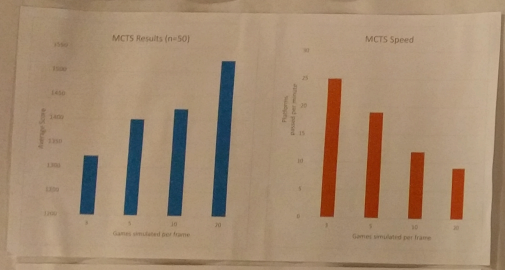
Monte Carlo Tree Search (MCTS)

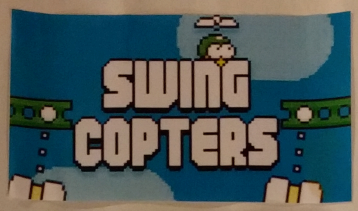
One difficult aspect of Swing Copters is that the player can only control the direction of the avatar's acceleration. All actions produce magnified differences in the avatar's future position. To be successful a player must plan ahead.

We wanted to factor in these delayed changes in position by seeing how the game would play out given a particular action. MCTS is well suited for this because it estimates the reward of a particular action by performing a series of simulated games.

Another aspect of Swing Copters that makes MCTS a worthwhile approach is that consistent short term survival leads to success. A player using MCTS computes the expected reward by averaging randomly played games. In Swing Copters, although random play will not lead to long term success, it can give a good idea of the likelihood of short term survival.



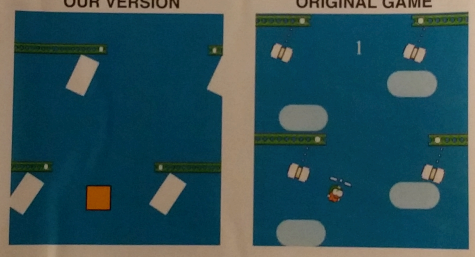
The MCTS algorithm produced a mediocre Swing Copters player. The player regularly navigated through a few platforms, but was not consistently successful, and rarely scored better than a practiced human player. Scores improved slightly with more simulated games. However, the computation time required increased drastically, making it impractical as a way to play the game in real time.

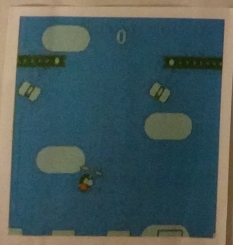


Swing Copters – The Game

The goal of Swing Copters is to keep the avatar alive as long as possible. The avatar dies when he collides with a platform, a hammer, or either edge of the screen (which we will call a "wall"). A player has one way to control the avatar's trajectory – clicking to reverse the avatar's acceleration.

Despite being a simple game, it is surprisingly difficult to play. Beginning players can rarely even make it past the first platform.





Screenshot of Swing Copters after performing K-means clustering ($K = 10$). Averaging the locations of the pixels in the orange cluster allow us to accurately estimate the avatar's location.

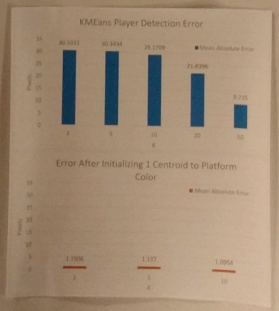
Avatar Detection

Originally we wanted to create a player that could play Swing Copters on an android emulator using a series of screenshots from the game as input. Due to time constraints and complicated nature of working with emulators, we decided to implement our own version of the game instead. However, we also implemented a proof of concept for how a player could deduce information about the current game state from a screenshot of the game.

K-Means Avatar Detection:

Using K-Means we split the game screenshots into clusters of similar colors. Then, we averaged the location of the pixels in the color most similar to the avatar to get an estimated location.

By forcing one of the clusters to be similar to the avatar's distinctive orange color, and initializing one of the clusters to a green similar to the platforms, we were able to bring the error down to within an average of 1 pixel.



Q-Learning

Define features $\varphi(s, a)$ and weights $\mathbf{w}$

$Q_{\varphi\mathbf{w}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \varphi(s, a)$

On each (s, a, r, s'):

$\mathbf{w} = \mathbf{w} - \eta[Q_{\varphi\mathbf{w}}(s, a; \mathbf{w}) - (r + \gamma * V_{\varphi\mathbf{w}}(s'))]\varphi(s, a)$

Another approach we took to make Swing Copters player was using reinforcement learning. The biggest challenge to this approach was coming up with useful features for the problem.

Features

Interesting Terms We Considered

- ~ absolute x distance to gap
- ~ absolute velocity
- ~ vel times acceleration
- ~ x distance to gap times acceleration for various different velocities
- ~ x distance to the platform it's facing
- ~ x distance to the nearest wall
- ~ x distance to the wall it's facing
- ~ x distance to the closest wall
- ~ distance from center times velocity times acceleration
- ~ signed distance to the gap between platforms times acceleration
- ~ signed x distance to the gap between platforms times acceleration

For Q Learning we need indicator features

1) We can create indicators for whether a term is between two values or larger / smaller than a set value (discretization)
2) Few terms are inherently worse or better; the desired value of most distances, velocities, accelerations, and even combinations of them depends hugely on many factors.

Key Questions to Develop Useful Features

- Do we have enough time to stop before hitting the wall we are moving towards?
- Do we have enough time to stop before passing the platform gap?
- Do we have enough time to avoid hitting the hammers?

The first features that improved our results significantly were indicators on the avatar's "stopping distance" from a wall – how close to a wall the avatar would be if the player tried to stop immediately. These features allowed our player to quickly learn how to avoid crashing into the walls (sides of the playing area).

Next, we added a feature indicating whether the avatar could stop before making contact with the oncoming platform. With the combination of these features our player's performance was comparable to that of the MCTS player.

Finally, we included several features corresponding to the avatar's ability to stop within varying distances of the oncoming platform and wall as well as a feature describing whether the avatar had enough vertical space to make it through the gap between platforms.

Once we implemented a set of appropriate features, we had great success using Q-Learning. Our player now learns perfect play that allows it to survive indefinitely.

Features	Survival Time (frames)
stopping distance from wall	577.45
stopping distance from wall + platform	1297.3
stopping distance from wall + platform (multiple distances)	indefinite

Monisha White and Nolan Walsh