A Penguin Attack AI

Jason Buck

Abstract

Penguin Attack is a clone of the Nintendo game originally released as "Panel-de-Pon". It is a project that Tom Kircher and myself conceived and worked on over the last decade. Penguin Attack is a competitive 2-player, block-swapping, color-matching puzzle game. One problem with our game was that if a player didn't have a human opponent handy, they couldn't really play the game (at least not for more than the few seconds it takes a non-playing opponent to lose the game). This semester for my integrative capstone CS class, I wrote Penguin Attack AI, which is a computer controlled network client that plays Penguin Attack. Never again shall one be left longing for a Penguin Attack opponent.

Introduction to the Problem My Software Addresses

About 10 years ago my friend Tom Kircher and I decided that we wanted to write our own implementation of our favorite puzzle game. Our intention has always been to make it free and open-source. Initially we made rapid progress in developing our implementation, but as so often happens with pet projects, it has been shelved off and on over the years. Last semester I took an Artificial Intelligence (AI) class. I found the topic fascinating, and started thinking about writing an AI that plays Penguin Attack. This semester I needed to take my CS integrative capstone course, and was required to select a project to work on for the semester. I chose to write a Penguin Attack AI because I thought it would be interesting, and because I thought it might reignite both Tom and my passion for the project. The project was successful in each of these areas, as Tom has been hard at work on the game itself, and I thoroughly enjoyed working on the AI.

In the game of Penguin Attack, each player is given a board that contains a stack of blocks, and a cursor which can be used to highlight two horizontally adjacent blocks to swap them. Each player also has an invisible timer called the stop timer. The value of the stop timer is initially set to zero. Whenever the value of the stop timer is zero, the stack slowly rises, with new randomly selected blocks filling in from the bottom, until it is touching the top of the board. When the stack first touches the top of the board, five seconds of 'crunch time' are added to the stop timer. If any portion of the stack is touching the top of the board and the value of the stop timer is zero, the player loses the game. By swapping horizontally adjacent blocks, the player attempts to align (either vertically or horizontally) three or more blocks of the same color (called making a combo) causing them to disappear, and causing the pieces above them to fall into the previously occupied spaces. When those pieces fall into their new spaces, if they create one or more additional combos, the player has executed what is called a chain. By executing combos and chains, the player adds time to their stop timer (up to a maximum of 10 seconds), and sends garbage blocks to their opponent. To break garbage blocks, a player must perform a combo that touches those garbage blocks. When the player does this, the garbage blocks break apart into normal blocks and fall into place. Figure 1 highlights some of the concepts discussed in this paragraph.

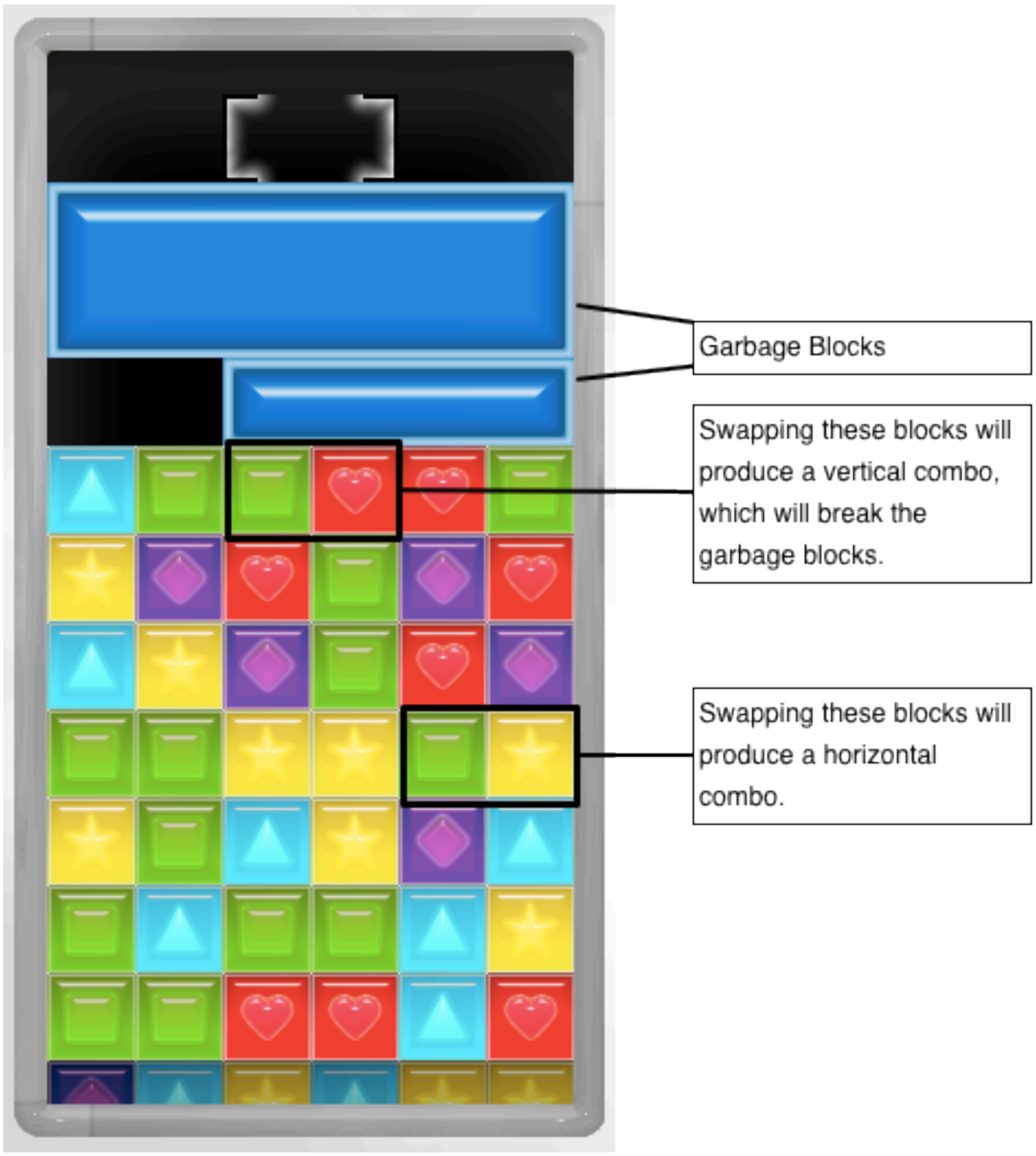
Garbage Blocks
Swapping these blocks will produce a vertical combo, which will break the garbage blocks.
Swapping these blocks will produce a horizontal combo.

Figure 1: A single player's Penguin Attack board.

Penguin Attack is a two-player competitive puzzle game. Because of this, if a second player is unavailable, there isn't really a way for you to play the game. This is the problem that my project addresses, by providing a single player with a computer-controlled opponent.

Overview of Other Parties Involved

Tom wrote most of the actual code for Penguin Attack, and he agreed at the beginning of the semester that he would implement the changes and additions necessary to make the game work with my AI. We were in constant communication throughout the semester, and he regularly provided me with updated builds of Penguin Attack.

It's also worth noting that a small portion of the networking code I used was retooled from the Java AIClient abstract class that my team produced for our Software Engineering project last semester. I had pair programmed this code with Mathew Bergt, and then refactored it significantly after the end of the semester. It seemed perfectly suited to what I was trying to do here, and my team had all agreed that any of us should feel free to use any of the code we generated last semester in future projects. In fact, we released that code under the very liberal BSD-new license. I didn't see any point in re-inventing the wheel, so with some minor tweaks it was adjusted to fit my current use.

Description of the Planning Process That I Used

I didn't do very much initial planning; rather I delved straight to coding using the evolutionary prototype design methodology. The reason for this is that I didn't have a clear picture in my head of what exactly the problems I would need to solve were, much less how I would go about solving them. The problem seemed too large for me to wrap my mind around as a whole, and if I had insisted on hammering all of the details out before writing any code, I doubt I ever would have written a line. Instead for this problem it was easier for me to "break a chunk off" and solve the problem one small piece at a time.

Requirements

Functional Requirements:

When I proposed this project I had two strict requirements:

1. The AI should be a network client, which would connect to Penguin Attack, receive and interpret board states, and be able to generate and send the move sequences necessary to play the game.
2. The AI should play the game competently. I defined this as meaning that it could consistently last for at least three minutes without losing when playing with me as an opponent.

In the process of writing the first generation of my prototype, more specific requirements fell out of those first two requirements. The AI would need methods of finding and executing both vertical and horizontal combos. It would need to break garbage in a timely manner. It needed methods of keeping its block stack relatively level. Finally, it needed methods of keeping as many blocks on the screen as possible so that it didn't corner itself into not having any available moves.

System Requirements:

I planned to develop the AI using Java, simply because it is the language I am most comfortable and familiar with. This may not have been the ideal choice, considering the time sensitive nature of the AI, and the overhead of the Java Virtual Machine (JVM). Since I planned to develop and test this AI on my Asus EEE PC 1000HE, the specs of that machine pretty much dictated the system requirements I was shooting for: an Intel Atom N280 Processor operating at 1.66 GHz with a 667MHz front side bus and one gigabyte of RAM. This hardware is pretty modest, so I was a little bit concerned that it would prove inadequate, especially given the above-mentioned concern about the JVM overhead. Thankfully this was not an issue.

Description of My Design

Network Protocol:

The very first bit of design that needed to be done was to establish a protocol for the AI client and the game to communicate with. In my proposal I had suggested a very simple protocol, and that mostly describes what we stuck with. But Tom and I decided that rather than merely sending the board state information that I could immediately foresee the AI needing, we should instead send a complete board state that describes everything there is to know about the board at that moment in time. We felt this could save us time if I decided that the AI needed more information, as well as with work we might do in the future involving Penguin Attack network clients. Our revised board state message appears as follows:

```
b{
 _   _   _   _   _   _
 _   _   _   _   _   _
 _   _   _   _   _  0n1.
1n1h _   _   _   _  0n5.
1n3h1n4h _   _  0n1.0n4.
2n5h1n2h1n5h0n3.0n4.0n1.
 _   _   _  0n2.0n2.0n5.
 _   _  0n1.0n3.0n3.0n4.
0n5.0n1.0n2.0n4.0n4.0n5.
0n4.0n3.0n5.0n1.0n2.0n1.
0n1.0n3.0n1.0n4.0n4.0n2.
0n1.0n2.0n3.0n3.0n2.0n2.
0n5.0n5.0n4.0n4.0n1.0n1.
0n3q0n3q0n4q0n4q0n1q0n4q
}
```

The first line of this message indicates to the client that a board state is to follow. The last line indicates that the entire board state has been sent. The lines in between represent the current board state, with every four characters representing a space on the board. The first character indicates the chain or garbage depth of the block, an integer between 0 and 9 inclusive, or a space (' ') if that location is not occupied. This data did not end up being used by the AI, so its discussion is beyond the scope of this report. The second character indicates what type of block occupies that space on the board, 'n' for a normal block, 'g' for a garbage block, or '–' for an unoccupied space. The third character represents what color the block is, with each color having arbitrarily been assigned an integer value between 1 and 6 inclusive, or a space (' ') for an unoccupied location. The fourth and final character indicates the state of the block occupying the space (or again a space (' ') for an unoccupied location). There are a large number of possible values to represent the large number of states a block can be in (swapping, breaking, falling, etc.). The only value the AI ever needed to know is if the character is a '.' or not, which indicates that a block is in an idle state.

When this data is parsed, a 12 x 6 (the number of playable spaces on the board) array is populated with elements of ADT PenguinAttackBlock. Each block is constructed with its representative four-character string (discussed above). In this manner the network protocol is encapsulated in the data structure, and the AI itself never has to be aware of the format discussed in the previous paragraph.

Program Structure:

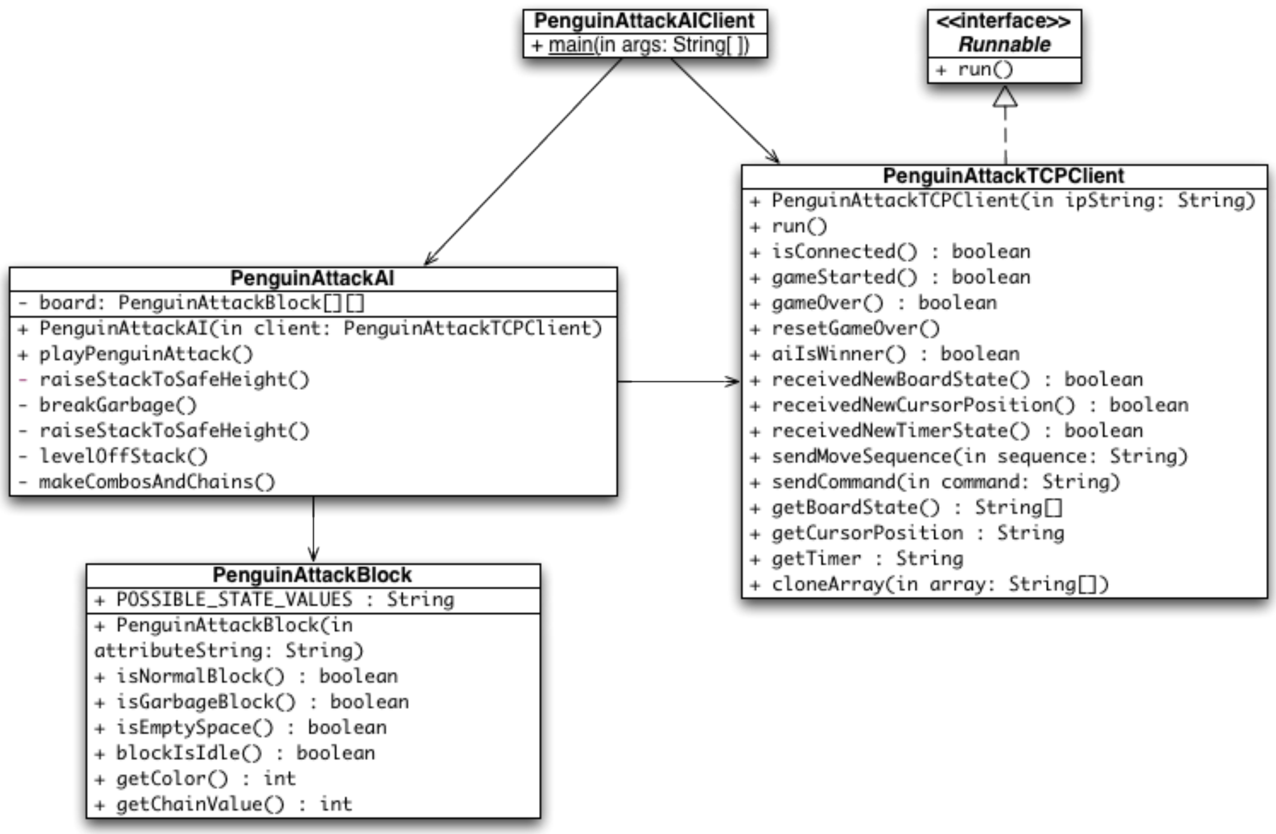
PenguinAttackAIClient
+ main(in args: String[ ])
<<interface>>
Runnable
+ run()
PenguinAttackTCPClient
+ PenguinAttackTCPClient(in ipString: String)
+ run()
+ isConnected() : boolean
+ gameStarted() : boolean
+ gameOver() : boolean
+ resetGameOver()
+ aiIsWinner() : boolean
+ receivedNewBoardState() : boolean
+ receivedNewCursorPosition() : boolean
+ receivedNewTimerState() : boolean
+ sendMoveSequence(in sequence: String)
+ sendCommand(in command: String)
+ getBoardState() : String[]
+ getCursorPosition : String
+ getTimer : String
+ cloneArray(in array: String[])
PenguinAttackAI
- board: PenguinAttackBlock[][]
+ PenguinAttackAI(in client: PenguinAttackTCPClient)
+ playPenguinAttack()
- raiseStackToSafeHeight()
- breakGarbage()
- raiseStackToSafeHeight()
- levelOffStack()
- makeCombosAndChains()
PenguinAttackBlock
+ POSSIBLE_STATE_VALUES : String
+ PenguinAttackBlock(in attributeString: String)
+ isNormalBlock() : boolean
+ isGarbageBlock() : boolean
+ isEmptySpace() : boolean
+ blockIsIdle() : boolean
+ getColor() : int
+ getChainValue() : int

Figure 2: Penguin Attack AI Class Diagram. This diagram shows the public methods and fields of each class, and a few private methods and fields of interest in the class containing the actual AI logic.

As can be seen in the class diagram featured in Figure 2, the AI program is separated into four distinct modules:

1) `PenguinAttackAIClient` – This is where the main method lies. It creates an object of type `PenguinAttackTCPClient`. Assuming the client successfully connects to the server, its method to read from the network is spun off into its own thread (otherwise the program terminates). An object of class `PenguinAttackAI` is then instantiated and given control of the main thread.
2) `PenguinAttackTCPClient` – I quickly realized that the networking code could be encapsulated and hidden from the rest of the program. The public interface to this method consists of methods that let the AI request updates and issue commands.
3) `PenguinAttackAI` – This is the part of the program that actually is the AI. Upon construction it is given a reference to a `PenguinAttackTCPClient` object, from which it requests updates and sends commands. Its main data structure is a 12 x 6 array of ADT `PenguinAttackBlock`. This is also the only module with any algorithms that are worth discussing, and so we shall revisit it in the next section.
4) `PenguinAttackBlock` – This ADT keeps track of the state information that is relevant to an instance of a block. It provides a public interface that encapsulates the network board state portion of the network protocol by allowing the AI to query it with methods such as `isNormalBlock()` or `isGarbageBlock()`.

Algorithms

The algorithms used in this AI are relatively simple to explain, and only slightly more complex to implement. The AI runs through an event loop in which it repeatedly makes the following method calls:

```
raiseStackToSafeHeight();
breakGarbage();
raiseStackToSafeHeight();
levelOffStack();
raiseStackToSafeHeight();
breakGarbage();
raiseStackToSafeHeight();
makeCombosAndChains();
```

Each method is responsible for asking the TCP client to acquire updates to the relevant game state data. The stack needs to be constantly checked to see that it stays raised to the maximum height in which it doesn't start the stop timer. This is because the AI needs to be sure not to "starve" itself of usable blocks. The next most important thing for the AI to do is to break any garbage it might have received. By regularly breaking garbage the AI stops the stack from rising, which gives it time to level the stack and make combos and chains. Lets look at how each of these methods works algorithmically.

`raiseStackToSafeHeight()`: This method first finds where the top of the stack is. There are 12 rows on the board, and the stop timer isn't triggered until the 12th row has blocks in it. So if it finds that the stack is lower than the 11th row, it sends the raise command until this is no longer the case.

`breakGarbage()`: This method starts from the bottom of the stack and proceeds until it reaches the top, searching for garbage blocks. For each one that it finds it looks first for a potential vertical combo that could be used to break it, and, if unsuccessful, then for a horizontal combo that could be used to break it. We will examine the algorithms for finding combos shortly.

`levelOffStack()`: This method starts searching from the bottom of the stack for empty spaces. When it finds one, it looks in the row above that space to see if there are any normal blocks. If so it drags the nearest normal block over the space, and the block fills the space. When it doesn't find any more such spaces, leveling is complete.

`makeCombosAndChains()`: To make a horizontal combo, the AI must simply search for a line that has three blocks of the same color in it. When it finds one, it simply generates the move sequence to drag those blocks together. Making a vertical combo is similar, but you have to find three blocks of the same color in three vertically adjacent rows. The name of this method is currently a bit misleading, as I never successfully designed an algorithm that intentionally makes chains.

Software Development Process

Since I didn't do a lot of ahead-of-time planning, I also didn't follow a very strict schedule. I started coding on the Thursday before spring break. My coding sessions typically lasted for about 12 hours. I had these sessions during all but two days of spring break, and on each remaining Friday of the semester until it was time to give demos. I didn't keep rigorous logs of the time I spent testing, coding and debugging, since I was pretty constantly switching between those three tasks. I would estimate that about 50% of that time was spent testing, 40% was spent coding and 10% was spent debugging.

A fair amount of the time was spent writing code that was ultimately not used. I implemented three completely different leveling algorithms before deciding which one worked best through testing. I also spent a fair amount of time working on algorithms to intentionally make chains, an endeavor that has thus far not been met with any success. I managed to write an algorithm that did an effective job at planning out which combos it would make to build a chain, but combos that were accidentally made when it attempted to execute that chain foiled the algorithm. I tried to write an algorithm that prevented accidental combos, but it just ended up creating new accidental combos.

After my first day of coding I had a (barely) working prototype. Once a prototype was working, I tested the prototype, writing down areas that could use improvement. I would implement those improvements as best I could, and then go back to testing to find areas for further improvement. Every day of programming I ended up with better and better prototypes.

Toward the end of the semester, my classmates reviewed selected portions of my code. The feedback I received was all either rated 'low-severity' or 'question'. The questions I think I answered adequately in class.

One low severity issue was that there were a few places where I was using ‘magic numbers’ which were in fact based on the dimensions of my board array. In many cases I opted to address this issue, but there were a few places in the code where I simply felt that it made the code harder to read. No matter how many years I step away from the project, I don’t think that I will ever forget that the board is 12 x 6, and anyone else writing code for this project would have to be aware of that as well.

Another low-priority issue was that there were some places where I used ‘`i`’ and ‘`j`’ for iteration and others where I used ‘`row`’ and ‘`col`’. In most cases I remedied this issue by using ‘`row`’ and ‘`col`’. There were some places where ‘`row`’ and ‘`col`’ weren’t appropriate. In these cases I attempted to find more suitable names, but there are a few areas where I still use ‘`i`’ and ‘`j`’. It was suggested that I could consider using `foreach` loops to remedy the issue. I still needed iteration variables to access indices of the array, however, so this suggestion didn’t really solve the issue.

There was a complaint about my initializing an iteration variable outside of a `for` loop. I considered this a non-issue.

It was also suggested that I use enumeration for storing block colors. I considered this, but in practice the only time the color code is used is in a manner in which this would not affect readability (`if(thisBlock.getColor() == thatBlock.getColor())`).

Finally someone pointed out that I usually used `this.board`, but that there were a few instances where I didn’t. I remedied this issue.

Analysis, Results, Discussion

Overall I am reasonably happy with the code that I am turning in. While I never got the AI creating intentional chains, it creates enough accidentally that it still ends up being a formidable opponent. It regularly lasts against me as an opponent for over 15 minutes in a round, easily beating the goals I laid down in my proposal. I have some ideas about how to overcome the difficulties I had creating a chaining algorithm, and I am eager to work on implementing those ideas.

The AI is fast, it’s mesmerizing to watch, and it provides enough challenge to make me want to keep playing with it. The overhead of the JVM was a non-issue even on modest hardware. In fact, I could run several instances of they AI on my EEE PC without it breaking a sweat.

There remain a couple of bugs that present themselves extremely rarely, and so have been difficult to nail down. If I play against the AI for an hour, there’s a chance I might see one of them. This made them extremely difficult to try and isolate. I’m not sure if the bugs are related, and have only the vaguest of suspicions as to what could be causing them. They might not even be in my code; they might be in Tom’s code. They present themselves so rarely that I don’t consider them to be showstoppers, especially since the only result is that the AI loses the particular round in which they are encountered.

The leveling algorithm still has some room for improvement. When a block that the AI wants to move to fill a space is on the same line as a block of garbage, the AI attempts (in vain) to drag that block across the garbage. There are also a few sets of particular circumstances where it doesn't make the stack completely level. These are extremely minor issues, and so I kept putting them aside. They shouldn't be very hard to fix, but it always seemed as if there was something more important to work on. I will get around to fixing them, but it will be after the semester is over.

The AI wastes a lot of time waiting for swaps to take place. I imagine that some sort of useful analysis could be performed while the AI waits, but I haven't quite determined what that analysis is. This is yet another part of the project that I intend to work on after the semester.

Conclusions and Lessons Learned

I'm really glad that I chose this as my semester project. I had a lot of fun writing it, and am very happy with the results. I felt like it gave me the opportunity to employ skills and knowledge I have acquired across all of my CS courses; Algorithms, data structures, networking, AI, threading, object-oriented design, etc. They all played a role in my solution.

Most of what I got out of doing this project was an opportunity to employ the knowledge I have gained in a wide variety of CS related topics, and experience working on a project that was of a more 'real-world' scale (albeit still fairly small) than you typically work on in a CS class.

I suppose if there was a major lesson I learned from this project, it is that some problems are conducive to solving directly in code, and some problems are not. I think I would have met more success designing a chaining algorithm if I had sat down with a pen and paper to work out a solution to the problem instead of trying to code a solution directly.

References:

The only place I referred to during this entire project is Oracle’s Java documentation at http://download.oracle.com/javase/6/docs/api/index.html.

Appendix A: User Manual

Minimum System Requirements: This will probably run just fine on anything that meets the system requirements of Java, but the minimum I have tested it on is an Intel Atom N280 Processor operating at 1.66 GHz with a 667MHz front side bus and one gigabyte of RAM. It was designed and tested using Java SE 6.

Since Penguin Attack has not actually been publically released yet, these instructions aren't likely to be very useful to anyone for the moment. For the time being if you need to see the AI in action, probably the easiest thing to do is contact me (jcbuck@uaa.alaska.edu) and arrange an on-site demonstration.

Once we release Penguin Attack as an easy-to-install binary, the usage of the AI becomes simple:

1) Using a command line interface, navigate to the location of the `PenguinAttackAIClient.java` file.
2) Type '`javac PenguinAttackAIClient.java`'.
3) Type '`java PenguinAttackAIClient <IP Address>`', replacing `<IP Address>` with the IP address of the machine running Penguin Attack, or if the AI and Penguin Attack are running on the same machine, simply leave it blank.