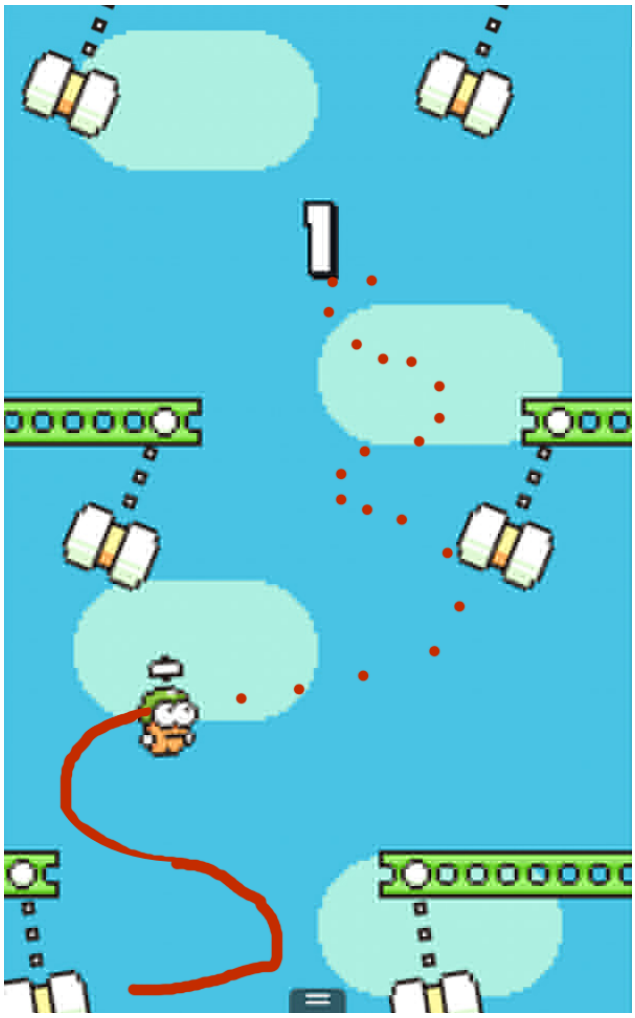
CS 4100 Semester Project:
Creating Autonomous Swing Copters using RL techniques

December 12, 2014

Jared Miller & Chris Beiser

1 Project Description

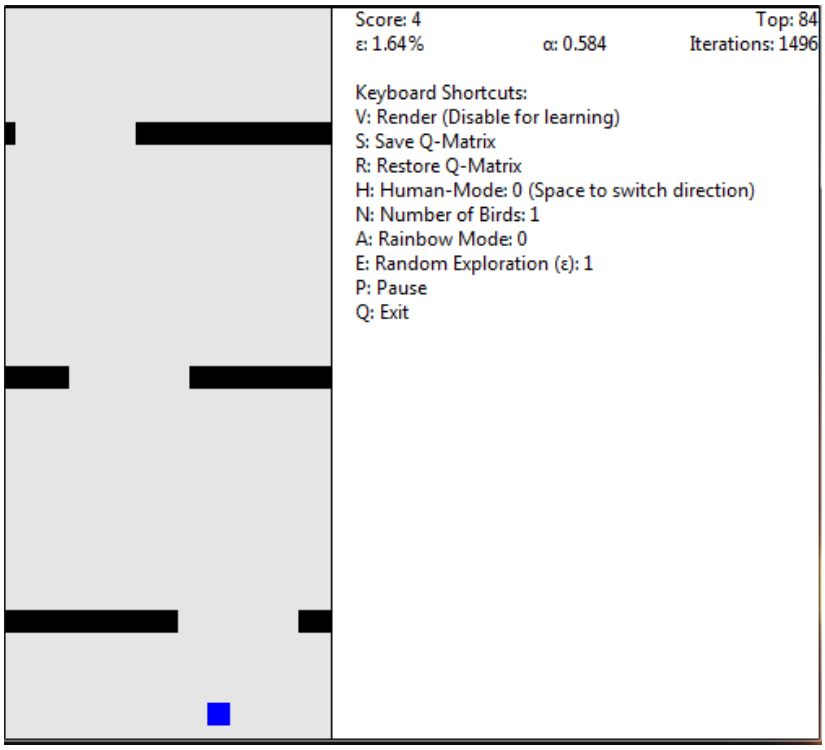
Swing Copters is the second popular and frustrating game created by Nguyễn Hà Đông (Dong Nguyen), on the heels of the wildly successful Flappy Bird. In Swing Copters, players control a tiny helicopter which moving upwards at a constant rate, and moving while constantly accelerating to either the left or right. Tapping the screen toggles the direction of the helicopter's acceleration between left and right, but leaves the magnitude unchanged. The helicopter must pass through gaps and dodge swinging hammers. As a result, the helicopter has a somewhat parabolic path, as superimposed in red on the screenshot below. The game is scored based on the number of gaps successfully passed through.



We re-implemented a simplified version of Swing Copters, with no hanging hammers, but with the accelerative physics largely unchanged, and similar scoring. Then, we wrote a Reinforcement Learning agent to learn to play our representation of Swing Copters. To play the game manually, press the "H" key on the keyboard to switch into Human Mode, in which you can toggle the direction with the spacebar. In Human Mode, no learning data is stored, since Q learning is unsupervised.

All work was done in Python 2.7.x. We used Tkinter as a front end to display the game. No logic or collision detection was done in Tkinter, so we were able to turn off rendering in order to increase the speed for learning. This is accessible by pressing "V" on the keyboard. Numpy was used to facilitate multidimensional array computation and storage of the Q matrix.

2. Algorithms



The final version of our simulation.

We used Q-Learning in order to train our agent. Q-learning is a model-free Reinforcement Learning technique that, while slow to converge, will eventually form an optimal policy. Reinforcement learning is particularly well suited to situations where there's an underlying MDP, which is the case in Swing-Copters. Further, reinforcement learning is especially well suited to deterministic situations. While Swing-Copters is deterministic, in both our implementation and the original, after discretizing the state space, with a given space, the next space is not deterministic. However, the shrunken state space's behavior is very easily approximated as a stochastic MDP.

We discretized the state space over vertical distance to the next gap, horizontal distance to the next gap, the acceleration and velocity of the helicopter, and the x-position of the helicopter. Velocity was quadratically distributed, all other quantities were linearly partitioned or were a boolean. There are two possible actions: flip the direction of acceleration or wait. In total, the size of the state-action space was 2*2*12*11*8*5 = 21,120 elements. The state space was experimentally determined to be large enough to have positive effects while remaining small enough to be quickly calculable.

Each frame, a positive reward that decreases linearly with distance from the center of the next gap is given if the copter stayed alive, and a large negative reward is given if the copter dies. This results in the copter gaining feedback on the proper place to be much more quickly than if they were to only gain feedback based on death. Once the agent made a decision whether to flip the direction of its acceleration or not, the agent would move and all of the walls would fall at a constant downwards velocity (6 pixels/frame). The agent only has knowledge of the nearest wall and not of future walls.

We also included a terminal (maximal) velocity that is less than the true maximum velocity (if the agent started from rest on one side and constantly accelerated to the other) to further challenge the agent.

We use a Decreasing Epsilon-Greedy strategy to manage the tradeoff between exploration and exploitation. Every frame, there is a ε chance that a random action is taken between flipping acceleration and waiting, where ε = 0.03 to start. This strategy is typically used to maximize exploration of the state-space early, and move towards more optimal behavior later. We use a relatively low epsilon value to start because of the high number of independent choices made between each result; with an overly high epsilon, it would become difficult to see any difference between choices occurring sooner after passing a gap.

Exponential simulated annealing is done on α and ε, such after 1,000 iterations ε goes from 3% to 2%, and after 10,000 iterations α goes from 0.6 to 0.5. ε can be toggled to between zero and the value from simulated annealing by pressing the "E" key, in order to quickly test the capabilities of the copter while training. Our initial learning rate (α) is 0.6 to slightly emphasize newer results, and our discount factor (λ) is 0.98 in order for the system to have near-permanent memory but still converge.

3. Results

Q learning is known for being a very slow algorithm, especially λ is very close to 1. There is high variability, but scores are definitely related to the number of iterations; the more iterations are evaluated, the better the agent is at playing the game. After 3000 iterations, an agent reached a score of 92, and after 4400 iterations, an agent reached a score of 110. Both of these results were achieved on the order of 1-2 hours. On another run of the program, an agent reached a high score of 253 after 3200 iterations. After 7200 iterations, the agent was averaging over 40 gates passed per run, much better than a typical human player, with a high score of 368. We believe that letting the program run for multiple hours or days would result in an agent that follows something very close to the optimal policy.

We found that having multiple agents playing the same game was actually detrimental to learning, and single agents are best when developing a policy. When there were multiple agents, the agents were iterated through and would all run on the same world state. Effective parallelization of reinforcement learning is a very difficult problem and

was outside the scope of this project. We could have improved performance by modifying the reinforcement learning algorithm used, such as implementing Speedy-Q Learning or Greedy Q learning.