Artificial Intelligence

Choosing Technologies for Developing a Path Finding System

Author: Alberto Martinez
Submitted to: Professor L.S. Young
Writing 227: Technical Report Writing
Oregon Institute of Technology
March 14, 2006

Contents

Figures

A Note From the Author

This report was created throughout development and research of the prototype for the path finding system developed for the Pitchfork Engine of the Plus Minus Software group. All work that is not my own has been cited using the APA style guide.

Throughout the text, there is reference to a system called Hermes. Hermes is a subsystem in development that defines how units will move and interact in a real time strategy video game engine. For the purposes of this report, the system is developed to the point where only singular unit movement is achieved with influence map weighting. Sounds like a lot but the terminology will become clearer through the report.

Some history that lead me to naming the system Hermes.

Hermes, the herald of the Olympian gods, is son of Zeus and the nymph Maia, daughter of Atlas and one of the Pleiades. Hermes is the god of shepherds, land travel, merchants, weights and measures, oratory, literature, athletics and thieves, and known for his cunning and shrewdness. Most importantly, he is the messenger of the gods. Besides that he was also a minor patron of poetry. He was worshiped throughout Greece -- especially in Arcadia -- and festivals in his honor were called Hermoea.

– Leadbetter, Hermes, MMV Encyclopedia Mythica, 2002

I named my subsystem after Hermes because my subsystem is a messenger to the units that are interacted with. The units will ask Hermes for a path and Hermes will tell the units the path that they will take, if any.

Most of what is in this report is the afterthought of research done in order to accomplish the goal of developing a functional prototype. No attempts have been made to take the work of other authors and claim it as my own. I include references that are not cited because they were my source of research and inspiration in piecing together my ideas and thoughts into a system. I would like to thank those authors for providing me education and direction in how to focus my thoughts and further research towards achieving a well-developed final system. Enjoy!

Abstract

Many people have played a video game sometime in their life. Some play more than others and others even more so than should be allowed for any person. It takes a good game developer to stand back from a game and ask the question that plagues many software engineers, "how did they do that?" This report covers specifically one aspect of a game that many engineers find curious. Path finding, what is it, how does a game developer implement it, and how is it done in commercial video games?

Artificial intelligence has been a favored subject of mine for quite some time. I am the kind of person who plays a game and is constantly considering how such behaviors are programmed. The hardest part about this void of knowledge was asking myself, a future software engineer, how I would go about designing such a system if I were ever faced with the need to produce one. When I became a member of the Plus Minus Software group, I was put in position as lead in the development of AI in the Pitchfork Engine. I needed to answer my questions very quickly.

This report is a guide on deciding what technologies to use when developing a path finding system. Through the duration of this guide, there will be many references to what I personally chose as a route in developing Hermes, a sub-system that exists in the Pitchfork Engine developed by PMSoft. After this report, an intermediate to advanced level software programmer should be able to walk through the guide and pick out techniques, algorithms and theories that will suite her or his needs.

By the end of the report many algorithm and data structure choices will be made. Algorithm definitions that will be explained include: Crash and Turn, Dijkstra's Shortest Path, Breadth-First, and A*. Data structures that will be discussed will include: vectors, lists and some custom-made trees. While some terminology will be explained through the duration of the report, the audience will be assumed and some knowledge will be prerequisite to fully understanding the content of this report.

The various decisions that will be made by the reader include overall process speed, memory footprints and defining what is to be an optimal path. Speed will be considered because in a real-time game, speed is most everything. Memory footprint is a large consideration depending on the content the game brings with it. Finally, there are some innate problems that should be considered given a

choice. The most obvious is with the crash and turn algorithm which if left unchecked encounters an unsolvable problem during run-time of the algorithm. The crash and turn and other algorithms all yield different results that are defined as the optimal path for that algorithm.

Other subjects that I will briefly discuss are those of human path finding and group dynamics. Human path finding will be discussed to put into light what the reader wishes to accomplish in developing their expert system. Group dynamics will be briefly covered as a topic that may need to be researched after reading this report. If the software developer reading my report is developing a system involving many entities moving in a common space then group tactics must be designed to successfully allow movement of entities.

For further clarification on some of the terminology that will be in the body of this report, please see **Appendix B: Glossary** for more information.

Getting Started

Define the Problem

A huge bottleneck that commonly occurs amongst new and avid game developers is getting stuck on figuring out what exactly they want. They know that they want an intelligent system but they cannot wrap the idea around limiting the possibilities their system will have to define a feasible problem that is solvable. Any given piece of software that is made is never actually complete because there can be processes carried out on the code to make it fundamentally better.

This ideology must be thrown out. A developer's system will not save the world and the sole purpose of it is to solve one problem. Once that problem is solved then the developer can move on to another problem and solve it. If all software engineers looked at the big picture the entire time, nothing would get accomplished because of the sheer fear of dealing with such a task. So if a problem gets out of hand, it must be broken up into smaller, more manageable sub-problems that can be easily dealt with.

That said, it is up to the developer to take a moment and define exactly what it is their system will be doing. If the task is very large and will involve multiple technologies to coincide with each other, then the developer should tackle these sub-problems individually and through this walkthrough one at a time. Once we clearly define a problem we can abstract it into a possible solution.

Some factors to take into consideration for the system include: speed, memory footprint and the necessity of determining the most optimal path. These three aspects are large considerations to account for in developing a path finding system. They are, for now, very general but will lead towards a more concise definition later on.

Determine an Optimal Solution

Now that the developer has determined a problem, they can use the problems and determine which route to take in finding useful and appropriate technologies. If the problem includes speed, then we must determine which technology yields a very fast process in determining a path. If the problem includes reducing the cost of memory, then we must find a technology that uses barely any memory. Finally if our problem includes defining the best possible path, then we must find a technology whose solution involves the best possible path.

This makes our definition of success very critical. If a developer desires a solution to all three problems, then the only possible solution is a technology that breaks even in solving all problems. If such a technology existed that would solve all of these problems then there would be no need for this walkthrough. There are benefits and consequences for using any of the technologies that will be discussed. It is up to the developer to decide on what strength to focus on for their desired system. A strong and confident problem will ultimately yield a strong and confident solution.

After a problem and solution have been defined the developer can now begin breaking down the available technologies into groups and start fine-graining the technologies towards a satisfactory choice.

For Hermes, I need a fast algorithm that will define a close to optimal path from one point to another. Already the need for memory is thrown out and disregarded because it is simply not that important for my desired implementation.

Local and Global Algorithms

Algorithms can first be split up into two categories for path finding, local and global. The difference between the two is simple. A local algorithm is defined as one that only uses a limited amount of information around the unit or, in some cases, within provided eyesight of an entity. A global algorithm is defined as an algorithm that is provided all information of maneuverability in a given space. The developer must decide how much information will be provided to this system. Will the system know all boundaries and obstacles or will it know only the area around it and a general direction to travel?

Time after time, games have been released with little information pertaining to in-game AI. What information is either provided or discovered in a game title shows that most AI can "see" everything inside the environment. However much some people may believe this amount of information provided to a computer is beneficial and deemed to be "cheating," it is common practice to implement such a system for the sake that it provides practical results.

Most often computer intelligence is provided all information so that the computer can be focused on other tasks. This focus spills over to create a disadvantage to the players, believe it or not. Take for example a real-time strategy game. An RTS uses global algorithms for path-finding in order to allow

a player and a computer to focus on other more important aspects of the game besides how a unit gets from one point to another. If a unit had a local algorithm attached to it, the task of traversing an area becomes both demanding and frustrating. In order for a local algorithm to be completely successful is to implement a "learning system" to create individual maps of the walking space (Russel et al, 1995).

The overhead for local algorithms involve processing power while global algorithms consume a larger memory space. Local algorithms are constantly testing areas and therefore constantly processing information. Global algorithms usually make a single process and then the entity traverses the result.

The developer must choose what their system is to accomplish. Local algorithms yield more realistic results in traversal, but to what end? Global algorithms can mimic "lost" behavior well enough that they are the primary choice for game developers.

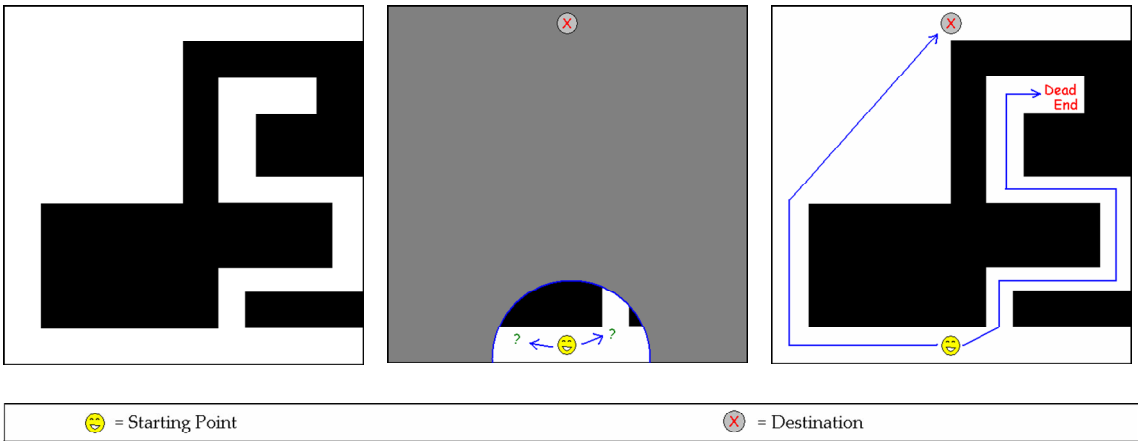
Human Path Finding

To get some perspective on path finding algorithms, let's stop and think about how humans trace paths in the real world. We tend to rely on very different methods than those used by machines (Dalmau, 2004, pp. 235). As discussed earlier in the text, there is a large difference between local and global algorithms. If somebody were to be dropped off in the middle of a city, that person would start with a local algorithm and expand using traits of a global algorithm to reach her/his destination. As Dalmau described the process, "people use a hybrid local-global algorithm"(Dalmau, 2004, pp. 235).

The biggest question this leads up to is "how much intelligence do I want my algorithm to display?" The answer depends on how human the implemented system is going to be. In the most common approaches to game path-finding, the local versus global choice depends on the genre. First-person shooters commonly stick to local algorithms with general direction being the only leaked information. In this scenario, if a computer is provided all information, then it would simply traverse to the player very quickly and eliminate the player as a computer's aim could also be unfair. Real-time strategies and role-playing games, which is what Hermes is being developed for, tend to stick to global algorithms for the sake of maintaining player interest. The idea here, that if a unit were to mimic the behavior of a human, the game play would become increasingly frustrating as units would, at first, have no idea of where to go. It

also falls into the idea that a general, in a common army situation, will tell a unit exactly where to go since the general knows the terrain.

In short, in choosing between local and global algorithms, the developer must take a moment to consider what exactly the system will be mimicking and how successful that concept fits into the scenarios it will be faced with. Determining this is no easy task, but through some foresight into the goals of the system, the developer should be able to make this choice fairly easily.



(**left**) Map of problem. (**middle**) Which direction would you choose, given this amount of information? (**right**) Results when provided all information. This is the Computational result. The Human would most likely traverse the right side first and then traverse the entire way back in order to find the solution. A computer would test the right side instead of traverse first.

Figure 1: Human Decision

Path Finding Technologies

Brief Overview

So far, the developer knows what optimal solution is being aimed for and whether the system at question will involve a local or global algorithm. Now it is time to pick an appropriate technology. To accomplish this task, the technologies must be defined clearly and concisely related to what problems and solutions they instantiate.

Crash and Turn

The first algorithm we will explore is a local method that quite closely resembles what a simple animal would try to do. In the most simplistic form the algorithm breaks down into several small processes. After an entity has a target chosen, the entity attempts to move in a straight line towards the objective. When a collision into an obstacle occurs, the entity randomly chooses a direction and follows the side of the obstacle until either another collision happens and the entity begins

following that wall, or the wall ends and the entity is allowed to walk in the desired direction towards the objective.

Already, the developer should be able to pick up some possible issues with an algorithm such as this one. The use of randomly generated choice is an obvious flaw in most intelligent systems. To leave a decision up to randomness is begging for problems to follow close by. However, the use of a random result in this algorithm yields surprisingly good results. Though somewhat distant, a solution may be for even the simplistic of traversals, simple heuristics can be used to determine a more educated choice of direction (Dalmau et al, 2004).

Breadth-First

The Breadth-First search method is an easy-concept algorithm that is not admissible. Starting from the first node, the surrounding nodes are collected and given a weight. The list obtained is then used to keep adding to the list. The node at the beginning of the list has connecting nodes added to the bottom of the list and so-forth. This continues until the goal is reached where the path is generated from the destination to our starting location. I like to refer to this algorithm as the round-about algorithm.

I call it this because if one were to break down the algorithms to visualize the process, you would see that starting from the starting node, the iteration seems to corkscrew or rotate around in an outward motion. The obvious problem with this algorithm is the level of predictability it demonstrates. The algorithm will essentially always do the same thing which means that as unobstructed distance between two nodes increases, the processing the algorithm requires grows exponentially. That said, the Breadth-First is not a recommended algorithm to use in large, open space areas.

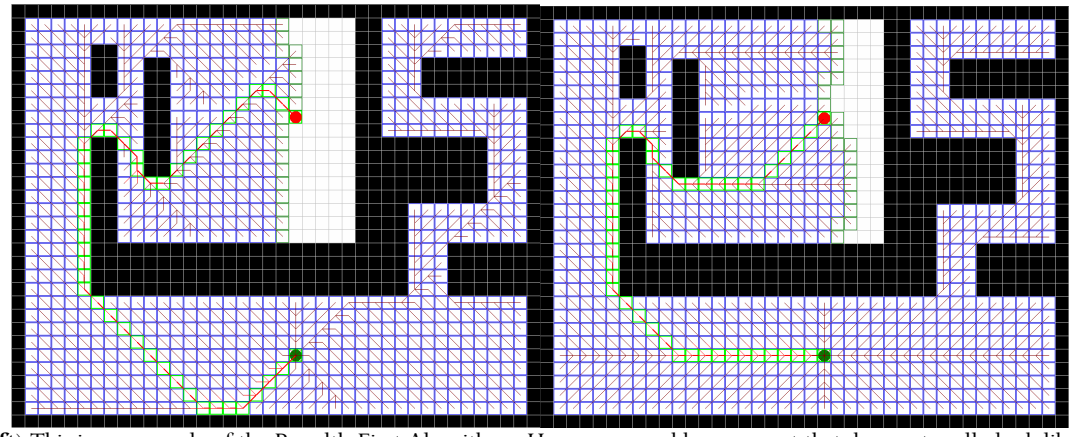
Dijkstra's Shortest Path

Created by Edsger W. Dijkstra, a Dutch computer scientist and mathematician, this is an admissible algorithm. Dijkstra's method is defined using a set of edges and vertices. In the most simplistic approach to describing this algorithm, Dijkstra's is a Breadth-First search with weights. Where, starting from the root node, the algorithm expands from the center weighing every possibility until the goal node has been reached. The cost of traversing from one node to the other is stored in the edge that connects two vertices.

The problem with Dijkstra's Shortest Path is the predictability that follows it. Every single time the algorithm runs, the process is the same. Starting from the

first node, it expands its sight until it collides into the destination node. After the current "layer" is calculated, then the path is created by following the destination node backwards to the starting node using the calculated weights. What this means is that the algorithms process time, an inherited trait from the Breadth-First algorithm, grows almost exponentially as the distance between the start and finish grows in length.

Comparing Dijkstra's to Breadth-First yields two conclusions. First off, Dijkstra's algorithm returns what looks like a very concise and desirable path that would create some intelligent-looking traversal across a 2D space. Figure 2 below shows that Dijkstra's generated path is a bit more direct than a path chosen by a breadth-first generated path. The distance of these paths is actually equal; what is being compared is the behavior the two algorithms display.



(**Left**) This is an example of the Breadth-First Algorithm. Here we see odd movement that does not really look like anything other than something random. However this is an optimal solution. (**Right**) This is the same example using Dijkstra's Shortest Path. This path seems more reasonable and a good choice for acting out intelligent movement. From: Stout, Bryan."Path Search Demo."

Figure 2: Breadth-First vs Dijkstra's

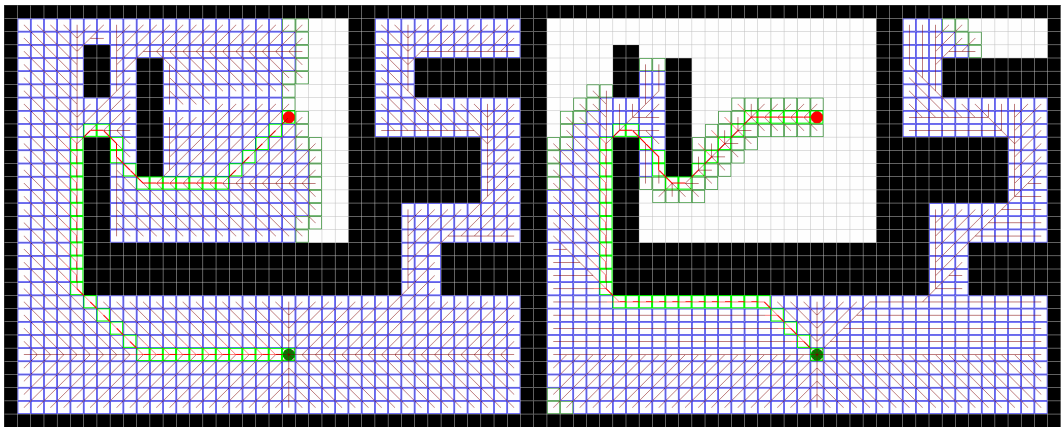
A*

Simply put, A* is a super-powered version of Dijkstra's. In the previous description, Dijkstra's algorithm proved somewhat inefficient since no care for possible optimization was taken, the algorithm proceeded every time with brute force. A* resolves this issue and turns Dijktra's algorithm into a solid choice for path finding. Let me explain exactly what separates A* from Dijkstra's.

Returning to a previous topic on human path finding, remember the situation where a human is put into the middle of a city with a compass and general direction of where he/she is going. This concept is what gives A* it's well deserved fame. In Dijkstra's approach, all connecting nodes expanding out from

the root node are considered. However, if the algorithm attempted a possibly better direction first then some processing time may be reduced. Using heuristics and some data structure tricks A* accomplishes this. Combining the weight of movements away from the root and heuristic values providing a rough guess on how close a node is from the goal, the algorithm distinguishes between nodes more worthy of attention and nodes not worth any time at all.

So instead of simply obtaining a list of nodes to calculate next, A* prioritizes gathered nodes by their rate. A lower rate value means a simple path will be found quickly and without the need to calculate more nodes. Figure 3 below compares the two algorithms. This shows a small obstructed path being calculated by both algorithms. Dijkstra's proves to have a larger amount of nodes being looked at to calculate a path. It is this characteristic that makes A* so appealing to any developer. It has a very flexible capability for behavior. If A* uses a heuristic that never overweighs node traversal to the goal then A* can be proven admissible and it is this optional behavior that allows A* to have a diverse behavior set in any application.



(**Left**) Dijkstra's Shortest Path Algorithm. (**Right**) A* Algorithm. The white space shows visually that A* looks for a path by ignoring area that is least likely to find the destination.
From: Stout, Bryan."Path Search Demo."
Figure 3: Dijkstra's vs A*

So why would one ever want to use Dijkstra's over A*? The answer lies in memory. While describing how A* worked there was mention that a new score and method of storing nodes were needed to accomplish A*. The rating of the nodes doubles the amount of information a node holds thereby increasing A*'s memory footprint as well as processing time depending on what calculations for the score is required. Tied together with the need to prioritize nodes by these new values and A* has turned into a memory and processor hog. Tweaking these properties of A* though has put it into a shining light for software

developers. Since speed is always so important for real-time systems A* is a common choice for developers. This is the choice algorithm that will be used in the Hermes path finding subsystem. If A* is to become a choice technology, then having a complete understanding of the algorithm is very important as it's behavior will vary significantly from implementation to implementation.

Iterative Deepening A*

The sister algorithm to A*, Iterative Deepening A*, or IDA*, begins with an approximate answer and then expands that approximation until an absolute answer is discovered. The pitfall to this algorithm lies in that one word "discovered." IDA* only tests a path for success then expands that path to further test until success has been found, testing and testing nodes constantly to progressively build a path. What this yields is an algorithm with a small memory footprint but large processing time.

This probably sounds completely different from A* and it is commonly asked "what makes this any sort of A*?" Well the answer is just as simple as why A* is not Dijkstra's and that is approximation. Where A* can approximate a direction to possibly optimize the time and memory required to calculate path, IDA* can approximate a goal to simply attempt optimizing process time.

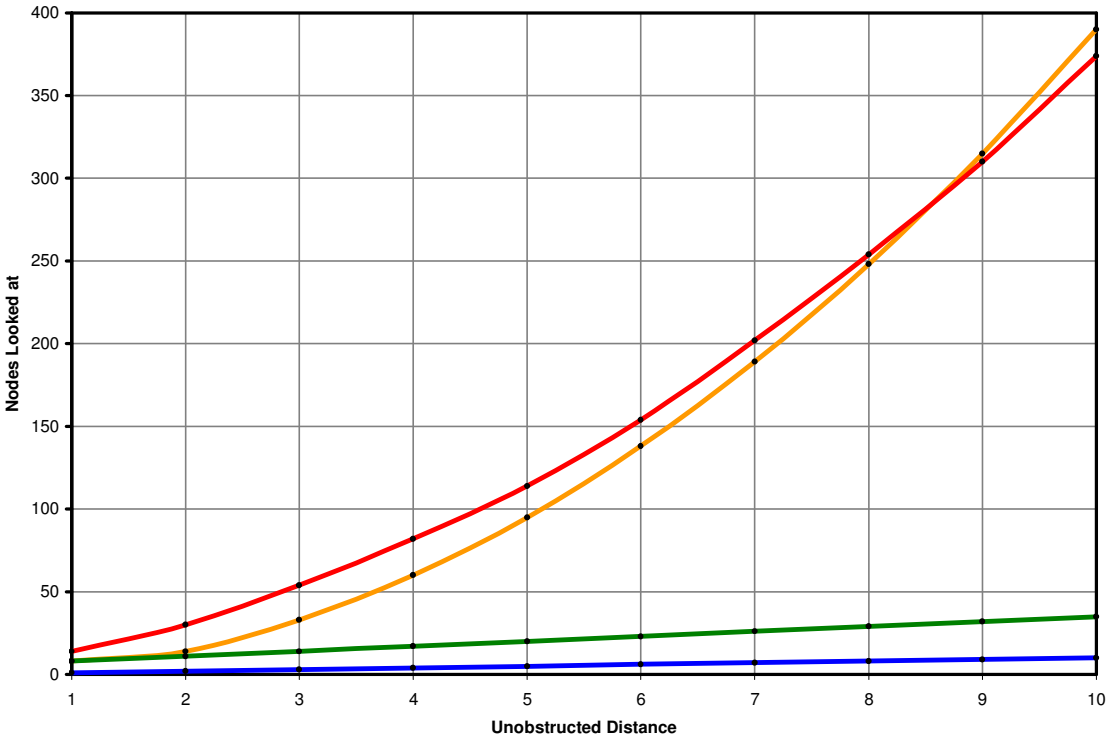
Now the same concept of why A* is more memory dependent flips over when it comes to IDA*. IDA* is thought of as the inverted algorithm of A* because the memory footprint of it is so small and the processing required is larger in comparison to A*. The optimal path both find are great though not as comparable to that of Dijkstra's but it is within speed and memory that A* and IDA* both become a prime choice.

Algorithms at a Glance

All this explaining on who does what and all that is remaining is one tough decision. "Which algorithm is the best for what I want to do?" The answer is not simple, and hopefully there has been enough explaining to come to a good decision. Let's break down the criteria for making a choice again and see where each algorithm belongs. The three motivations for choice are: speed, memory footprint and optimal path.

Memory footprint and process time are easily estimated. What is looked at here, for Breadth-First, Dijkstra's and A* are how many nodes are considered or looked at for a solution. This means that the nodes for the algorithms are calculated or even instantiated which gives a very good estimate on how much

memory and how much process an algorithm will consume during calculation of a path. In figure 4 below we see these relationships to each of the algorithms explained in the previous sections. Crash and turn has been left out of this comparison for two reasons. The first is simple, Crash and Turn yields no memory constraints. Since it is a real-time algorithm, it needs only state of the entity it is driving to determine new movement for the entity. The second reason is because the Crash and Turn method utilizes random movement which cannot be successfully contained and tested to produce comparable results.



Red: Dijkstra's Shortest Path **Orange:** Breadth-First Search **Green:** A* **Blue:** IDA*
Data From: Stout, Bryan."Path Search Demo."

Figure 4: Approximate Unobstructed Comparison of Algorithms

Using the STL

For those software developers planning to implement a path finding system using C++, use the STL. This is very important for achieving satisfactory results in a system. The STL is intended to be a "fast, sleek machine" (Meyers, 2001, pp. 1). Moreover, the STL is "error-free because it is used daily by thousands of programmers worldwide" "the fastest access algorithms are built into the system" (Dalmau, 2004, pp. 88 – 90). What this means is that by using the STL, an algorithm can be developed without worry that the underlying data structures are the direct cause of odd behavior. The STL also provides, if implemented correctly, the fastest bridge from one end of the algorithm to the other.

Group Dynamics

Brief Introduction

At this point a path finding algorithm should be picked out of the mix and ready for implementation. However, there is one detail that may be overlooked or it simply is not a consideration for the system being developed. Nonetheless it is a question that needs answering. The technology chosen easily deals with single entity movement across a given 2D space. Now the issue comes to light that the system to be implemented may not have the luxury of only having to deal with single entity movement but multiple entity movement. A decision must be made on how to deal with such an issue.

The most simplistic approach for deciding a direction for research is to determine 3 things. Will entities simply move in the 2D space considering one another when determining paths? Will the entities move together in any given situation? Finally, will the entities accomplish both of these tasks? The breakdown of these questions points to what is to be discussed later in this topic.

Boids Three Laws

"One of the best examples of synthetic group behavior can be found in the boids algorithm introduced by Craig W. Reynolds in the 1990s" (Dalmau, 2004, pp. 237). The boids algorithm is defined by variances in the three laws that drive it. These are:

- Separation
- Alignment
- Cohesion

Separation is the law that defines a kind of social threshold or boundary that exists in a group. The common example of this would be an occasion where one

entity is "too close for comfort" to another entity. Both entities resolve this "violation of space" by both moving away until a comfortable space is acquired. The second law, alignment, is easily exampled by an attacking army. Alignment enforces all army-members to face in the same general direction of the attack because it would not make sense to do otherwise. The third law, cohesion, destroys individuality. Cohesion is a programmed "desire" to stay close to a groups' barycenter which is described by a group's center or focus.

Varying enforcement of each of the laws produces different emergent behaviors. A diverse system implementing the boids algorithm is ready to change the values associated with each law determined by situation to produce "life-like" response to situations.

Collision Detection

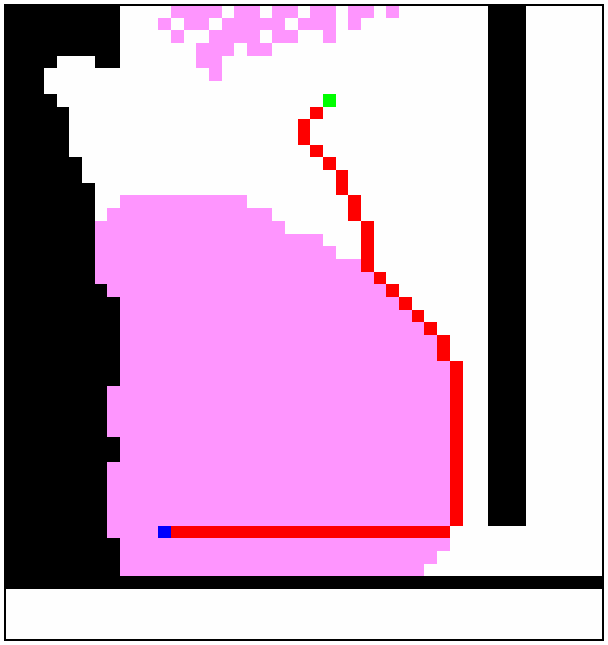
No group dynamic system will function correctly without the deployment of some collision detection. Collision detection is the detection of two entities colliding with each other and how to deal with such a situation. Left unchecked, entities are allowed to freely traverse through one another. The degree to which a collision detection system must be is again dependent on the optimal solution to a problem. Possible implementations include finding collisions before they happen in real-time and dealing with those collisions.

The most common way to implement a collision detection system is to implement one separate from any path finding or group dynamic algorithm. This way the collision detection stops a collision from happening and tells the appropriate system to recalculate. This method resolves unexpected results which may be very important to a system. If the system does not expect a collision but is ready to deal with one, then the system is mimicking the form of intelligence that realizes, "hey, we all make mistakes."

Influence Mapping

Earlier, the idea was discussed that for a path finding algorithm to function correctly and accurately it must be able to weigh nodes in order to find an optimal path. Influence mapping expands that idea in order to provide a medium for automation. Allow the setup of this scenario. Say a soldier is secretly infiltrating an enemy base. The soldier has exact knowledge of where the enemy can, and cannot see around the base. The soldier is going to find the safest path possible inside the base. The safest route would be defined by the route at which he is in the least amount of danger. This is the primary concept behind influence mapping.

Given an enemy entity and the sight range that entity has, the nodes around said entity will have a greater amount of influence than nodes that are outside the entity's sight. Using this information in the algorithm, an entity can determine the safest path around or to that enemy entity. In Hermes, influence mapping is transparent to the user and not considered when the user chooses a location for her/his units to move. However, influence mapping provides a mean to automate strategy for computer AI. In the screen shot below we can see this automation of strategy and the unit has chosen the path of "least resistance" into the area of influence.



This figure shows Hermes' implementation of influence mapping. The chosen path into influence is the shortest path considering threat of the influence.

Figure 5: Hermes' Implementation of Influence mapping

Conclusion

Path finding is such a small subject inside the realm of artificial intelligence. An idea to create an expert inside of a computer that can expand so fast through statements that all people take for granted. As with all applications of AI, knowing when to draw the line is important to accomplishing goals and successfully implementing a system in enough time to worry about missed details of it.

The contents just explained in the report can be summed up into 5 complex steps.

1) Define the problem
2) Determine a solution to that problem
3) Choose the technology that meets or exceeds the requirements for solving the problem
4) Implement the technology
5) Consider other desirable features

The fifth item on the list is the one to be worried about. The common idea behind developing software is that software is never finished and never complete. Be sure to stay within the bounds of the problem. Once that problem is solved then ascertain any exceptional problems that the system shows. Redefine the problem and start over. Going beyond a problem at a mere whim is dangerous because of time constraints and logical constraints on a system. If it is a word of advice is to be here it would be "know when to be done."

Appendix A: Implementation of Chosen Technology and Methods

Description

The purpose of this experiment is to implement a technology and methods chosen using this report. I will describe the results that were implemented in Hermes and reflect on how the described process illustrated in this report help the success or failure of Hermes.

Purpose

The purpose of this experiment will be to determine if the steps illustrated in this report prove useful in choosing a technology.

Relevance

This will show the audience the result of using this guide and how much time it can potentially save. "Wise man learn from his mistakes, wiser man learn from others' mistakes."

Contribution to Subject

I will use the results of this experiment throughout my paper. If I make an inappropriate choice pertaining to a technology then I will describe to the reader the consequences I had to overcome in order to achieve my goal. The goal that I am trying to achieve will be my connection to the reader. My success in this endeavor will "sell" this report to software developers considering achievement of the same goals that I had to achieve.

Specific Methodology

What: Implementation of Path-Finding Technology in the Hermes sub-system of Project Pitchfork. How: C++ and standard GDI graphics output. When: Throughout the Winter term of 2005. Where: On my desktop computer.

Results

The results of the experiment were a complete success. Hermes implemented a modularized A* function that calculates the shortest path on a simple bitmap pixel-is-a-node interface. The results of which are included with this report.

Please see **Architectural Prototype** for more information.

Appendix B: Glossary

Admissible – "A search algorithm that is guaranteed to find the shortest path to a goal is called admissible" (Wikipedia, the free encyclopedia, 2006)

Artificial Intelligence – 1 : the capability of a machine to imitate intelligent human behavior **2 :** a branch of computer science dealing with the simulation of intelligent behavior in computers (Merriam-Webster OnLine, 2005)

Barycenter – Also known as the Center of mass. The barycenter in group dynamics is the definition of the point where a group of entitys are averagely spread out with respect to this point.

Group Dynamics – the interacting forces within a small human group; *also* **:** the sociological study of these forces (Merriam Webster OnLine, 2005)

Hermes – A subsystem of Project Pitchfork which delegates where units move. Please see www.PMSoft.org for more information.

Heuristic – involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods <*heuristic* techniques> <a *heuristic* assumption> (Merriam-Webster OnLine, 2005)

Node – a point at which subsidiary parts originate or center (Merriam-Webster OnLine, 2005)

RTS – Real-Time-Strategy

STL – Standard Template Library. A C++ library of container classes, algorithms, and iterators.

References

Dalmau, D. (2004). Core techniques and algorithms in game programming. Indianapolis, IN: New Riders.

Drozdek, A. (2001). Data structures and algorithms in c++. 2nd ed. Pacific Grove, CA: Brooks/Cole.

Leadbetter, R. (2002, Mar 26). Hermes. *MMV Encyclopedia Mythica*, Retrieved Mar 11, 2005, from http://www.pantheon.org/articles/h/hermes.html.

Marriam-Webster, (2005). Retrieved Mar. 11, 2005, from MERRIAM-WEBSTER ONLINE Web site: http://www.webster.com.

McConnel, S. (2004). Code complete. 2nd ed. Redmond, WA: Microsoft Press.

Meyers, S. (2001). *Effective stl: 50 specific ways to improve your use of the standard template library*. Indianapolis, IN: ADDISON-WESLEY.

Patel, A. J. (2004). Thoughts on path-finding and a-star. Retrieved Feb. 23, 2005, from http://theory.stanford.edu/~amitp/GameProgramming.

Petzold, C. (1999). Programming windows. 5th ed. Redmond, WA: Microsoft Press.

Plus Minus Software, (n.d.). Retrieved Mar. 14, 2006, from Pitchfork Engine Web site: http://pitchfork.pmsoft.org.

Russel, S., & Norvig, P. (1995). Artificial intelligence: a modern approach. Englewood Cliffs, NJ: Prentice Hall.

Schreiner, T. (n.d.). Artificial intelligence in game design. Retrieved Feb. 23, 2005, from AI-depot Web site: http://ai-depot.com/GameAI/Design.htm.

Stout, B. (1996). Smart moves: intelligent path-finding. Game Developer Magazine.

Stout, Bryan."Path Search Demo." PATHDEMO.exe. DELPHI, . 29 Sep 1996. Program. 24 Feb 2005 <http://www.gamasutra.com/features/1990212/pathdemo.zip>.

Wikipedia, the free encyclopedia, (n.d.). Retrieved Mar. 10, 2006, from A* Search Algorithm Web site: http://en.wikipedia.org/wiki/A%2A.

Wikipedia, the free encyclopedia, (n.d.). Retrieved Mar. 10, 2006, from Boids Web site: http://en.wikipedia.org/wiki/Boids.

Design Choices

For my first, second and third level headings I decided to keep it simple. Everything in the report, with the exception of some figures, is kept to the left side. This maintains a very strong line at the left of the page where the reader should start her/his reading. The level one headings are Palatino Linotype, bold, size 14. The second level is same typeface, bold, 12. Third level headings are not present in my report. I excluded such a division to help keep simplicity maintained throughout the report.

My choice for contrasting the different heading levels is also kept very simple. Level two headings are preceded by level one heading with no white space division dictating the separation, only font size. This is enough to make the level one look more important but does not create a double inclusion of contrast.

The paragraphs within body text are separated by one line of white space. The paragraphs are not indented for the purposes of, again, avoiding double contrast inclusion. This paragraph segmentation helps the reader stay focused on text one paragraph at a time. The figures are contrasted by a modest changed in font typeface and size. What this helps to accomplish is segmentation of the surrounding body and the figure. The contrast is not so drastic that the reader will stay focused on the figure, but instead will help the reader choose focus on the figure or the body.

The font choices were made to be easy on the eye. Technical topics are hard to keep focus on and are generally head-nodders (they make you sleepy). Making the text pleasant to read was a must for the success of this report.

Architectural Prototype
Hermes Path Finding Sub-System

Developed by: Alberto Martinez

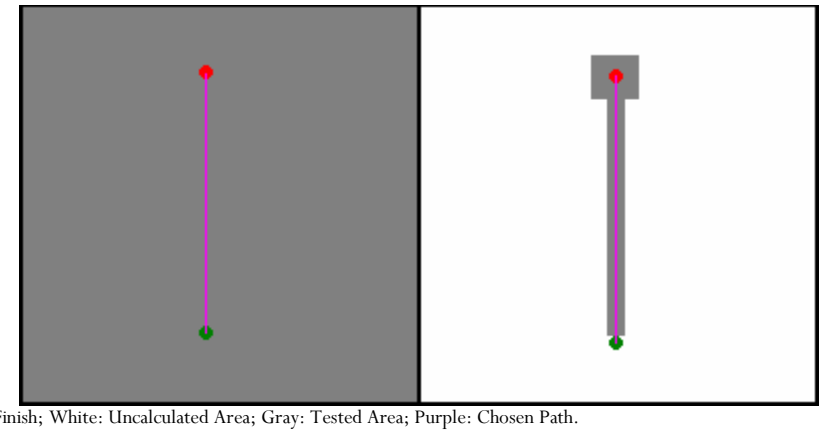
Abstract

This prototype has been developed to demonstrate and absolve the highest priority requirements that reflect unit interaction. Unit interaction, put simply, is the interaction between an actor on the system and the instantiated data objects that represent the actor's units during game play. The actor not immediately involved with the result of this system is the developer. This subsystem lives inside of our system and is therefore not directly approachable by the developer. However, some details may exist to kind of tweak the behavior that this subsystem is capable of demonstrating.

This prototype will demonstrate one of these small developer features but for the most part the prototype is much more focused on interaction of actors other than the developer.

Goals

The primary objective of this prototype was to achieve the goal of singular unit movement in any given 2D space. To achieve this, Hermes employs the use of the A* algorithm. A brief overview on how A* works. A* can be seen as a beefed up Dijkstra's Shortest Path Algorithm. It is, for all intents and purposes, Dijkstra's but with a modified behavior provided by heuristic weighting. Dijsktra's spans out from a node to all nodes surrounding it and spans out from nodes connected to those nodes and so forth. A* does this but puts priority on nodes it wishes to focus on. A* also ends at the first occurrence where the goal has been reached, optimizing the time the algorithm takes. To illustrate this I have included a figure illustrating exactly what I mean.



Green: Start; Red: Finish; White: Uncalculated Area; Gray: Tested Area; Purple: Chosen Path.
Left: Dijkstra's algorithm. Every node is checked and weighted to find an optimal path. Right: A* favors a direction and checks node in that direction to save some time and precious resources.

Figure P1: Two Algorithms at a glance

As you can see Dijkstra's is more of a brute force approach to path finding. It is for this reason that I chose A* because A* can possibly reduce how much memory and processing is required for a solution. The concern for memory must be issued because the common approach to implementing a path finding algorithm is pre-instantiation of nodes required for calculation. The Hermes system employs the instantiation of nodes on the fly to decrease immediate demand for memory. This allows the use of multiple paths to be calculated on a common map. In retrospect to developing this system, this decision was a mistake. Since units will constantly be searching for a new path, the memory we are trying to save, instead slows to system down significantly because instead of simply traversing nodes, we are instantiating and deleting nodes constantly. A more appropriate method to applying this algorithm would have been to have a single instantiation of all nodes required and then another layer to manage all the

unit's request for a path. This would then allow the algorithm to execute without allocating or deallocating memory, therefore, a much faster and efficient process.

Another goal with this prototype was to begin to introduce the concept of influence mapping to the path finding algorithm. What an influence map accomplishes is a sort of derived artificial strategician that can be used to drive the AI in the system. As far as movement is concerned with a computer player, easily derived targets are attacked and put through the path finding which calculates a level of strategy for attacking units to follow. Stepping back just a little allows me to discuss exactly what influence mapping is.

Given a unit at a location, one of the properties of the unit is sight. Sight allows the owner of the unit to see objects and/or other units inside of a given radius that the unit has. This sight, for a computer, can be approached as a assumption that if one of it's units is in the sight of an enemy player then that unit is in a threatening zone. The computer will obviously want to keep it's units out of a threatening zone unless itself chose to become the threat. So as units are traveling the map we may want to give the computer enough information so that it may calculate paths that are non-threatening.

So in the prototype, these influences are displayed by pink pixels in white area on the map. One of the goals of the prototype is to consider these nodes as walk able areas but as influenced walk able areas.

Finally, the ultimate goal of the prototype was the implement all of the above in accordance with the architecture proposed through design process. As the reader, if you decide to play around with the Hermes prototype please be sure to look at the code and the sequence diagrams that reflect the code, or vice versa. You will see that the two are one to one.

For further information on the subsystem, please visit the website of the project containing Hermes: http://pitchfork.pmsoft.org
As well as my personal website: http://zenphoenix.net