Tiny Wings with Concurrent AI Players

A Study of the Benefits of Multithreading the AI in Real Time Games

Jamie Miller and Lindsey Tubbs | April 2015 | Concurrent Programming

Introduction

When making a computer game it is often fun to include an opponent controlled by artificial intelligence (AI) to compete with the human player. In a live action game the game state is typically updated a certain number of times per second, often referred to as frames per second or (FPS). In each frame, user input is read and AI algorithms are invoked to get everyone's decisions and then compute what happened and redraw the screen. A high enough FPS will result in a smooth animation that the player will interpret as live action, typically 30-60 FPS is desired, and is dependent on the maximum refresh rate of the physical screen and the processing capability of the computer. Making it possible to run a game at higher FPS with cheaper hardware is highly desirable since not everyone can afford a fancy computer and selling a game to a wider audience earns more money.

Running the AI algorithms concurrently has a high potential for increasing FPS, and is naturally parallelizable since each AI makes decisions relatively independently in most games. Many games don't run the AI concurrently, high budget ones that do usually block until all AI algorithms are done before moving to the next frame. Some of this has historical roots, many general AI algorithms were developed for turn based games like chess, which aren't all that applicable for a live action game. Our project will investigate the potential for holding the AI to the same standard as a human player, in that we will send them a copy of the screen but we don't care at all if they take too long to respond.

The primary problem this weaker guarantee introduces is the possibility that an entire frame could pass without the AI having a chance to respond. If this happens too much it could make it look unresponsive to the player. On the other hand, for certain games, this problem could actually be a desirable feature. It is a bit unrealistic that the AI always has perfect response time. It introduces purely accidental timing mistakes by the AI which could potentially make it more realistic and fun for the player.

We chose the popular phone game Tiny Wings for our project, since it is a relatively simple live action game with few controls and we were unhappy with how unrealistically precisely timed the AI opponents are in the default game.

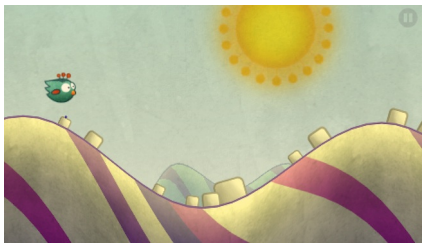

Figure 1: Example of Tiny Wings in single player mode.

Game Mechanics

The premise of the game is that you are a bird with wings that are too tiny to fly, but by dropping at precise times onto downward slopes you can speed up enough to propel oneself through the air for a short time. The object of the game is to travel as quickly as possible, competing with other birds. See Figure 2 for an idea of where you want to aim the bird to travel the fastest.

Figure 2: Best landing spots for the fastest speed in the game

The controls for the game are as simple as possible, a single boolean that represents if you want to drop now or not. On the phone this is represented by pressing the screen or not.

We made a simplified desktop version of the game in Java. Among our simplifications were removal of many obstacles, power ups, removal of the score system, and extremely simple graphics.

Concurrent AI Implementation

We wanted the AI in this project is to be more realistic to what a human would do. Therefore, we had the AI use a screen like the player would see to find when it should press so that all players including both the real and AI players have the same information. Additionally the AI in these situations does not have predetermined actions, meaning that the AI must make all of the calculations in real time like the player does.This makes it necessary for the AI to make quick calculations that are good enough rather than very precise. It must also adjust and account for mistakes and frames that it has missed. This results in a program that uses the lack of real time guarantee as its randomness factor so that the AI is more realistic than other games.

For our implementation we created a common interface for all Birds either Player or AI, which will be given a unique screen to view and a boolean to flip on and off. The screens are single writer multiple reader and the booleans are single reader single writer. The main thread

will start up all the bird threads, write the screens, and read the booleans. Each bird thread will only read the screen and write the boolean.

Screens are implemented with AtomicStampedReference<BufferedImage>. BufferedImage is a convenient way to represent an image that can be drawn to with graphics primitives and read from each pixel. AtomicStampedReference was chosen because we wanted to include a frame number with each screen, and that was an easy way to ensure they are set and read at the same time atomically.

Originally we thought we would need to ensure that even though each bird had an individual screen to read since they had different views of the world, that one bird didn't get an unfair advantage by being given its screen first. This problem would arise since the main thread updates each bird's screen in sequence. The solution we came up with was to force each bird to check all the other bird's stamps to ensure they are greater than or equal to its own stamp before using that screen. We have a wait free implementation of that, as we discussed at progress reports, but we found that it caused all of the birds to synchronize which stamps they were responding to and to do so all at the same time. We removed it because of that problem, and despite it being technically incorrect, we doubt that it really made it that unfair since the scheduler has many opportunities to be unfair to particular birds. Scheduling one bird before another after the screens are updated is just as bad as not giving the screen to one of them for a while. Interestingly, when playing the game, it feels like the yellow bird tends to do the best at the race even though it is given its screen last.

Non-Concurrent Implementation

To contrast the performance and correctness of the concurrent implementation, we also made a non-concurrent implementation that waits for each AI to complete in sequence between each frame.

Performance Measurement

First we wanted to see if the concurrent version was indeed faster than the non-concurrent version. To do this we ran each of them at various FPS and counted how many times it failed to generate the frames fast enough. When the program isn't able to keep up with advertized frame rates the effect is a very noticeable choppiness that is undesirable. Generally one tries to keep the FPS high but not so high that the program can't keep up. "18ms AI" refers to the fact that our AI implementation on average takes 18ms to process and respond to a single frame.

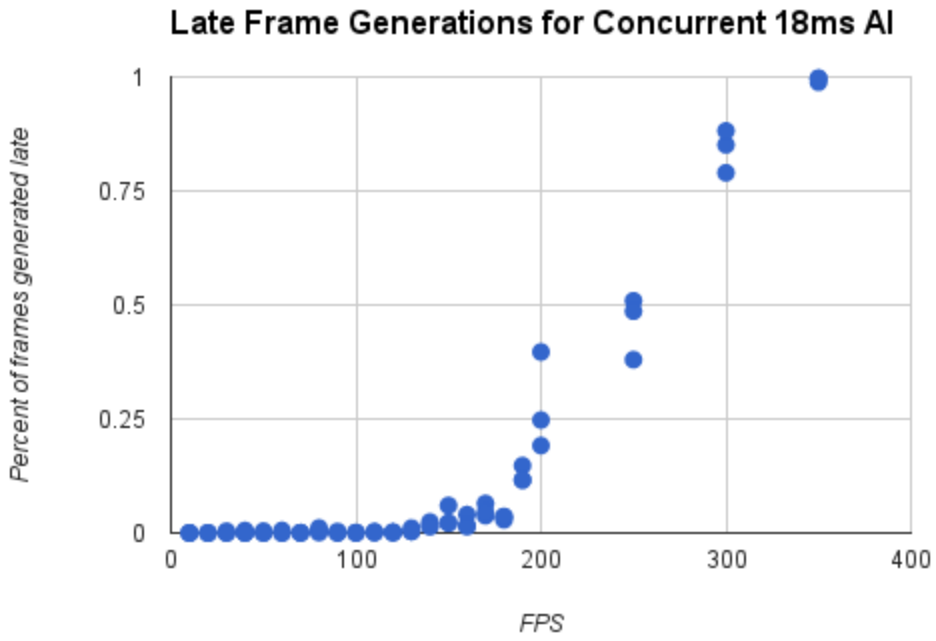

Figure 3: Percent of frames generated late for the concurrent implementation.

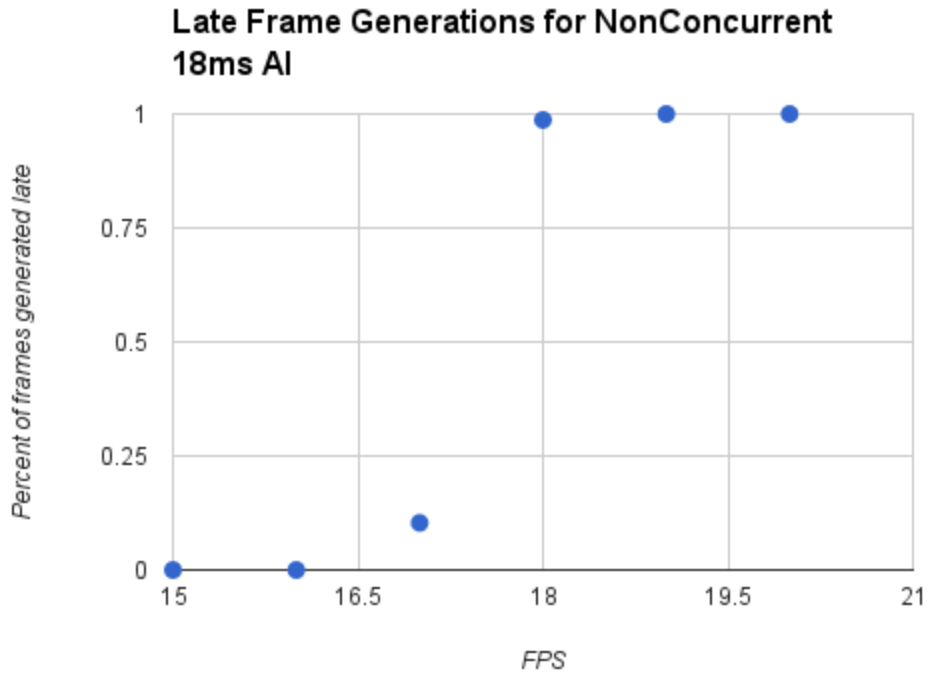

Figure 4: Percent of frames generated late for the non-concurrent implementation.

From this data we can conclude that the concurrent implementation will start to break down at about 150 FPS and can be safely run at 120 FPS without concern, and the non-concurrent implementation will start to break down at 17 FPS and can be safely run at 16 FPS without concern. Of interest to note is that a perfect non-concurrent implementation could never do better than 18.52 FPS if there are three AI birds that take 18ms each to process a

frame. (3*18ms = 54ms, 1Frame/54ms = 18.52 FPS). Another implementation that was discussed is using multiple threads but still requiring each bird to respond for each frame, the theoretical upper bound on FPS for that would be 55.56 FPS (1Frame/18ms = 55.56 FPS), so the 120 FPS of our concurrent version with no real time guarantee allows for a massive increase in performance.

Correctness

A performance increase of that magnitude raises questions about if the AI players still act intelligently. To measure this we had the AI compare each stamp from the AtomicStampedInteger containing the screen to see if any numbers were skipped. When the AI realizes a frame was skipped it will update a counter. Each bird had its own counter that are summed together at the end to get the total missed frames.

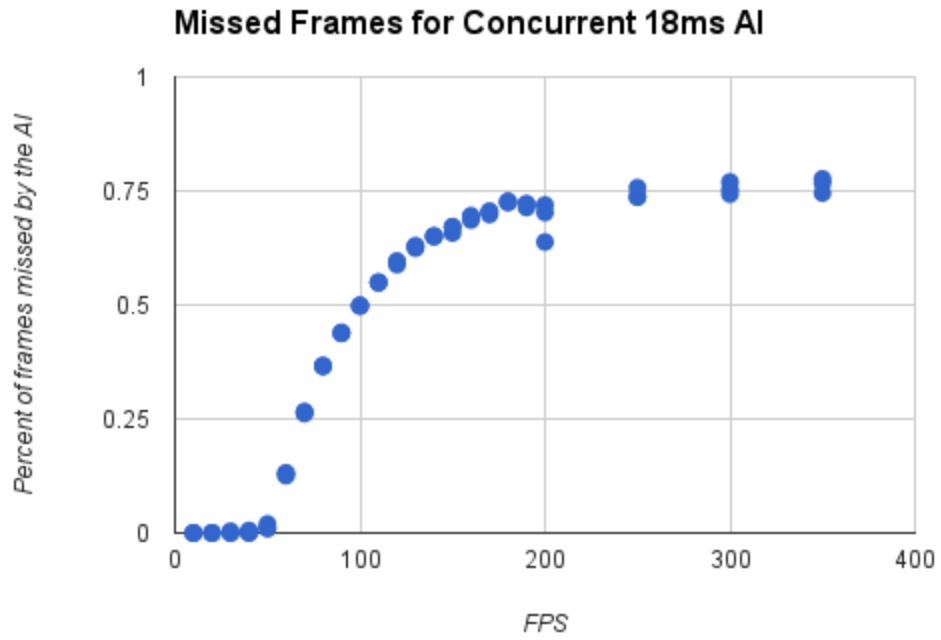

Figure 5: Percent of frames missed by the AI.

These results appear bad but not entirely unexpected. Note that the theoretical maximum for zero missed frames for a concurrent implementation is 55.56 FPS. After that, the percentage of frames missed increases rapidly till most frames are missed. While it appears to level off around 75% at high FPS, in actuality it is still upwardly sloped. Regardless, in order to run at 120 FPS like we would like, we'd have to be prepared for the AI to miss well over half of the frames.

But how bad is that? For the purposes of our game, one of our hopes was that we would get a more human like reaction time from the AI players. According to humanbenchmark.com[1] the average reaction time for a human to see something on a screen and press a button is 265ms. At 120 FPS, the AI skipped 66.67% of the frames, or responded

[1] http://www.humanbenchmark.com/tests/reactiontime/statistics April 2015

to 40 frames each second. At 40 frames a second, the AI has a reaction time of 25ms, still over ten times faster than an average human. By our definition, this AI is still correct, with a ton of margin for error.

Discussion / Conclusion

For some games, it may not be desirable to write AI this way since it makes them nondeterministic and without real time guarantees. In the case of live action games where AI opponents are meant to act like human players, it may have a lot of advantages. We've shown that for the game Tiny Wings, writing the AI in this way allowed for running the game at higher FPS with more complex AI and with more realistic mistakes made by the AI. This could be generalized for any live action game with opponents that are meant to act like humans. The main ingredients are some knowledge about the world published where the AI threads can see it, and the ability for the AI threads to make a decision and act upon that knowledge at their leisure. We find that requiring the AI to respond by a certain time will only slow down the game and reduce the realism of the AI in such situations.

Potential problems to investigate in the future is if the game changes difficulty significantly on different computers. Normally it wouldn't since the AI is guaranteed to respond for each frame, but now we are using the operating system's scheduler to introduce randomness so there is some risk, likely imperceptible to a casual player. Additionally we found the addition of image processing to be extremely CPU intensive. While it worked well for this report for the purposes of slowing down the AI and making sure it has no information a human player wouldn't have, running the CPUs that hard wouldn't usually make sense for a simple game like Tiny Wings, particularly on a phone.