Feynman Robot
(Black Hexapod)

Project 1 - Fall 2019

Meshal Albaiz - Tristan Cunderla
Brian Henson - Aron Schwartz

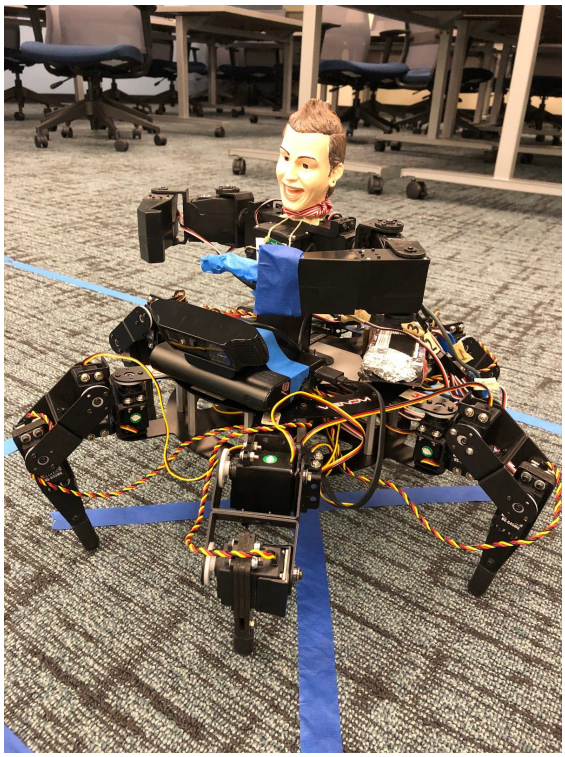
Table of Contents

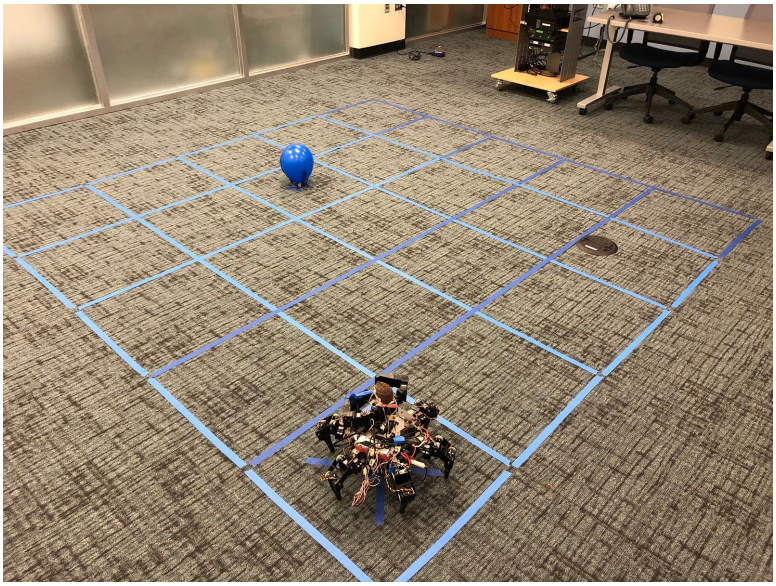
Project Description

The first project with the Feynman robot is to create an angel and demon quantum controlled game. In this game, the demon wants to detonate a bomb and the angel is trying to prevent that. The robot reads the light with a webcam and sends the light levels to the quantum circuit along with the angel/demon turn. The quantum circuit returns probabilities of the what action the robot will take, it either follows the command of the player or disobeys. Based on player control, the angel can move up or make no move, the demon can move up to the right or move right. The angel wins if it moves above the bomb on the y-axis or beyond it on the x-axis, it performs a dance when it wins. The demon wins when it goes to the bomb, it pops a balloon with a pin attached to its arm. Each player is allowed 5 moves, when both players run out of moves the angel wins the game. The number of moves allowed is configurable.

The project goals were achieved after being built on fundamental steps as follows. Testing complete functionality of the robot as it was developed by previous groups. The code was mostly functional, there were quite a few Python packages that needed to be installed, and some adjustments and documentation of the current state of the robot. Next, the game rules and logic were created, the logic was created based on the game rules as described above, and it includes a 2D virtual map of the game. The game code was integrated with the quantum circuit in order to determine whether the robot follows commands or not. The quantum circuit was created in a different module, it takes 3 bits in, 2-bit value of light level from the sensor input (webcam), and 1-bit boolean of who's turn it is, it runs the quantum circuit 1000 times and gathers the probabilities of the decision. The robot either obeys or disobeys the players command based on the probabilities produced by the quantum circuit. The robot checks the virtual map and determines whether it can move, whether it hits the bomb, or out of bounds. It then updates the robot position accordingly.

Pictures of Feynman Robot and The Game Board





Testing the Functionality of Prior Developments

This robot was developed by other groups in the past, so testing and verifying the functionality of the robot is a very necessary part of the project. The movement and servo functionality was tested using the prior group's project demo code (testing/project1_demo.py), it included movements of the hexapod body and the torso. The code and body of the robot was completely functional and there were no issues getting it to move. The code of the torso movements code was functional but a few servos were not connected correctly, one servo was connected to the wrong channel and another servo had no connection at all, both have been fixed. The legs' range of movement can be tested using (testing/manual_calibration.py).

The Pi camera was not functional, the most likely cause is the ribbon cable being bent, in order to fix that the hexapod would need to mostly be disassembled to reach the Raspberry Pi. This group is using a USB webcam as an alternative to detect light. The object detection software developed is a simple color detection script, it works well sometimes but it's not very reliable, it is not being used in this project.

The Raspberry Pi was functional but missing a lot of the libraries needed to run the existing code and newly created code. There is no way to connect to the internet on campus so the group had to take the SD card back home to another Pi and install everything needed. The servo HATs seem functional, all servos are able to run and no defects were detected.

The power supply for the servos is functional as is, but when running all the servos the capacity is limited. The battery is drained within 30-60 minutes of use, and takes twice as long to charge up. A great improvement for future projects would be to increase the battery capacity for the servos.

The servos for the right arm and left legs are connected to the top PWM hat, while the servos for the left arm, right legs, and waist are connected to the bottom PWM hat. The specific channel used for each servo is documented and configurable in (project_files/robot_drivers/hex_walker_constants.py), along with the PWM calibration values for each individual servo and other useful tuneable values. To change which hat a set of servos is connected to, the relevant Leg/Rotator object must be initialized with the different hat (in the main top-level file).

Game Logic and Virtual Game Map

The top level file (Angel_Demon_Game/angel_demon.py) handles the logic of the angel demon game (such as 2D grid tracking, win conditions, etc) as well as handles the integration with the other components of the decision making system (such as light levels and quantum randomness) to determine the ultimate move by the hexapod bot.

The game code is implemented as a python object to allow easy configuration of game settings. Different grid sizes, bot starting position, and bomb position can be easily changed to allow for a large variety of game setups and outcomes, by simply changing the variable values in Angel_Demon_Game.__init__().

```
#Game board initialization
self.game_width = 5
self.game_height = 5
self.game_grid = [["   " for x in range(self.game_width)]

#Initialize the bomb position
self.bomb_y_pos = 0
self.bomb_x_pos = 4
self.game_grid[self.bomb_y_pos][self.bomb_x_pos] = "BOMB"
```

Figure 1: Top level game code allows for custom board size and bomb position

The initial player (angel or demon) is chosen randomly and prompted to select a valid move. After accepting the user's desired move, the light levels are captured and passed into the quantum circuit. The light levels are returned as a 2-bit vector and combined with the turn bit to make a 3 bit vector in total. [TURN_BIT, LIGHT_BIT_0, LIGHT_BIT_1]

The quantum result is returned to the game code as an integer code, and this code is then analyzed to make the proper decision on the physical bot. The combination of all factors allows the possibility of the robot "obeying", "disobeying neutrally", and "disobeying maliciously" as possible outcomes, corresponding to moves that could be desired or potentially disastrous for a given player. The quantum output and light levels are combined such that darker conditions raise the chance of the bot listening to the devil, while brighter conditions raise the chance of it listening to the angel. A snippet of the "decision" game code can be seen below:

```
if (quantum == 0):
    print("Devil successfully tells bot to move up-right!")

    #move_status = self.move_board_bot_up_right()
    move_status = self.move_board_bot_up()

    #Return code of '1' means we hit the bomb
    if (move_status == 1):
        torso.stab(stab_angle ,1)
        self.move_hexapod("UP")
        self.devil_victory = True
        break
    elif(move_status == 2):
        print("Out of bounds move! Bot staying still")
    else:
        self.move_hexapod("UP")
```

Figure 2: The quantum integer result is interpreted to execute the appropriate action

By virtually tracking the internal game grid, decisions regarding the validity and results of moves can be checked prior to sending the actual PWM signals to the servos. For example, if a move is determined to result in out-of-bounds, the internal tracker will detect this and prevent the move entirely before the physical bot moves. In general, the internal grid tracking has the purpose of monitoring whether the angel or demon has won and executes the appropriate victory moves and “bomb popping” pose as needed. The devil can win by popping the balloon, and the angel can win by either preventing this possibility (such as moving to an area where the devil can never reach the bomb), or by surviving through the last move. These conditions as well as other minor logic (such as turn tracking, etc) are smoothly handled in the game object into a complete playable program. A screenshot of the 2D grid tracking and game code output can be seen below.

```
********* Starting turn  0  **********

****** Devils Turn *****

[['    ' '    ' '    ' '    ' 'BOMB']
 ['    ' '    ' '    ' '    ' '    ']
 ['    ' '    ' '    ' '    ' '    ']
 ['    ' '    ' '    ' '    ' '    ']
 ['BOT' '    ' '    ' '    ' '    ']]

1 - Right
2 - Up-Right
Enter choice: 1
```

Figure 3: Game grid is displayed on each move along with valid options for user

Driver Updates: Interpolation & Threading

The group was unsatisfied by the jerky motion of the robot in its initial condition, so improving its motion was always one of the group's goals. After briefly exploring what would be needed for ROS, the group determined that it was best for "large-scale" parallelism and communication across multiple platforms, and unsuitable for small lightweight parallelism like the group wanted. Also, it would require significantly changing the way the group launched and interacted with the running program. So, the group instead went with Python threading.

The servos do not support any kind of speed control: when commanded, they simply move to their new position as fast as possible and hold it until given a new position. This is a fundamental hardware limitation that the group cannot circumvent.

The existing driver infrastructure used "pose"-based animation, where each "pose" was a set of positions for each servo of each leg of the hexwalker (or torso). The driver functions simply set the value of all servos in the robot to match a given pose, then waited for a static amount (default 0.2s) before doing the same for the next pose. This caused very jerky movement because of all the starting and stopping. To solve this problem while preserving the pose-based animation approach, the group chose to use rapid interpolation. By breaking up any given pose transition into several intermediate "frames", the overall movement causes much less shaking and is visually smoother and closer to linear. This linearity also lets the robot move in "slow motion" between two poses, a feature that was impossible with the previous drivers where it moved between poses only as fast as the servos could move.

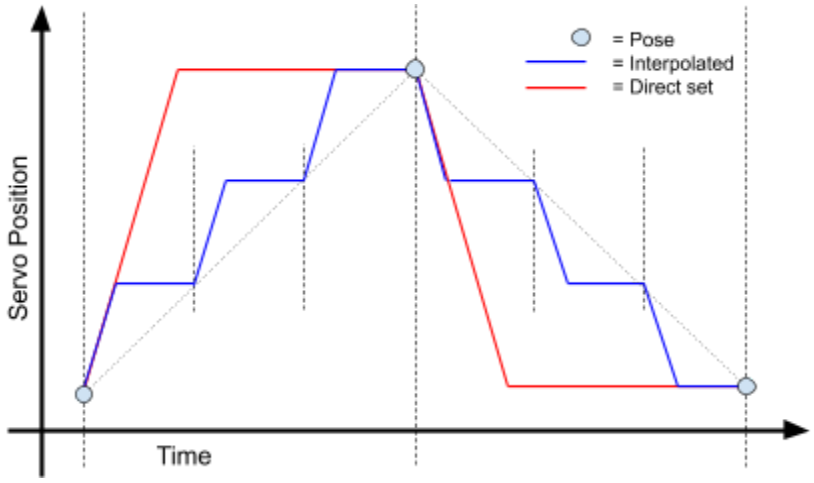


Figure 4: This shows how interpolation increases the linearity of the motion, even though the rate of change of the servo's position is the same across methods.

To implement this, each leg has an independent thread running the function “Frame_Thread_Func()” from (project_files/robot_drivers/leg_thread.py) that has access to that leg’s members and functions. Communication is done via the running_flag/sleeping_flag and the frame_queue. When the leg is given a new destination pose and a duration, the motion from the current position to the destination pose is interpolated into several frames that are appended onto the frame_queue. The number of frames varies depending on the requested duration. The time between frames is constant, defined by INTERPOLATE_TIME in “hex_walker_constants.py”. The frame thread is reading that queue, consuming the frames in the queue and deleting them as it goes. For each one it sets the servos to the appropriate values, then sleeps for the frame-duration, in a repeating loop until the frame queue is empty. Once the frame queue is empty, the leg is holding the position specified by the original pose. At the leg level, the system also supports adding more frames onto the queue (for the following pose transition) before the first transition has even finished, as well as supporting variable lengths of time between poses, but higher levels of the program don’t yet use these features.

The old non-threading approach is handled in (project_files/robot_drivers/hex_walker_driver_v2.py) by using functions Leg.set_servo_angle() and Leg.set_leg_position(). The new interpolating approach is handled by Leg.set_servo_angle_thread() and Leg.set_leg_position_thread(). The Rotator class was changed to be a subclass of the Leg class, and inherits these same functions.

As a result of these changes, the group needed to make several points of the drivers thread-safe. The Python threading library defines a threading.Lock() object which works as a mutex, and the group used several of these in the drivers to protect the frame_queue for each leg (accessed by the leg’s thread and the main thread) and some other members of the leg object, to ensure operations with these members were atomic. Similarly, because the Adafruit PWM hat is actual hardware that is accessed by each thread, the group created a Pwm_Wrapper object in (project_files/robot_drivers/pwm_wrapper.py) that contains the PWM hat object and implements another lock object around the calls to the real PWM object.

A side-benefit of this Pwm_Wrapper object is that it includes a variable PWM_WRAPPER_USE_HARDWARE. When this variable is set to False, the drivers no longer depends on the “Adafruit_PCA9685” library or attempt to actually control the PWM hat, meaning the code can be run on a desktop system (assuming other libraries and requirements like a webcam are satisfied).

The Hex_Walker and Torso classes have the Leg objects as members, and serve as a higher level of control. At this level, these changes required some way to “wait until these legs are done moving”. This behavior is done by the Hex_Walker.synchronize() function, which simply waits until the specified leg threads are all done with their operations. This directly replaced the “sleep” calls between pose transitions in the previous version of the drivers. This is still a blocking wait function, just like sleep was. Lastly, a Hex_Walker.abort() function was added that can serve as an emergency stop feature. It simply clears the frame queues for each leg and waits for them to stop. As of project 1, only the Hex_Walker.do_move_set() function has been updated to use the new threading features, and the rest will be converted for project 2. The Torso class has not been updated at all to use these features, and doesn’t yet have a synchronize() or abort() function.

As a final note, Python threads don’t include any sort of “priority” system. The OS decides which threads execute when, and ensures that no threads are starved, but other than that no guarantees are given about relative execution order. This means that each motion may be *slightly* slower than it would be in a priority-based system, and might cause timing issues down the line, but for the project as it is now, everything works just fine.

Robot Sensors

The robot behavior is somewhat dictated by the amount of light that the robot is exposed to. The group has implemented a web camera and the Python library OpenCV to obtain a brightness value for the light in front of the robot. The web camera is used to capture an image which is then converted to grayscale. The image is then divided into 15 different sections where the brightness of the each section is calculated. The maximum and minimum brightness values are discarded and the average brightness is calculated for the remaining 13 sections. The average brightness is indicated by a value between 0 and 255 with 0 being completely dark and 255 being completely light. The full brightness spectrum was split into four different light zones. These light zones dictate the values of two of the input bits for the quantum circuit. The light zones and their respective bit values can be seen in Table 1.

Light Zone	Brightness Values	Q_0	Q_1
Dark	0 - 99	0	0
Slightly Dark	100 -125	0	1
Slightly Light	126 - 152	1	0
Light	153 - 255	1	1

Table 1: Light Zone Breakdown

If the robot is in a dark or slightly dark brightness zone then the robot is more likely to obey the devil and if the robot is in a light or slightly light zone then the robot is more likely to obey the angel. Once the light zone has been determined using the webcam, bits Q_0 and Q_1 are sent to the quantum circuit as two of the input bits.

Quantum Circuit

For project 1, the group designed and implemented a quantum circuit that determines the “mood” of the robot depending on the current player and the amount of the light robot is exposed to. The development of the quantum circuit was done in the IBM Q Experience, which is an online resource provided by IBM that allows users to design and simulate quantum circuits. The derived quantum circuit can be seen in Figure 4.

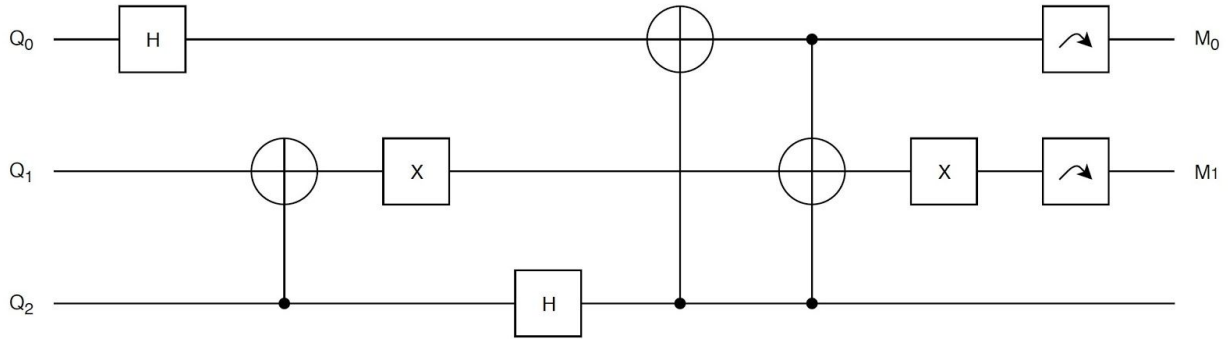


Figure 5: Quantum circuit

The circuit accepts three input bits: Q_0, Q_1 and Q_2. Bits Q_0 and Q_1 are determined by the sensor input function and indicate which light zone the robot is currently operating in. Bit Q_2 corresponds to the current player. The specified light zones, their respective brightness values and output bits can be found in the previous section in Table 1. If it is currently the devil’s turn then Q_2 will be equal to 0 and if the current player is the angel then Q_2 will be equal to 1.

The “mood” of the robot is determined using two measurements on Q_0 and Q_1 at the end of the circuit. These two bits are labeled M_0 and M_1 on the quantum circuit in Figure 4 above. Each robot mood and its description can be found in Table 2 below.

Robot Mood	Measurement Value [M_1M_0]	Description
Obey	00 or 11	The robot will listen to the player and execute the desired movement indicated by the player. *Example: Angel desires the robot moves forward and the robot moves forward.*
Neutral	01	The robot will execute the other possible move that the player is allowed to make. *Example: The angel desires the robot to move forward but instead the robot does not move.*
Disobey	10	The robot will not listen to the player and execute a move that may be beneficial to the other player. *Example: The angel desires the robot to move forward but the robot moves to the right instead.*

Table 2: Definitions of Robot Moods

The quantum circuit was produced in Python using the Cirq library. Cirq is library developed by Google that assists in writing, manipulating and optimizing quantum circuits. Cirq was chosen over other quantum Python libraries to simulate the quantum circuit because it is able to run on the Raspberry Pi without needing any other additional development tools or an internet connection. The code for the quantum circuit can be found in the file entitled quantum_circuit.py.

Challenges

One of the biggest challenges that the group faced during this project was the lack of access to the internet. The Raspberry Pi would not connect to PSU Secure at all or get access to the internet with PSU Guest. This stopped the group from being able to update and install packages on the Raspberry Pi, the solution at first was to take the SD card home and install it on another Raspberry Pi then install all the necessary packages. Another solution that was used was making a WiFi hotspot on a phone and connecting the Pi to it and downloading packages.

Another challenge was the minimal documentation and hardware mismatches by prior groups. The movement code is not all commented and it can be hard to understand what's going on and how to implement a new pose or set of movements. Eventually, the group was able to use the code and add new poses and sets of movement. One of the issues that was faced was a mismatch of some torso servo channels from their code to the actual hardware connection, including a missing connection. The channels were corrected in software and the missing channel was added.

The Cirq package used for creating and running the quantum circuit also uses matplotlib, and for some reason it requires a display output in order to run even though it isn't displaying anything. For this reason, the program cannot be run purely through SSH, the solution would be connecting a display to the robot or using a VNC client to access the Raspberry Pi and run the game, which is what the group opted for.

Future Plans and Improvement Paths

The group plans to continue modifying the Hexwalker object to take advantage of the threading strategy. Currently, it is only benefitting from the interpolation part and smoother movement, but at the Hexwalker level it still uses blocking wait operations. the group wants to allow execution of other code processes in the downtime while waiting for the legs to finish their movements. (The “abort” function is currently pointless, because due to the blocking wait operations it can’t actually be called while the robot is moving. Listening for an abort command from outside is just one kind of process that could be running in parallel.) Supporting true parallelism like this will be immensely beneficial regardless of what game/task the group use the robot for next. After that, additional “macro-motion” functions such as different kinds of dancing or walk cycles might be explored.

Another improvement would be upgrading the servo battery, it would be better to have a higher-capacity battery that is able to run the robot for longer periods of time. This would allow for creation of more “life like” movements, as well as improve the general usability and longevity of the bot in between charging. Better power management could also allow higher flexibility and creativity with regards to programmed move sequences.

Another potential improvement to the current project would be adding speech to text functionality that could be used for sending commands to the robot, as opposed to keyboard interaction. The idea would be to remove the terminal/console interaction of the game, and transfer it to a speech-to-command interaction with the robot. In addition, the group would like to explore implementing the Angel Demon game as a fully flushed Android Application. The idea would be to create a UI interface that would replace the less-flexible console output interaction. An android application could be taken advantage of to seamlessly integrate speech-to-command interaction.

Disclaimer: This is not a proposal for project 2, the group is not committing to any of these possible improvements but merely listing what can be improved. A separate project proposal will be submitted for project 2.

How To Run The Project

Prerequisites to run the game:

Python Libraries:*

- OpenCV
- Cirq
- Numpy
- Time
- Regular Expression
- Random
- MatPlotLib
- PIL
- Threading

*Python 3 is needed in order to properly utilize the listed libraries

Hardware:

- Raspberry Pi
- USB Webcam
- ECE578 Feynman Bot*

Feynman bot is not needed for playing the game but if you would like to experience the full effect of the game using the bot is recommended.

Playing the game

1. Create desired game board grid, with each individual grid square measuring 2ftx2ft.
2. Download repo to Pi.
3. VNC into Raspberry Pi using your preferred VNC client.
4. The angel places two light sources shining onto the game board to create light zones favorable to themselves.
5. The devil has the option to move one of the lights wherever they please but it must still be shining onto the game grid.
6. Using the terminal on the Pi, navigate to the directory where the angel_demon.py file is located and execute the file by typing "`python3 angel_demon.py`"
7. Follow on screen instructions to play and complete the game.

Project Log / Member Contributions

Meshal Albaiz:

- Verified the functionality of the hexapod robot, both the hardware and the software.
- Verified the functionality of the robot torso, both the hardware and the software.
- Installed all necessary libraries and packages.
- Contributed to creating the Angel and Devil game logic.

Tristan Cunderla:

- Created the sensor input module that reads the image from the camera and determines its brightness.
- Designed and created the quantum circuit that outputs the “robot mood” based on the brightness of the image from the camera.
- Contributed to integrating the game logic, quantum circuit and sensor input.

Brian Henson:

- Researched ROS & Python threading methods that could be used to improve the robot movement.
- General organization & improvement to the hex walker driver files.
- Designed & implemented the interpolation-and-threading strategy to allow smoother motion & animation.

Aron Schwartz:

- Created and implemented the “virtual map” of the game using a 2D array with the bot position and bomb position.
- Contributed to creating the Angel and Devil game logic.
- Integrated the game logic with the quantum circuit module.

Videos and Repository

Videos

Driver improvements, before & after (normal speed): https://youtu.be/p2TAcD7aNjc
Driver improvements, before & after (slow speed): https://youtu.be/fD-FDFSJfj8

Angel win demo:

- Move 1: https://youtu.be/PU57y2oQyRE
- Move 2: https://youtu.be/DgCHj_qU_T4
- Move 3: https://youtu.be/kgqghzgDbN8

Devil win demo:

- Move 1: https://youtu.be/k0lNv-ry3x0
- Move 2: https://youtu.be/CkNmdVjbyMk
- Move 3: https://youtu.be/EXe-jz1WxsE

GitHub Repository

GitHub repository: https://github.com/aronjschwartz/ECE_578_Fall_2019