Robot Jousting: Final Report

Alexander Cruz, En Lei, Sunil Srinivasan, Darrel Weng

1. Overview

The goal of this project is to create a physical interactive jousting game using two competitive holonomic robots. Each robot consists of a basic two-wheeled platform equipped with a joust and a hitbox sensor region (a shield) that will register loss of life points when hit. Through a robot-computer interface, vehicles can be controlled either by a user via a Nintendo Wii Remote or by an AI algorithm. Certain regions of the game field contain an embedded magnet to signify power-ups that can be recognized by the robots. The first robot to reach zero life points loses the game.

2. System Modules

There are three major modules to our project: the robot vehicles/interface, the computer vision system, and the AI algorithm. The computer vision module utilizes a camera positioned directly above the game field to capture images of the entire field, tracks the robots on the field, and makes their positions and orientations available via an interface. The AI module consumes this information as well as the game status to maintain an internal representation of the game state and employs a set of rules or algorithms to send commands to the autonomous robot through the robot-computer interface. See Figure 1 for the flowchart diagram of our system and Figure 2 for the image of our physical setup. The robot-computer interface handled translating commands into actuation for both the autonomous and user-controlled robots.

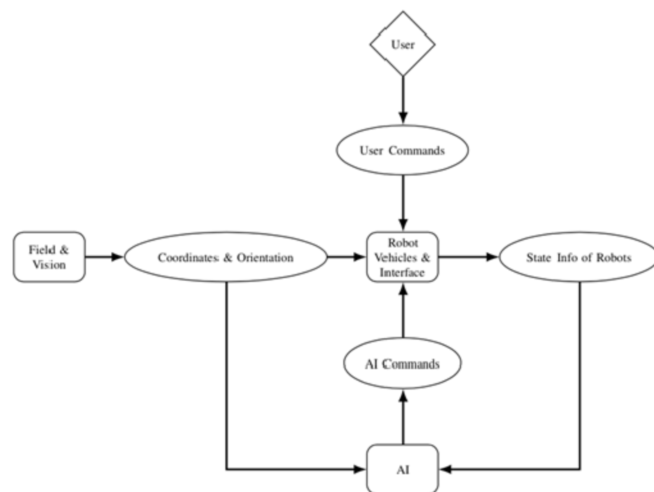

Figure 1: Flowchart diagram of system

Figure 2: Physical setup

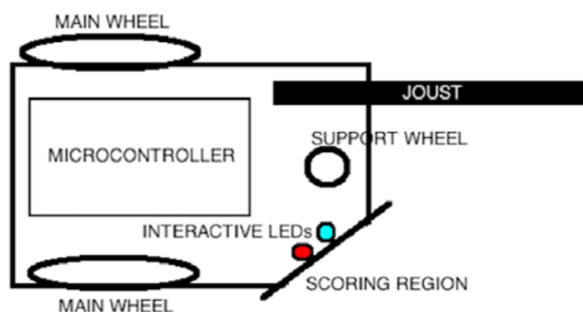

Figure 3: Robot setup

2.1. Robot Vehicles and Interface

We designed and constructed two robots that properly implemented the planned game mechanics and conformed to the interface requirements we had set when developing the project.

1) Game Mechanics

Concerning the game mechanics, both robots require a joust, a hitbox sensor, and a power-up sensor. Implementation of the joust was trivial, and thus design focused on modeling and implementing the hitbox and power-up sensors. The basic robot setup is shown in Figure 3. Our implementation employs a push-button to model the hitbox and a hall sensor to model the power-up sensor. An RGB LED is used to indicate the life points (maximum 3) with colors ranging from white through blue to dim blue/off to indicate decreasing life points. A second, red LED is used to indicate that a robot has been hit or has collected a power-up, as visual indication of a constraint we designed for the game: the robot is invulnerable for a short period after it has been hit or after

it picks up an invulnerability power-up. For particular invulnerability times, we used two seconds for after each hit, and three seconds for invulnerability power-ups. The timed automaton modeling this behavior is shown in Figure 4.

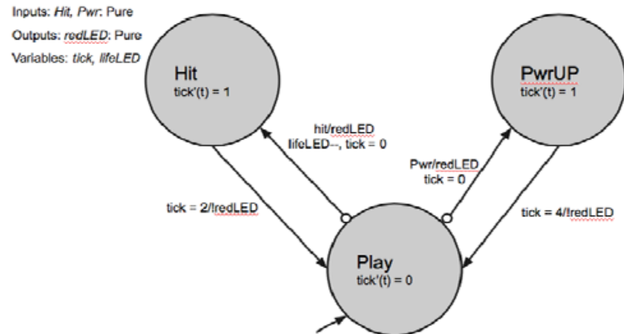

Figure 4: Timed automaton modeling invulnerability behavior

We analyzed the correctness of this model by ensuring that the following two LTL constraints are satisfied:

```
G(Hit=>X(Play))
G(PwrUP=>X(Play))
```

It is easy to see that since no transitions occur between Hit and PwrUP, and the automaton will not stutter, the above constraints are satisfied. Note that the transitions out of Play are preemptive and are therefore implemented as interrupts on the microcontrollers we used. Also note that the two-second delay after each hit registers provides a solution for hardware debouncing of the push button.

2) Interfaces

Our setup requires the robots to have a communication interface with a computer for user and AI control. Wireless serial communication was achieved using a pair of XBee modules for the autonomous robot, and a pair of HC05 Bluetooth modules for the user-controlled robot. Our original plan was to have a Nintendo Wii Remote connect directly to the user-controlled robot, but we discovered that the HC05 module only supports Bluetooth SSP for connection and does not work with the HID interface the Wii Remote uses. Had we known this earlier in the design process, we would have modified our design to use three XBee modules in a mesh network so that all agents in our system could have information about the other agents. Instead, we connected the user-controlled robot to a computer using the HC05 Bluetooth module, then connected a Wii Remote to the computer using a separate, HID-supported Bluetooth module (this created a serial connection). We had initially tested user input using keyboard inputs to send commands to the user-controlled robot, so we used the free software "GlovePIE" to map Wii Remote inputs to keyboard presses such that a user could opt to use either method of input. We implemented several control schemes (tilt-to-turn or button-to-turn) for the Wii Remote-keyboard mapping.

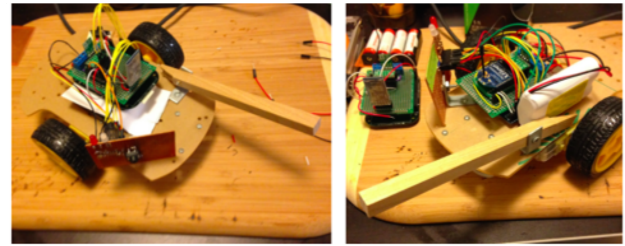
Figure 5: Robot skeletons. Left: User, Right: AI

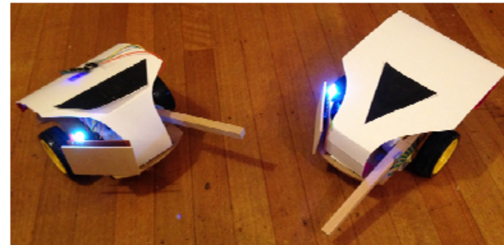
Figure 6: Final robots with markers and shields

3) Hardware

We used the following hardware for our implementations:

- User-controlled robot:
  - CZ-HC-05 gomcu Bluetooth boards
  - PL2303HX USB to TTL to UART converter
  - FRDM KL25Z mBed
  - Radio Shack AA batteries
- AI-controlled robot:
  - XBee Series 1 radio by DigiKey
  - Sparkfun XBee Explorer USB
  - Arduino Uno microcontroller
  - Tenergy 7.4V 2200mAh Li-Ion Battery
- Shared hardware:
  - Pololu DRV8833 Dual Motor Driver Carrier
  - Pololu Adjustable Step-up/Step-down Voltage regulator S7V8A
  - Sunkhee Hall Effect Sensor
  - Motors and Chassis from Emgreat Motor Robot Kit

We consciously chose to use different hardware for the AI and user robots in order to explore the usage of the mBed microcontrollers and Bluetooth communication in comparison to Arduino and XBee. Batteries were used based on availability. See Figures 5 and 6 for our robot hardware.

4) Pitfalls and Future Improvement

The greatest issue we faced in implementation came from the magnetic field produced by the DC motors, which rendered the Hall Effect sensor infeasible to use for power-up sensing. For future improvement, we could replace these sensors with color sensors and change our power-up indicators to use colored markers instead of magnets. In addition, in the interest of symmetry, it may be better to use the same hardware on both robots to facilitate adding new features to gameplay (e.g. additional players).

2.2. Computer Vision System

The purpose of our vision module was to provide the AI robot with a way to determine the location and orientation of all robots on the game field. To achieve this, we mounted a Phillips webcam above the game field and attached it to a computer. Simple black shapes (see Figure 7) were pasted on top of each robot, and a Python program running on the computer read the webcam data and determined the location of each shape. This was achieved using the free OpenCV library for Python (developed by Intel Russia and maintained by Willow Garage and Itseez). The vision algorithm initially took both the triangle and trapezoid shapes as template inputs and determined the contours and 'base' orientations of these templates. It then repeatedly read frames from the webcam. For each frame, it would convert to grayscale and then threshold the image to create a black and white image. On this black and white image, OpenCV functions were used to detect contours in the image. After discarding any contours that contained other contours or were underneath a set area threshold, the OpenCV matchShapes function was used to determine which contours in the image most closely matched the templates. The location of each shape was saved as the centroid of the best-matched contour for each template, and the orientation of the tracked shape was determined by finding the orientation of the contour in the image and subtracting the 'base' orientation of the template, modulo 360 degrees. Contour orientation (for both templates and matches) was determined by finding the orientation of the vector from the centroid to the center of the minimal enclosing circle for the contour. All this functionality was encapsulated in a "ShapeTracker" class, which other modules, notably the AI module, could instantiate with arbitrary template images to process webcam images. An interface for defining templates and then retrieving the detected locations and orientations of those templates was established for this class, as well as a method for requesting that another image from the webcam be processed. Figure 8 depicts a visualization of the vision module's capabilities, with faint blue circles showing the retrieved locations and orientations of each template shape.

Figure 7: Templates used by AI module for Vision module initialization

Figure 8: Visualization of vision module capabilities

The initial design called for orientation to be retrieved using the moments of each contour (a set of descriptors). However, we found that the moments could only be used to retrieve an axis of orientation for each contour (-90 to 90 degrees), rather than a unique orientation out of 360 degrees. We therefore modified our design to use the previously described method for extracting orientation. This issue, as well as an issue with contour hierarchies (solved by discarding contours that contained other contours) and a minor lighting-based issue, were the only issues that arose during development of this module. However, the module may still output 'false positives', where shapes in the images are falsely detected as matches for template shapes. The module has a modifiable threshold that can be tuned to increase or decrease the leniency involved in matching shapes.

2.3. AI Algorithm

We designed our AI robot to follow a greedy algorithm to produce an aggressive opponent using the following requirements:

- Active Pursuit: AI robot pursues the user robot if it is not directly in front of the user robot
- Reorientation to Attack: If the AI robot finds itself in front of the user robot, it reorients itself to point joust towards the hitbox on the user robot
- Boundary Avoidance: If the AI robot is near the boundaries set by the vision module, it attempts to move away from the boundary by backing up and turning 180 degrees, then moving forward away from the boundary. Relative directions for the robot are as shown in Figure 9.

We implemented our algorithm as a state machine coded in Python (see Figure 10 for the state machine diagram).

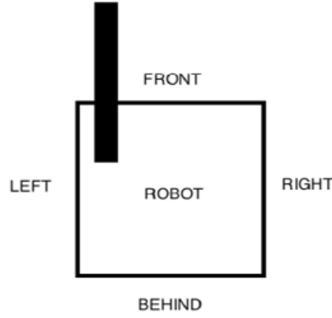

Figure 9: Directions as referred to in description of algorithm

Specific features of our state machine include:

- 7 states describing AI robot behavior
  - STOP - not moving (initial state)
  - FWRD - moving forward at set speed
  - LEFT - rotating counterclockwise in place at set angular velocity
  - RIGHT - rotation clockwise in place at set angular velocity
  - ADJST - AI robot is in front of the user robot and must start to aim at the hitbox
  - ADJ_L - rotation counterclockwise to aim at hitbox
  - ADJ_R - rotation clockwise to aim at hitbox
- Inputs:
  - Vector orientations from both robots (0 to 359 discrete degrees), obtained from vision module (θ_{AI} and θ_{USER} in Figure 10)
  - Relative angles of orientation of each robot to the other (0 to 359 degrees), calculated from position and orientation information (Φ_{AI} and Φ_{USER} in Figure 10, describing angle of AI compared to user and user compared to AI, respectively)
  - Distance (D) between the two robots as a positive real number in unit pixels, calculated from position information
- Output: the next move for the AI robot to take, with four directions being the available values; this is later translated to commands sent to the robot

We also assign variable names to the evaluations of our guards:

- **isFwd**: $\Phi_{AI} \leq 30$ or $\Phi_{AI} \geq 330$
- **isLeft**: $180 < \Phi_{AI} < 330$
- **isRight**: $30 < \Phi_{AI} < 180$
- **inHitRange**: ($\Phi_{USER} \leq 30$ or $\Phi_{USER} \geq 330$) and $D \leq 25$
- **orientMatch**: $-15 \leq |\Phi_{AI} - \Phi_{USER}| - 180 \leq 15$ (checks if AI is pointing towards scoring region of user)
- **isHit**: **orientMatch** and $D \leq 10$

Please refer to Figure 10 for each state's update function.

Inputs: θ_{AI}, θ_{USER}: $\{0,..,359\}$
ϕ_{AI}, ϕ_{USER}: $\{0,..,359\}$
distance: $\mathbb{R}^+$

Output: move: {forward, left, right, stop}

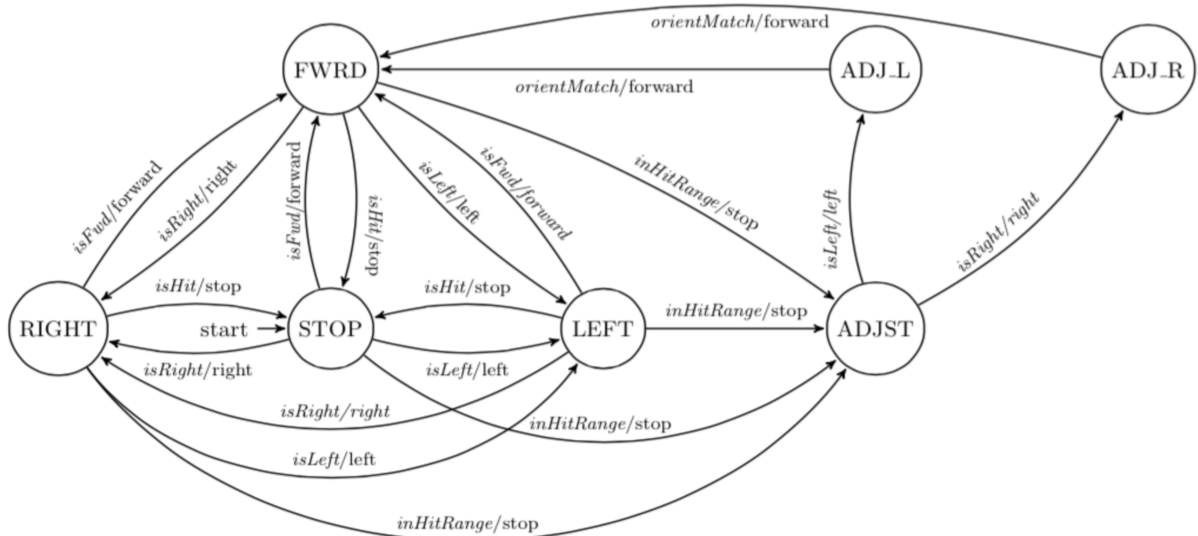

Figure 10: AI robot state machine

Simulation test results of our AI algorithm proved to work satisfactorily and exhibited acceptable behavior. However, after integrating the AI module into our overall system, we found that our AI implementation suffered from latency issues. In particular, the time the AI needed to obtain the vision data, compute the next move, and then send the command over the XBee serial connection was a much longer duration than the acceptable response time range of our robot. Ultimately, this resulted in our AI having delayed reactions and causing it to exhibit suboptimal behavior. We tried to account for such delays by modifying the AI implementation to run faster and implementing a buffer to speed up the data exchange from the vision module to the AI module. This did reduce the amount of lag, and the AI behavior improved as a result.

Another issue that arose with the module involved the dependence on information from the vision module. In environments where the lighting conditions are dim, flooring is uneven, or there are noisy 'false positive' shapes, the vision module may experience performance issues when the AI module repeatedly polls it, or shapes may flicker in and out of view. Correcting the lighting conditions or other environmental conditions helped account for some of these issues. However, if shapes are not detected in any images, then the AI module does not perform any updates, as it cannot obtain any new input data from the vision module. Incorrectly identifying shapes results in the AI outputting incorrect moves, as expected. As such, potential factors that affect the proper function of the vision module also affect the AI behavior. However, we improved the AI module to be more robust to these issues, such as with a median filter for orientation tracking.

Overall, the AI algorithm as a standalone module works in theory and simulation, and we managed to improve the AI module such that any limitations it has do not impair its intended behavior.

3. Conclusion

Our modules fit together fairly well, and we even managed to improve our AI robot behavior to the point that the previous issues disappeared (as in our project video), but during the live demonstration, the relevant environmental conditions were not reproducible, though the AI robot still made for an entertaining opponent. While there are some Python-specific issues with our programs, we are satisfied with the result, since our goal was to make something fun to interact with using the concepts discussed. We have mentioned possible future directions for this project, such as the addition of more players or more game complexity, so there are certainly interesting extensions to consider. Overall, we gained a better understanding of the process of embedded system design (such as consideration of environmental factors), and were able to create an enjoyable experience.

References

- Anaconda by Continuum Analytics: Python, NumPy. http://store.continuum.io/cshop/anaconda/.
- Arduino. http://arduino.cc/
- ARM mBed. http://developer.mbed.org/
- GlovePIE. http://glovepie.org/
- OpenCV. http://opencv.org/
- Python programming language. http://python.org/