to represent knowledge because it is easy to have the information structured so that it can be perceived and understood logically. A graph is made up of nodes and with arcs being connections between them. Each node or arc can have information stored within it. When an agent explores its environment, it can store locations it has been as well as the information it has gathered using its sensors. This information can be stored in the form of a node in a graph. Figure 2.4 portrays an image of a graph as well as the information that could be stored within a node. Graphs are also useful because for traversing them, well known and understood graph searching algorithms can be used.

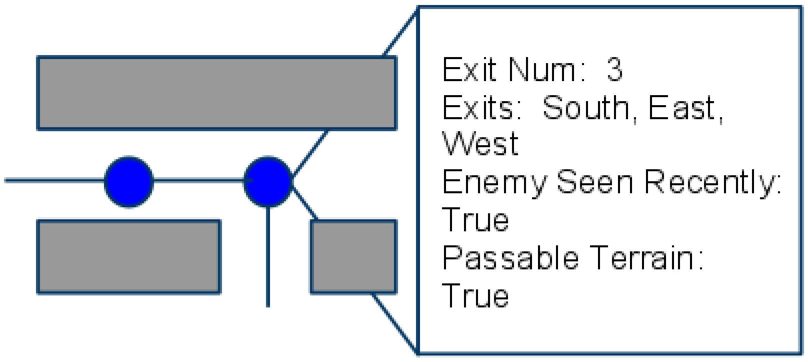


Figure 2.4: Graph node and example information.

2.2.2 Potential Fields within Graphs

Potential fields are a type of knowledge representation where every object within an environment exerts a force upon that environment. The force that is exerted is referred to as a potential and can be either attractive or repulsive. If the force is repellent, then the agent traversing the environment will automatically move away from the object. If the force is an attractive force, then the agent will move toward the object. Potential fields are a way for an environment to be tagged in such a manner that the agents will be able to automatically and realistically cross the environment towards the designated goal. Each node itself has a potential, a node in this case is a position within the environment which the agent can move to. Nodes can be influenced by objects, agents or neighboring nodes within the environment.

When the potential field approach is applied to an environment, under most circumstances, repulsive forces are enough for obstacle avoidance. For obstacle avoidance it would be counter productive in most situations to attract the agent in an opposite direction as opposed to simply having it avoid the obstacle. Even if an attractive field is not actively influencing the environment, there will be an attractive field drawing the agent towards its designated goal. Along the way the agent will have to avoid any repulsive fields before it, which are created by the obstacles in its path. The attractive field towards the goal can be either an active field to draw the agent towards the goal, or merely a repulsive field that is expanding itself behind the agent driving it towards the goal. In either of those two scenarios the agent will actively seek out the designated goal and avoid the dangerous areas of the environment. Figure 2.5 shows environment traversal using potential fields. The agent will move towards the goal and avoid the dangers zones designated by the pink coloration. The blue river in the center of the environment is marked as hazardous, as is the pit directly ahead of the agent.

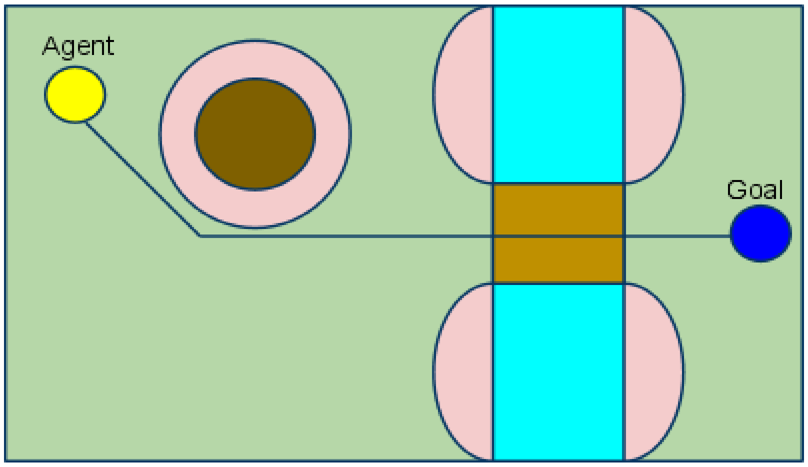


Figure 2.5: Agent traversal avoiding dangerous potential fields.

2.3 Planners

Planners are used in artificial intelligence so that the agent or agents can achieve their desired goals. A plan will consist of a series of actions which will require accurate decisions to be made about the implementation of the actions. To finalize an implementation of an action, the agent bases the decision on the knowledge acquired and how it is represented within memory.

2.3.1 Finite State Machines

As mentioned briefly previously and shown in Figure 1.3, finite state machines are composed of states based upon how the agent is supposed to interact with the environment. The specified states are how each of the agents determines its next series of actions. For example in Figure 2.5, the final achievement would be for the agent to move to the goal position. In order to do this, the agent might have multiple states. An observation state could be used to see if the goal was in the relative locality of the agent.

If it was not this condition could result in a transition to a search state, where the agent would move around in an effort to find the goal. In doing so, if it encountered the danger zone indicated by the trap, the agent would switch to yet another state for obstacle avoidance. Once the obstacle had been avoided then the agent could revert back to the search or observation states. As these states are being cycled through, the agent will be choosing positions to move to in the designated environment. Once these choices were made then it would indicate to the problem solving portion that it wished to move to that position.

2.3.2 Goal Oriented Action Planners

A GOAP system does not replace the need for a FSM [Fu03] but greatly simplifies the required FSM. (Orkin, 2003) GOAP chooses a series of actions that are atomic in their construction which can be used to satisfy a desired goal. The individual actions are created with simplified FSM's and create a graph which can be used to find a path to the desired goal of the system. A single action within the GOAP architecture can be either a single state or an entire FSM.

In a GOAP system the actions are the nodes in the graph and the arcs are what is handled internally so that the plan can be completed through the series of actions chosen. As an example, suppose the agent wishes to harvest some berries. The first action would be to pick up a container to place the berries into. Secondly the agent would have to move to the berry bush. The final action would be to begin harvesting berries. The plan chosen could be varied if, perhaps for a container, the agent used a basket, bucket or hat.

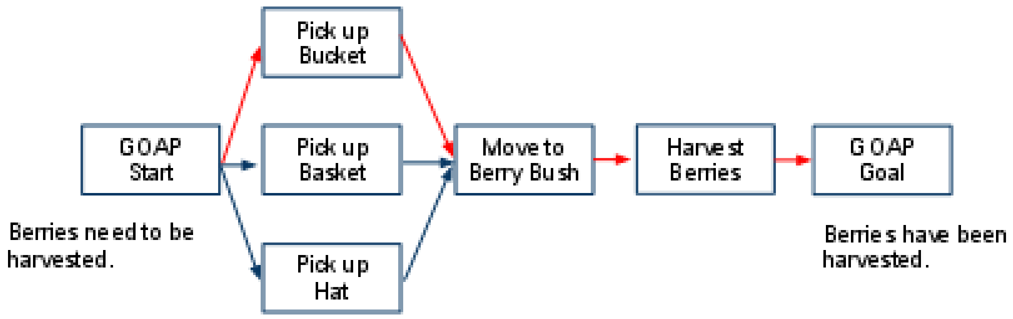


Figure 2.6: GOAP berry harvesting example.

In the aforementioned example of GOAP architecture is shown in Figure 2.6, the actions are shown as the nodes and can have numerous choices at the initial level. Initially there are multiple objects which the agent can pick up to harvest berries. The planner in this case chooses to pick up the bucket to harvest berries. A red arrows designate the path that is chosen by the GOAP planner to achieve the desired goal.

2.3.3 Hierarchical Task Network

HTN's provide an expressive language and framework for representing procedural domain knowledge in structured forms and organizing it. (Nejati, Konik and Kuter, 2009) A specified task is resolved by breaking it down into smaller subtasks which can be completed with knowledge acquired previously during execution of the application. To illustrate an example of an HTN and of breaking down tasks into subtasks within the context of the game of Pac-Man, a task can be designated for Pac to attack a ghost. To achieve the higher level task of attacking a ghost, Pac would first have to find a power pellet. Then he would move towards the power pellet and consume it. After that Pac would find a ghost and move towards it, finally ending by moving on top of the ghost and eating him. Each of these is a sub task that has to be accomplished to complete the over all task of eating the ghost.

Figure 2.7 illustrates the previous example by breaking down the task into smaller sub-tasks. Once the sub-tasks are broken down sufficiently to their component parts, then the resulting leaves of the network are combined into a resulting plan. The red arrows in Figure 2.7 denote this connected plan. Each of the subsequent leaves or actions could be a single state from an FSM.

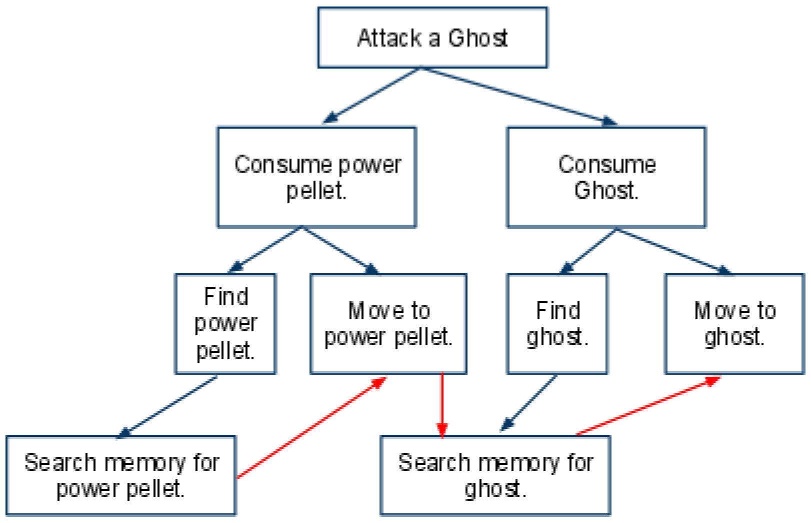


Figure 2.7: HTN of Pac attacking a ghost.

GOAP and HTN's are both ways of combining previously existing FSM methodologies in new ways. Both use object oriented methodologies in an effort to abstract and allow for a more generalized approach to a problem, while retaining the underlying architecture of a FSM. GOAP's actions directly relate to an underlying FSM, whereas an HTN uses a form of parsing to abstract the actions into connected terms forming more complicated actions.

2.4 Problem Solvers

Problem solving is a broad topic and encompasses many different methodologies. Within game development problem solving is commonly restricted to traversing an environment using the shortest path possible between two points.

2.4.1 Environment Traversal

Problem solvers when used to traverse an environment are going to be a type of path-finding algorithm. While there are numerous algorithms used to find a path the most commonly used one in game development is A*. Alex Champandard an industry veteran, put it best when he said, "A* and simplified versions of HPA* are the most common – the reason for this is that it's good enough!"[5] HPA* stands for Hierarchical Path-Finding A*, which is the use of the A* path-finding algorithm in a hierarchical manner, with the hierarchy comprised of different levels of detail along the path.

The generalized A* algorithm is a state-space based searching algorithm which uses a heuristic and is guaranteed to find the shortest path between two points in an application space as long as such a path exists and the heuristic used is admissible. A heuristic is an educated guess of the distance remaining to reach the goal. To be admissible a heuristic cannot overestimate the distance between the current state and end state. The current state in this instance refers to the state that the agent is in within the algorithm itself as it moves along within the environment. There is an application or environmental state which is the state with which the application is currently running. That includes all of the positions of the agents and pellets at a specific time. The end state refers to the state of the application where the agent has moved to the designated end point. Each possible choice of movement by the agent inside the application state is a new algorithmic state within the A* algorithm.

For example, if an agent is trying to find the shortest path between itself and an object on the other side of the environment, it will take a look at its initial state. The initial state is the state that the application is in at the moment the algorithm runs. Every direction the agent is capable of moving within the environment designates a new algorithmic state. As the algorithm chooses an algorithmic state based upon a possible choice, it will judge which possible change in the algorithmic state will be to the most beneficial. The formula used in the A* algorithm to estimate the total distance f(x) from the start position to the goal state is shown below in Figure 2.8. This estimate is used to determine which of the next algorithmic states is the best choice. H(x) is the heuristic value of what remains left to be traveled from the current algorithmic state under consideration by the algorithm to the goal state, with g(x) being the distance which has been traveled in the current algorithmic state.

$$f(x) = g(x) + h(x)$$

Figure 2.8: The formula used by the A* algorithm.

A popularly used heuristic when using A* is Manhattan Distance, shown in Figure 2.9. The name is taken from the similarity in the heuristic's computed value to traveling along city blocks. The heuristic is calculated by taking the current state and the goal state and finding the difference between the two. The difference between the X and Y positions is added together and forms the value for the heuristic.

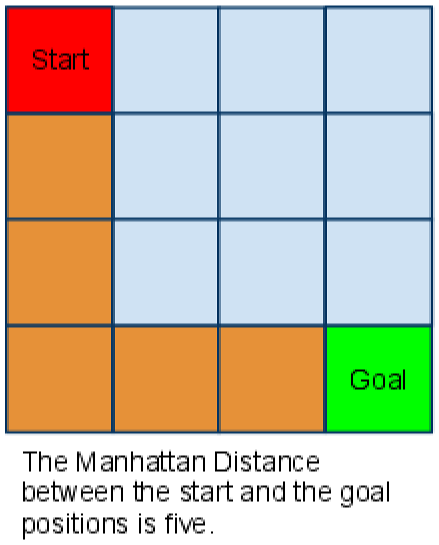

Start
Goal
The Manhattan Distance between the start and the goal positions is five.

Figure 2.9: Manhattan Distance Calculation.

Each iteration of the algorithm updates the values of known and estimated distances in the application state space, using the application state with the lowest known added to the current estimated distance gives f(x) as its next point in the search. As each value of f(x) found in this manner is compared to find the lowest value. The algorithm will continue to find the f(x) with the lowest distance by expanding points within the application space until the minimum f(x) is the goal state.

Using the methods described in this chapter as a basis for an artificial agent, the following chapter will explain what needs to be changed and expanded upon, so that the agent exhibits realistic behavior and makes intelligently realistic decisions.

Chapter 3: Contribution and Extension

The original question which we were trying to solve *In a game environment what does a realistic artificially intelligent agent require?* has a key concept in it, realism. Of the above methodologies, which ones had the greatest basis in realism and how could the methodologies be extended to allow for a greater degree of realism? When writing the application for this thesis, that was the governing constraint. The application itself is a recreation of the Pac-Man game. In it both Pac and the ghost's are agents within the game. Each agent has goals that it must complete, these goals are completed using specific actions. Actions as well as the goals are influenced and determined with the use of sensors. The agent's use of sensors needs to be realistic in order for the agent to realistically interact with the environment. The following extensions are additions to specific concepts which were previously mentioned. Combined together the extended concepts will create an intelligent agent which is more realistic in its interactions with both other agents and the environment.

3.1 Implementation of Knowledge Acquisition

When creating a realistically intelligent agent, the way knowledge is acquired has to be as realistic as possible within the environmental constraints. Pac-Man is a very linearly constrained environment. This means that each move is made in a linear fashion. Either north, south, east or west will be chosen as the direction to move;

diagonal directions are not valid moves within the environment given. Therefore in order to accurately gather information about the environment the constraints of using ray casting for knowledge acquisition seemed as though they would work.

3.1.1 Ray Casting for Knowledge Acquisition

Once that choice was made it was implemented as shown below. Unfortunately as displayed in Figure 3.1, although the environment might be structured for using ray-casting for knowledge acquisition, it does not realistically gather the information presented. In Figure 3.1, the yellow dot represents the Pac agent, and the red dot represents the ghost agent. The green bar is the forward vision seen by Pac and the pink bar represents the previously seen information. Shown in Figure 3.1 is the Pac agent being unable to see the ghost agent, although clearly the ghost should be seen. The code for how this was implemented is shown in Appendix A. The pseudo code for ray casting perception is shown in Figure 3.2.

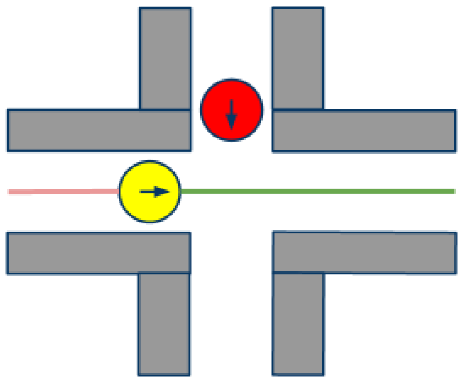


Figure 3.1: Ray-casting perception.

```
while( has ! faced in all directions)
{
    while(camera position ! = wall)
    {
        memory=seen graph node at position;
        position incremented in direction facing;
    }
face in new direction;
}
```
Figure 3.2: Ray-casting perception pseudocode.

3.1.2 Field of View Implementation

As we have seen, ray casting perception does not provide a great deal of realism. An approach with a greater degree of realism is one that mimics a human in terms of visual perception. Human perception when using both eyes, is called binocular perception. Each eye sees a slightly different field of view. When the images are combined and merged together a human will have a unified field of view with a central angle of about 140 degrees. On either side of that field of view there will be approximately twenty degrees that each eye sees as peripheral vision. When creating a form of knowledge acquisition for the application it was chosen to mimic the unified field of view demonstrated in human vision. Figure 3.3 displays the unified field of view as well as both peripheral fields.

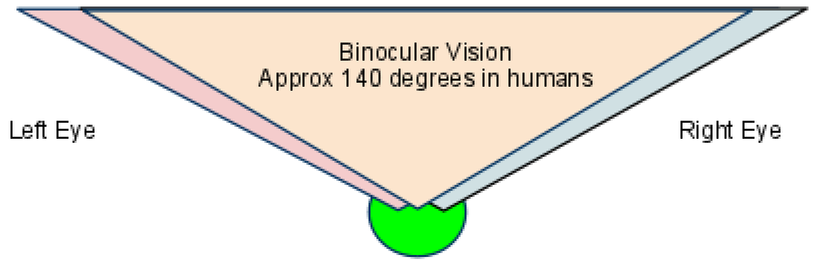


Figure 3.3: Binocular Vision display with the overlayed left and right eye fields of view.

Using the same layout as the example of Figure 3.3, Figure 3.4 shows the new field of view implementation within the current application. The view is consistent and correctly mimic's a human's perception of the environment. As such the agent's view within the environment has a single field of view that has an angular size of 140 degrees.

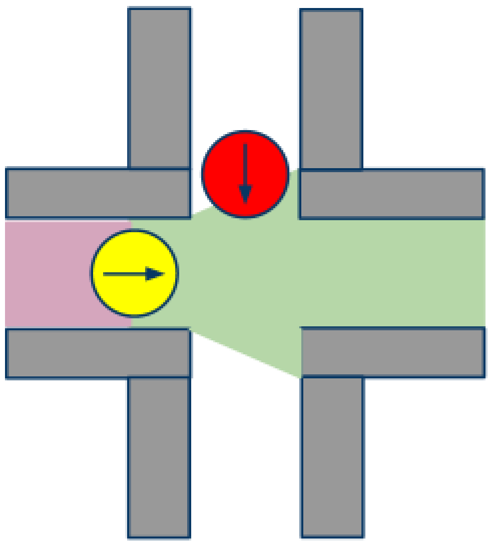


Figure 3.4: Pac agent's example with field of view.

In order to calculate the field of view the mathematical cosine function was used. The distance ahead of the agent, which the agent is allowed to see is pre-defined. This is because realistically humans cannot see details at great distances. Once the distance was set determining how far in front of the agent the agent is able to see, then the angle of view is split in half and the cosine is calculated as shown in Figure 3.5. The interior of the two formed triangles is then swept and all relevant information is gathered and stored to be analyzed. Figure 3.5 shows a hypotenuse value of seventy-three. That value means that for seventy-three units at an angle of seventy degrees from the agents position both above and below the agents center ray, the information is retained in memory. The determined value of seventy-three is calculated from the determined distance over which the agent is able to accurately view. The value will not change unless the distance viewed is changed.

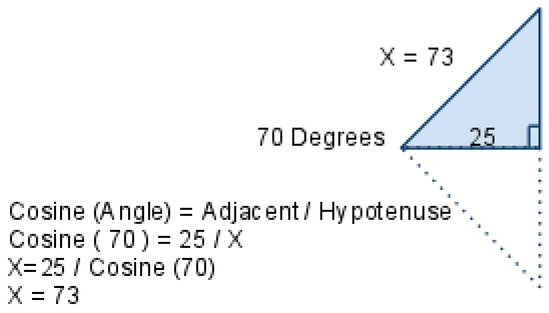


Figure 3.5: Calculating a sample cosine.

3.2 Implementation of Knowledge Representation

To represent knowledge accurately and realistically, as discussed previously, one can use a potential field within a graph structure. A graph allows nodes to be added easily to represent new information. In this case a node would be designated as a new position within the environment which the agent was exploring. The arcs between the nodes would be the connections to other nodes which have been entered into the graph through knowledge acquisition. A potential field will help the agent to represent the knowledge in such a way that it can be used to realistically traverse an environment and avoid dangerous locations, as will be elaborated upon in more detail later.

As a new position is seen, using the field of view method described above, it is added to a memory graph of the environment. The memory graph in the application is stored within three data structures, each one is a two dimensional array. The first data structure retains what information is seen at the specified node. The second data structure contains the arc weights between the nodes themselves. The final data structure contains the connection information as to which direction the nodes are connected to each other. Each of the data structures are determined based upon the size of the initial environment. When adding a node to the graph, information is updated to each of these arrays based upon what knowledge has been acquired.

Figure 3.6 gives an example of how the connection information is assigned to the specific data structure. The allowable directions of movement within the application are the cardinal directions, north, south, east and west. Each of these directions is assigned a numerical weight which is equal to a power of two. The reason numbers with a power of two were chosen was so that bit wise operators could be used as a way to speed up the computation. Each direction that is open or valid to be moved into is added to a sum total. The total can easily be referenced to find if there is a connection in a specified direction, by using a bit wise and operation.

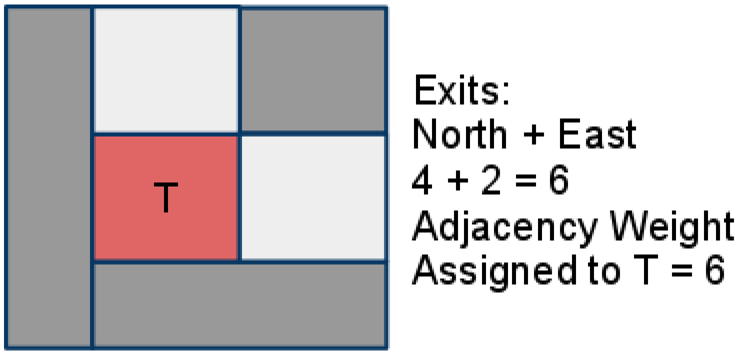
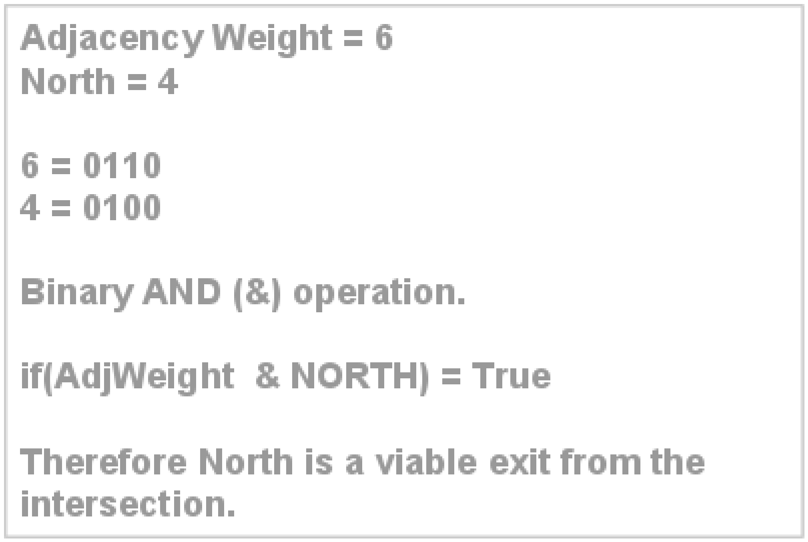


Figure 3.6: Nodal connection weight assignment.

The target position in Figure 3.6 is a corner, the possible exits of north and east are shown. If the agent was at the target position T and wished to know what exits were free then it would do a bitwise and operation with the desired direction and if the value was true, then the agent would know that it would be able to move in that direction. An example of the calculation is shown in Figure 3.7.


Adjacency Weight = 6
North = 4

6 = 0110
4 = 0100

Binary AND (&) operation.

if(AdjWeight & NORTH) = True

Therefore North is a viable exit from the intersection.

3.7: Example of the nodal connection calculation.

The second data structure associated with the graph is used for the arc weights. The arc weights relate to potential fields and the threat assessment of each node within the environment. The agent will progress moving around the environment in an effort to achieve its specified goal. If an individual agent was to spot an agent or agents who are threatening to them (in the case of the Pac agent this would be if he sees a ghost agent) the position where the agent was spotted as well as the area around that position will increase its threat value. This will help the agent to avoid that area, because in the area of greater threat will be a repulsive potential field. The potential threat field will decrease and weaken the farther it gets away from the threatening object much in the same way a threat is perceived realistically. The closer the antagonizing agent, the greater the threat.

An example being, if a human were to spot a bear at a distance, it would not be much of a threat. However if the same human walked into a cave and saw a bear, the bear would be a much greater threat. Figure 3.8 shows an example of the potential threat fields. The red circle indicates the conflicting agent. The colors of the nodes in the background represent the threat value associated with the node. The farther away the Pac agent who is denoted in yellow, is from the conflicting agent, the less of a threat the conflicting agent poses to him. When using an ordinary repulsive potential field there is a chance that the agent will move away from the field but in an incorrectly assessed direction, merely because it is towards a lesser threatening area. In the application this is avoided because the agents have goals which they work to complete. These goals when used in conjunction with repulsive fields help to guide the agent into making the correctly perceived decision.

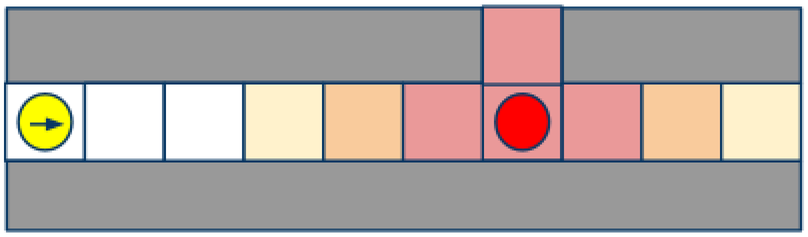


Figure 3.8: Potential threat field.

3.3 Implementation of a FSM

Referencing what was done prominently in industry and academia, the primary decision maker used was a finite state machine. The FSM was extended, however, to use the additional information gathered from the realistic knowledge acquisition system. Each agent type has core behaviors that are common and shared between both types of agents, as well as behaviors specific to each agent type. These behaviors are encapsulated within individual states. Core behaviors for the agents are rather simple in nature, exploration and observing. The FSMs for this are two-state machines, with the states being either exploration or observation. Two-state machines do not exhibit very complex behavior though they do give the impression of such by always actively seeking out an unvisited zone within the environment. The two-state FSMs used for the core behaviors each agent possesses, are expanded upon as the individual agent types are created and have more complex behaviors are associated with them.

Two-state machines which are referenced as singular actions that can be combined to form higher level behaviors, are similar to the GOAP architecture which uses a series of actions traversed across to create a plan. The expansion of the FSM is done by adding states and behaviors that are individual to each agent. The Pac agent has multiple states in addition to simple observation and exploration. These additional states are used to govern its actions within the goals assigned to the Pac agent within the environment and will be elaborated upon later.

In addition to adding states within the FSMs of the application for the agents, sometimes entirely new states are created as the actions for those states are more complicated for a specified task. By reorganizing the core states into sub-states which are subjectively used as needed by the higher level states for each individual agent type, the structure changes its complexity and shares characteristics with an HTN as well.

In my application the two-state FSM is both added to and expanded hierarchically as needed to form individual agent behaviors. Both of these characteristics of hierarchical expansion and state addition are shared individually by HTN's and the GOAP architecture. By combining them, a more fluid and realistic agent is constructed.

3.3.1 Pac's Planner

The core behaviors for each type of agent are inherited and built upon to form the next level of behaviors for the agents. Pac's agent needs to have additional behaviors added in order to accomplish the task presented to him, which is to eat all of the pellets and survive. Due to the additional requirements, Pac's FSM needs three additional states. The states that were added to the FSM are an attack state, a feed state and a flee state. A fourth state, the search state is included in Pac's FSM which inherits the previously mentioned two-state FSM. These behaviors are essential to Pac in the environment for the game. Pac will need to search and explore in order to find food pellets which he will consume. If he spies a ghost, depending upon the threat level the ghost represents, Pac may choose to flee. Or if he remembers where a power up pellet is, he may choose to consume that and attack the ghost seen.

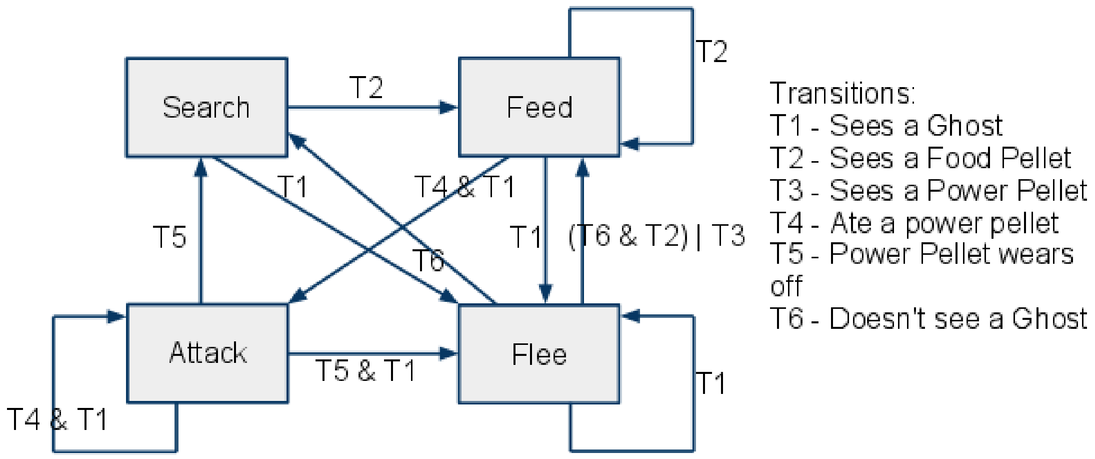


Figure 3.9: Pac agent's FSM.

Figure 3.9 is the diagram of the FSM used for Pac within the game. Shown with the arrows are the conditions that are needed to transition from one state to another. For example Pac will transition from the search state to the feed state if, while in the search state he sees a food pellet; but he will move to the flee state if it see's a ghost.

The planner for the Pac agent is not simply a finite state machine, that decides what state the agent is in and what goal it is pursuing at the time. To plan where to move towards additional factors come into play. If perhaps, as is the case quite often Pac sees food pellets in more than one place, he will use the problem solver method mentioned in section 3.4 to locate the nearest one and move towards it to consume it. The problem solver takes into account the potential threat fields of the ghosts in an effort to provide the shortest and safest path possible for the Pac agent. Once the shortest and safest path has been plotted to the goal object that the planner has decided upon, then the Pac agent will move towards that location. As this information processing is handled in real time, those goal positions may change if additional factors change the environment or influence the agent in some way.

3.3.2 Ghost's Planner

The ghost agent is not required to feed on the food pellets, its only goal is to find and attack the Pac agent. Its state machine only has three states; search, attack and flee. Each of the states represents a behavior that the ghost exhibits over the course of the game. Unlike in the original Pac-Man's artificial intelligence, the ghost agents do not have a specified way in which they interact with the environment. Instead they behave in a more realistic fashion. They hunt down and attack the Pac agent only fleeing if they are in danger of being attacked themselves when Pac has consumed a power pellet. Figure 3.10 displays the FSM used for the ghost agents.

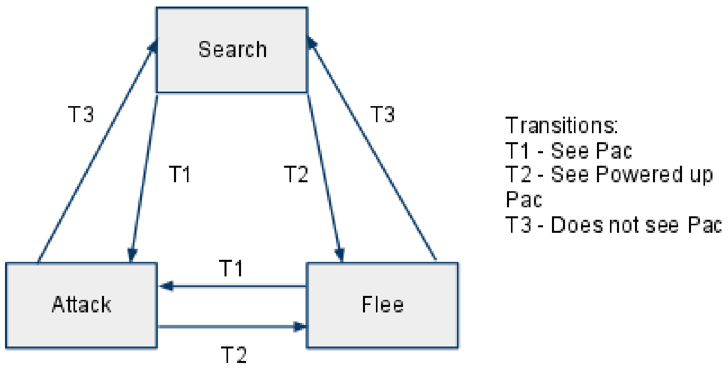


Figure 3.10: Ghost agent's FSM.

For the ghost, planning the next position to go is fairly straight forward. If Pac is seen the ghost will move to that position by the shortest path possible. If for some reason the ghost has seen the Pac agent at an angle and cannot move directly to that position, it will back track along the path which the algorithm has found, to the closest point to which the agent is capable of moving to. The agent will move to that point, hoping perhaps to find the remainder of the path along the way, or perhaps seeing Pac at a new location to which they can move. This is similar to what would be done when pursuing an object or opponent realistically within an unknown environment.

3.4 Problem Solving

The problem to solve as mentioned above is the shortest path problem. Once solved the solution plays a key part in the decision making process to decide where to go. Much of academia and the game developing community use some version of A*, for the application the agents of Pac and the ghosts it was chosen to use A* as well. The algorithm works well and is guaranteed to find a solution if one exists. The version of A* used is the same as mentioned previously, with the heuristic used being Manhattan Distance. Both types of agents use this algorithm to find the shortest path.

A* plays an important role in the decision process of choosing where to move next. The way this is handled is based upon the current goal the agent is actively pursuing, which may create a need to find the shortest path to a location. For example, if the ghost agent was chasing the Pac agent, then A* would be used to find the shortest path from the position of the ghost to the position of Pac. This takes into account certain issues such as whether or not the area is completely explored all the way to the goal position. If it were not fully explored the algorithm would move to the ghost agent to closest point to the Pac agent that was explored.

Pac on the other hand makes a much greater use of this algorithm in the decision making process. While in the feed state Pac will likely have a number of different options on where to move in order to eat a food pellet. In this instance A* will find the shortest path to each of Pac's possible targets. By finding the shortest path to each of these individual target pellets, A* can relay that information to the Pac agent to make a decision based upon how close it is to a designated food pellet. That information will be used to decide which food pellet to proceed towards, as the Pac agent is biased to move towards the closest food pellet which is safe for it to consume.

Although the individual artificial intelligence techniques have a greater degree of realism once they have been expanded, they still need to be combined together to form an entirely realistic intelligent agent. The next chapter explains how all the pieces go together to create a realistic artificially intelligent agent.

Chapter 4: Combining the Parts

Each of the parts and segments of the program can be taken as smaller pieces or can be combined. As small parts, they each can work individually and solve small subsets of the given problems, but it is when they are combined that a semblance of a higher level agent begins to emerge.

In Chapter Three the parts of the application were outlined as to how they had expanded upon or extrapolated from previously created concepts and applications to extend the agents to have a greater degree of realism when interacting with the environment. Creating a higher level agent requires combining all of the aforementioned parts.

The program itself is sequential. However there is multi-threading for each individual agent, so that the agents interactions are not static and cyclical; they can each be planning and moving about simultaneously. Each agent is controlled by its brain, the brain handles all of the separate functions that the agent exhibits. The brain calls individual functions to sense, plan and finally execute the decision of where to move about within the environment to achieve its objective. Shown in Figure 4.1 is the overall program architecture, including how the agent brain is constructed.

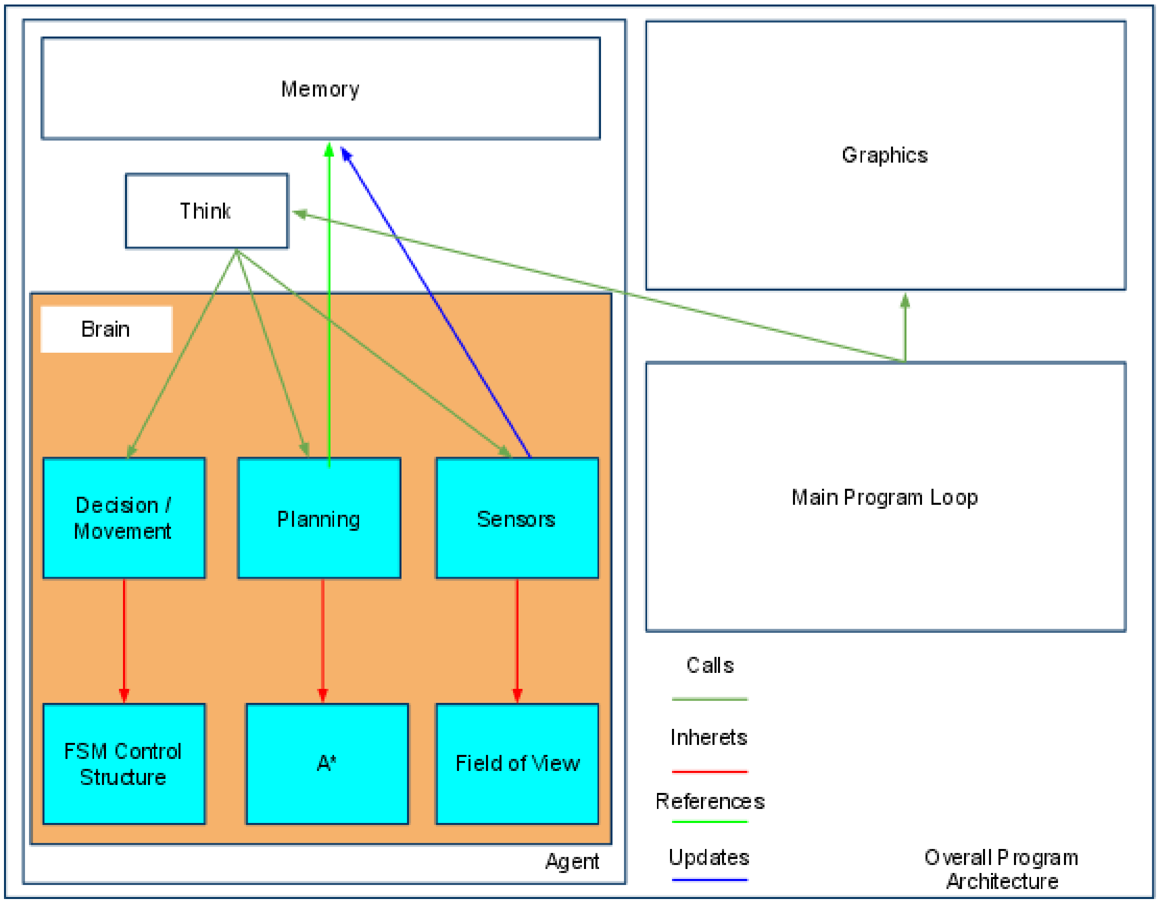


Figure 4.1: Overall program architecture.

Each agent type then has additional features as mentioned in regards to how it solves problems and interacts within the environment. They each use the architecture which is defined in Figure 4.1. However in the derived agent types certain functions are written differently. Using object oriented design, these features and behavior types can be easily added or modified without significant change to more than one section of the code, making the brain very modular in nature.

The main program is a very simple loop. The agents will call the think function which will in turn call the other required functions. Each agent calls its own individualized thinking function which is defined by the agent type. This loop is shown within Figure 4.2 and shows specifically how the agents, using multi-threading are all interacting with the environment simultaneously. The key component when doing this is making sure that the game environment is current as multiple agents are referencing the information stored within it.

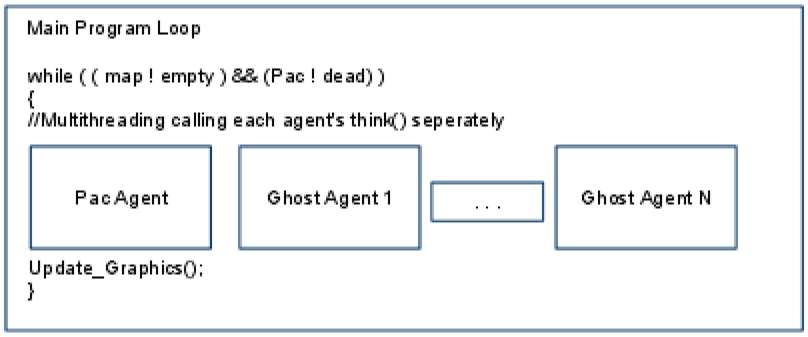
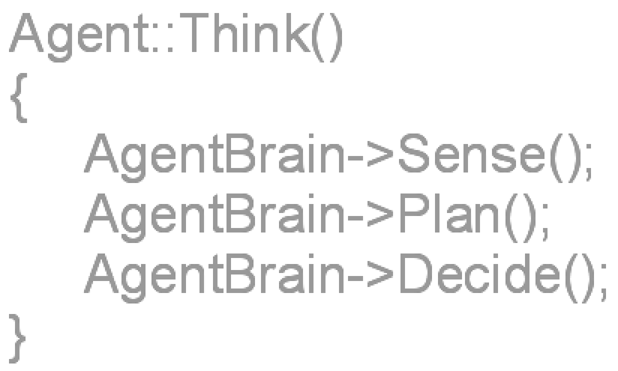

Main Program Loop
while ((map ! empty) && (Pac ! dead))
{
//Multithreading calling each agent's think() seperately
Pac Agent
Ghost Agent 1
. . .
Ghost Agent N
Update_Graphics();
}

Figure 4.2: Main program loop.

Inside the brain function for each agent, the individual function calls mentioned above are handled, in the order stated below. Figure 4.3 displays the agent's think function and the order in which the brain's functions are referenced. Initially the agent will have to sense the environment to see what has changed and update any needed information, such as possibly seeing a hostile agent. Once that information has been gathered the agent will plan how to achieve its designated objective and then execute that decision. This type of architecture is known as SPA or Sense Plan Act Architecture. It is used as a paradigm to break down the control structure into manageable functions which are independent of each other. (Mataric, 2007)



```
Agent::Think()
{
    AgentBrain->Sense();
    AgentBrain->Plan();
    AgentBrain->Decide();
}
```

Figure 4.3: Think function.

To acquire knowledge about its surroundings an agent uses the sense function as described in Chapter Three. The field of view is established and scans the environment to update the knowledge representation the agent has. It does this by creating a field of view in front of the agent. The functions needed to create the field of view as well as update the agent's knowledge representation or memory can be found within Appendix B.

Deciding what direction the agent should look towards is governed by what direction the agent is facing. Under normal circumstances the agents do not look behind themselves when they are moving towards their objective. This changes however if the agent is currently fleeing from a hostile agent. In this scenario the fleeing agent would become worried and look behind itself constantly to update the position of the agent pursuing it, until such a time as it does not see the hostile agent behind it anymore. This type of action invokes a parallel to realistically exhibited behavior when being pursued.

Once the knowledge of the surrounding environment is obtained the agent will use its goals to obtain a plan for action. The goals themselves vary between the different agent types. Pac's agent's goals would be to find the safest and nearest food pellet to consume and move towards it. The plan of action is decided by the next function accessed by the brain, the planning function. This function takes the required goals to be met and finds the next most beneficial step to take in that direction. Continuing with the Pac agent's example, the goal would be to consume all of the food pellets. To complete this, the agent has to find the nearest pellet to consume. It does this by searching its memory for available food pellets. Once it has found a pellet it will find the length of the path from itself to the pellet. After comparing all of the distances from itself to various pellets the agent will choose the pellet that is the safest to move towards. The pellet is chosen to be the safest using the A* path finding algorithm mentioned previously in conjunction with the weighted memory maps. The maps are weighted and updated as various threats in the environment are encountered. Each threat will influence the agents memory of the environment, and the path finding algorithm will take that into account as it finds the lowest weighted path to its desired pellet. Appendix C has the code for the decision making process.

These decisions being made can be influenced by external circumstances as well. For example, if the Pac agent has seen a ghost it will transition to a different state and therefore its decisions will be influenced and possibly changed, exhibiting a different type of behavior. The same is true if the ghost agent has seen the Pac agent and begins to hunt down the Pac agent to attack.

Once the decisions and choices of where to move have been made, then the brain will move the agent in the designated direction towards the designated goal. As the move is completed the brain starts the process over again and begins sensing the environment for any changes or updates that may possibly have occurred, thus updating the representation of knowledge within each individual agent's memory.

As the program runs to completion, the ghost agents inevitably win if there is more than one. With a single ghost, the Pac agent is continuously chased and is rarely ever caught as in most cases the Pac agent can avoid a single ghost. The exception being if the Pac agent and a ghost agent come into contact with each other around a corner and move simultaneously into the same position. In the case of multiple ghost agents, then the Pac agent is fairly easily cornered and trapped as there are now multiple agents to avoid and Pac cannot avoid multiple agents acting and thinking realistically when working together.

Chapter 5: Conclusion

In order to create an artificial agent that is realistic within a game environment the knowledge acquisition and representation needs to be as realistic as possible. The more realistic the lowest levels of the agent are, the more believable the highest level behaviors will be. Realistic intelligent agents within games have applications outside of games as well. There are many parallels that can be drawn between a realistic agent within a game environment and creating a realistic agent within the real world.

Appendix A:

//This function determines where a wall or unit is.

```
void Unit::LookWaU(direc dest, sees &thing, COORD &thingloc)//Look Wall and Unit
{
    int a;
    switch(dest)
    {
        case north://y-a
        {
            a=1;
         while(
Unit::map[Pos.Y-a][Pos.X]!='#'||Unit::map[Pos.Y-a][Pos.X]!='G'||Unit::map[Pos.Y-a][Pos.X]!='P')
            {
                    RememberPosition(dest,a);
                    if(Unit::map[Pos.Y-a][Pos.X]=='#')
                            {thing=wall; thingloc.Y=Pos.Y-a; thingloc.X=Pos.X;break;}
                    else if (Unit::map[Pos.Y-a][Pos.X]=='G'||Unit::map[Pos.Y-a][Pos.X]=='P')
                            {thing=unit; thingloc.Y=Pos.Y-a; thingloc.X=Pos.X;break;}
                    a++;
            }
            break;
        }
        case south://y+a
        {
            a=1;

while(Unit::map[Pos.Y+a][Pos.X]!='#'||Unit::map[Pos.Y+a][Pos.X]!='G'||Unit::map[Pos.Y+a][Pos.X]!='P')
            {
                    RememberPosition(dest,a);
                    if(Unit::map[Pos.Y+a][Pos.X]=='#')
                            {thing=wall; thingloc.Y=Pos.Y+a; thingloc.X=Pos.X;break;}
                    else if (Unit::map[Pos.Y+a][Pos.X]=='G'||Unit::map[Pos.Y+a][Pos.X]=='P')
                            {thing=unit; thingloc.Y=Pos.Y+a; thingloc.X=Pos.X;break;}
                    a++;
            }
            break;
        }
        case east://x-a
        {
            a=1;

while(Unit::map[Pos.Y][Pos.X-a]!='#'||Unit::map[Pos.Y][Pos.X-a]!='G'||Unit::map[Pos.Y][Pos.X-a]!='P')
            {
                    RememberPosition(dest,a);
                    if(Unit::map[Pos.Y][Pos.X-a]=='#')
                            {thing=wall; thingloc.Y=Pos.Y; thingloc.X=Pos.X-a;break;}
                    else if (Unit::map[Pos.Y][Pos.X-a]=='G'||Unit::map[Pos.Y][Pos.X-a]=='P')
                            {thing=unit; thingloc.Y=Pos.Y; thingloc.X=Pos.X-a;break;}
                    a++;
            }
            break;
        }
        case west://x+a
        {
            a=1;

while(Unit::map[Pos.Y][Pos.X+a]!='#'||Unit::map[Pos.Y][Pos.X+a]!='G'||Unit::map[Pos.Y][Pos.X+a]!='P')
            {
                    RememberPosition(dest,a);
                    if(Unit::map[Pos.Y][Pos.X+a]=='#')
                            {thing=wall; thingloc.Y=Pos.Y; thingloc.X=Pos.X+a;break;}
                    else if (Unit::map[Pos.Y][Pos.X+a]=='G'||Unit::map[Pos.Y][Pos.X+a]=='P')
                            {thing=unit; thingloc.Y=Pos.Y; thingloc.X=Pos.X+a;break;}
                    a++;
            }
            break;
        }
    }
}
```

Appendix B:

```
void vision::Look(const Map *aMapName, std::vector<PT2D> &aPointList)
{
    double tFovL, tFovR;
    double tSightAngle; //angle between base and sight vectors
    /*cheating by using a point*/
    QPointF tDiff;
    tDiff = (_SenseDirecVect.End() - _SenseDirecVect.Origin());


    VECT2D tBaseV(0,0,1,0);
    /* cheat by centering sight vector on origin.  this makes the calculations infinitely
       easier. after doing calculations, the vector is moved back to the current point
    */
    VECT2D tSightV(tBaseV.Origin(),tDiff);


    double tSVMag = _SenseDirecVect.Magnitude();
    /* since we centered sight vector at origin, baseVmag is always 1.  no need to calc
       baseVmag, and no need to include it in tSightAngle calculations
    */


    tSightAngle = acos((tSightV.End().x()/**tBaseV.End().x+ tSightV.End().y*tBaseV.End().y*/) /
                       (tSVMag/**tBaseVmag*/)) * (180/M_PI);


    /* if the view vector is in the 3rd or 4th quadrant, adjust the angle accordingly
       this is required because of the return value of arccos
    */
    if( tSightV.End().y()>0 )
    {
        tSightAngle = 360 - tSightAngle;
    }


    tFovR = -_FoVAngle/2; //right view boundary
    tFovL = _FoVAngle/2;  //left view boundary


    /*normalize tSightV*/
    if(tSVMag != 1)
    {
        tSightV.Normalize();
    }


    double tDegrees = 0;
    double tAnglePrecision = 0.1;
    VECT2D tTestV(0,0, 0,0);


    //sweep from the left to the right, incrementing by the angle of precision
    for(tDegrees = tFovL;
        tDegrees >= tFovR;
        tDegrees -= tAnglePrecision)
    {
        //centered sight vector, rotated to current test angle
        tTestV.End().setX(cos( tDegrees*(M_PI/180) ) * tDiff.x() - sin(tDegrees*(M_PI/180)) * tDiff.y()
);
        tTestV.End().setY(sin( tDegrees*(M_PI/180) ) * tDiff.x() + cos(tDegrees*(M_PI/180)) * tDiff.y()
);


        VECT2D tV = tTestV;
        double tGridPrecision = 0.1;
        PT2D tTstPt;


        //sweep from closest position to furthest, exiting if the view is blocked
        //otherwise storing what is seen
        for(double i = 0;
            i <= _MaxViewDist;
            i += tGridPrecision)
        {   /*scale magnitude of test vector*/
            tV.End() = (tTestV.End() * i);
            /*compute from origin, then shift to current position for testing*/
            tTstPt.setX( roundI( _SenseDirecVect.Origin().x()+tV.End().x()) );
            tTstPt.setY( roundI( _SenseDirecVect.Origin().y()+tV.End().y()) );


            /*bounds checking*/
            int tSizeX, tSizeY;
            tSizeX = const_cast<Map*>(aMapName)->DimensionX();
            tSizeY = const_cast<Map*>(aMapName)->DimensionY();


            if(tTstPt.x()<0 || tTstPt.y()<0 || tTstPt.x()>tSizeX|| tTstPt.y()>tSizeY)
            {
                continue;
            }


            if(tTstPt!=_SenseDirecVect.Origin())
            {
                //if it's a blocking entity, break.  no sense examining further points since they can't be
seen
                if( const_cast<Map*>(aMapName)->IsBlockingEntity(tTstPt ) )
                {   //store the point, if not already known
                    if( std::find(aPointList.begin(),aPointList.end(), tTstPt) == aPointList.end() )
                    {
                        aPointList.push_back(tTstPt);
                    }
                    break;
                }
            }
            //store the point, if not already known
            if( std::find(aPointList.begin(),aPointList.end(), tTstPt) == aPointList.end() )
            {
                aPointList.push_back(tTstPt);
            }
        }
    }
}
```

Appendix C:

```
PT2D brain::PlanAction()
{
    PT2D tMovePoint(0,0);


    if( ( _AgentType == ePac )&&//Changes Pac back to feed after de-powering provided there are pellets.
        ( CheckMemoryForPellet())&&
        (_AgentState.InState(state::search)) )
    {
        _AgentState.ChangeState(state::feed);
    }


    if( (_AgentType==eGhost)&&//Changes the ghost to search state if he's gotten to his goal position and
Pac isn't there anymore and hasn't been seen again.
        (_CurrPos==_GoalPos)&&
        (_AgentState.InState(state::attack)))
    {
        _AgentState.SetState(state::search);
    }


    switch( _AgentState.GetState())
    {//Calls the functions that are applicable for each state.
    case state::feed://Pac only state.
        {
            tMovePoint = FindNearestPellet();
            break;
        }
    case state::flee://Both Agents have a flee state.
        {
            if(_AgentType == ePac)
            {
                tMovePoint = FindNearestPowerPellet();
            }
            else
            {
                tMovePoint = FindGhostFleePoint();
            }
            break;
        }
    case state::poweredup://Both Pac has eaten a power up pellet and is finding a ghost.
        {
            tMovePoint = FindNearestGhost();
            break;
        }
    case state::attack://Pac is fleeing but has seen a power up pellet, so goes to attack mode to get to
the pellet.
        {
            tMovePoint = _GoalPos;
            break;
        }
    default://Default is search
        {
            tMovePoint = Explore();
            break;
        }
    }
    return tMovePoint;
}
```

Works Referenced

Atkin, Marc S., David L. Westbrook, and Paul R. Cohen/University of Massachusetts. "Domain-General Simulation and Planning with Physical Schemas." Proc. of 2000 Winter Simulation Conference. 1730-738. Print.

Atkin, Marc S., Gary W. King, and David L. Westbrook /University of Massachusetts. "Hierarchical Agent Control: a Framework for Defining Agent Behavior." Proc. of Proceedings of the Fifth International Conference on Autonomous Agent, Montreal, Quebec, Canada. New York (N.Y.): Association for Computing Machinery, 2001. 425-31. Print.

Brooks, Rodney A. /MIT. "Intelligence without Representation." *Computation & Intelligence: Collected Readings* (1995): 343-62. American Association for Artificial Intelligence Menlo Park, CA, USA. Web.

Brooks, Rodney A./MIT. "A Robust Layered Control System For a Mobile Robot." *Technical Report: AIM-864* (1985). Web.

Bryant, Bobby D. "Evolving Visibly Intelligent Behavior for Embedded Game Agents." Thesis. The University of Texas at Austin, 2006. Print.

Buckland, Mat. *AI Techniques for Game Programming*. Cincinnati, Ohio: Premier, 2002. Print.

Buckland, Mat. *Programming Game AI by Example*. Plano, Tex.: Wordware Pub., 2005. Print.

Carlisle, Phil, Cloderic Mars, Baylor Wetzel, Alex J. Champandard, and Gwaredd Mountian. "AI Game Dev.Com Online Interview." Online interview. Sept. 2009.

Champandard, Alex J. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. Indianapolis, Ind.: New Riders, 2004. Print.

Duc, Le Minh, Amandeep Singh Sidhu, and Narendra S. Chaudhari/ Nanyang Technological University, Singapore. "Hierarchical Pathfinding and AI-based Learning Approach in Strategy Game Design." *International Journal of Computer Games Technology* 2008 (2008): Article 3. Web.

Gallagher, Marcus, and Amanda Ryan. "Learning to Play Pac-Man: An Evolutionary, Rule-based Approach." School of Information Technology and Electrical Engineering. Queensland,
Australia. Web. <http://www.itee.uq.edu.au/~marcusg/papers/CEC2003-1432.pdf
>.

Gruber, Thomas, and Paul Cohen/University of Massachusetts. "The Design of an Automated Assistant for Acquiring Strategic Knowledge." *ACM SIGART Bulletin* 108 (Apr. 1989): 147-151. Print.

Hagelback, Johan, and Stefan J. Johansson/Blekinge Institute of Technology, Ronneby, Sweden. "Using Multi-agent Potential Fields in Real-time Strategy Games." Proc. of Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, Estoril, Portugal. International

Foundation for Autonomous Agents and Multiagent Systems Richland, SC, 2008. 631-38. Print.

Hoang, Hai, Stephen Lee-Urban, and Hector Munoz-Avila, Lehigh University. "Hierarchical Plan Representations for Encoding Strategic Game AI." Proc. of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05) (2005). Print.

Karpov, Igor V., Thomas D'Silva, Craig Varrichio, Kenneth O. Stanley, and Risto Miikkulainen/University of Texas, Austin. "Integration and Evaluation of Exploration-Based Learning in Games." *CGames IEEE Dublin 2006*: 1-7. Web.

Koenig, Sven /USC, Maxim Likhachev /CMU, and Xiaoxun Sun/USC. "Speeding up Moving-Target Search." Proc. of Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, Honolulu, Hawaii. NY, NY: ACM. 11441151. Print.

Liang, Yuming, and Lihong Xu/Tongji University, Shanghai, China. "Global Path Planning for Mobile Robot Based Genetic Algorithm and Modified Simulated Annealing Algorithm." Proc. of Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation, Shanghai, China. New York, NY: ACM. 303-08. Print.

Lukasiewicz, Thomas /University of Oxford, UK, and Azzurra Ragone/Information Systems Laboratory Bari, Italy. "Combining Boolean Games with the Power of Ontologies for Automated Multi-attribute Negotiation in the

Semantic Web." Proc. of Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 02. IEEE Computer Society Washington, DC, USA, 2009. 395-402. Print.

Luo, Xudong, Nicholas R. Jennings, and Nigel Shadbolt/University of Southampton, UK. "Knowledge-based Acquisition of Tradeoff Preferences for Negotiating Agents." Proc. of 2003 ACM International Conference Proceeding Series; Proceedings of the 5th International Conference on Electronic Commerce, Pittsburgh, Pennsylvania. NYC: ACM. 138-49. Print.

McDowell, Patrick, and Cris Koutsougeras /Southwest Lousiana University. "Graph Memory Development in a Robot Control Architecture." *Journal of Computing Sciences in Colleges* 24.4 (2009): 7-13. Web.

Murphy, Robin. *Introduction to AI Robotics*. Cambridge, Mass.: MIT, 2000. Print.

Nejati, Negin/ Stanford University, Toiga Konik/ Stanford University, and Ugur Kuter /University of Maryland. "A Goal- and Dependency-directed Algorithm for Learning Hierarchical Task Networks." Proc. of The Fifth International Conference on Knowledge Capture, Redondo Beach, CA. New York, NY: ACM, 2009. 113-20. Print.

Orkin, Jeff/Monolith Productions/M.I.T. Media Lab. "Three States and a Plan: The A.I. of F.E.A.R." Proc. of GDC:06 Game Developers Conference 2006, San Jose, CA. 1-18. Print.

Pal, Hoimonti /Trinity University, San Antonio, TX. "Sustaining Learning in Critical Domains of Robotic Systems." *Journal of Computing Sciences in Colleges* 17.5 (2002): 66-71. Web.

Perkins, Simon, David Jacka, James Gain, and Patrick Marais/ University of Cape Town. "A Spatial Awareness Framework for Enhancing Game Agent Behaviour." Proc. of Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games, Los Angeles, California. New York, NY: ACM. 15-21. Print.

Rabin, Steve, and Jeff Orkin. "Applying Goal-Oriented Action Planning to Games." *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media, 2004. 217-28. Print.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Upper Saddle River, N.J.: Prentice Hall/Pearson Education, 2003. Print.

Terzopoulos, Demetri/Universtiy of CA. "Autonomous Virtual Humans and Lower Animals: from Biomechanics to Intelligence." Proc. of Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, Estoril, Portugal. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems. 17-20. Print.

White, Dustin/ Elon University, NC. "Clarifications and Extensions to Tactical Waypoint Graph Algorithms for Video Games." Proc. of Proceedings of the 45th

Annual Southeast Regional Conference, Winston-Salem, North Carolina. New York, NY: ACM. 316-20. Print.

notes

[1] Information taken from an interview with Alex J Champandard, AI Contractor for Killzone 2.

[2] All pictures of Pac-Man and target position obtained from 'The Pac-Man Dossier' with permission from their creator Jamey Pittman.

[3] Definition copied from John McCarthy's Stanford webpage.

[4] Phil Carlisle is a senior lecturer for University of Bolton in Bolton, England.

[5] Response taken from an interview questionnaire posed to respected industry veterans and academics on the aigamedev.com forums.