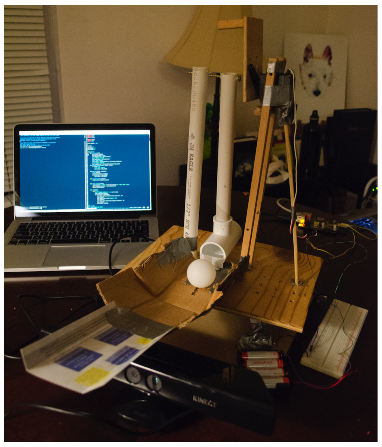
DogeFetch

Jessica Lin, Varun Rau, Wonjun Jeong
EE149 - Fall 2014

12-19-14

I. Introduction

DogeFetch is an automated robot to play fetch with a dog without human interaction. Utilizing a Kinect, Raspberry Pi, and spare parts scavenged in the Embeddeed Systems lab, our team has created a fully functional dog toy. High level description: DogeFetch will receive the ball from the dog. After detection of ball, DogeFetch will lcoate the dog in it's 57°viewing angle. The robot will then use a freefalling hammer to kick the ball in the direction opposite of the dog.



II. Electronic Materials

Raspberry Pi
The embedded device used to power the robot and run the detection algorithm and utilize the statemachine. The Pi connects to the Kinect through USB, and provides GPIO/PWM pins to interface with the Servos
Kinect
With a viewing angle of 57°, the camera that the Kinect provides detects the location and depth of the dog in it's field of vision.
Servo - Platform
A rotation servo used to orient the direction the ball will be launched
Servo - Hammer
A rotation servo used to lift the plastic hammer clockwise until gravity will cause the hammer to fall ahead of the servo. The servo then returns to it's orignial position, ready to repeat the action.
Microswitch
A switch to be triggered by the ball upon receipt from the dog.

III. Modules

I. Control - Raspberry Pi(Concurrency)

The core module is the Raspberry Pi, which serves as the control module to operate all other modules. The Pi is just as sophisticated as a computer. It provides USB ports for the Kinect to connect to, and GPIO pins to deliver PWM signals over to the servo motors. The current method of running DogeFetch is to login to the Pi, and run the main.py (statemachine) program. As soon as the main.py program is running, DogeFetch is fully capable to operate on it's own. The Pi is powered through a MicroUSB cable that is connected to the outlet at this point in time. The Pi requires a lot of power to receive video from the Kinect as well as operate the servo motors.
For the control module, DogeFetch is written in Python, at main.py. We will go more into depth of the statemachine in section IV.

```
# Code at: https://github.com/wonjun/DogeFetch
# In various modules below
# after initialization:
from threading import Thread
```

```
Thread(target=self.run, args=())
```

Using python, we were able to launch all of the modules below into different threads using the code above.. Thus by using concurrency, we were able to have callbacks within the code for the statemachine to change states. Every module was able to communicate with the main thread, and the main thread was able to act accordingly. Concurrency was needed in order to detect when the ball arrived in the machine, and also to rotate the platform after the detection of the ball.

II. Vision

The vision (vision.py) module is the component of DogeFetch that is responsible for detecting the location of the dog in the Kinect's viewing angle. Multiple steps were taken to get vision to work. There were several things that were taken into consideration when running our dog detection algorithm. First, because all computation happens on the Raspberry Pi, it is imperative to remain as economical as possible in regards to computations and memory usage. To that end, making the images as small as possible is essential. Because the dog used in this demonstration is a distinct shade of white, it was sufficient to filter out all colors outside RBG(255, 255, 255) - RGB(127, 127, 127). Applying this mask and thresholding the image meant that the Raspberry Pi needs to process less data, reducing the computational load on it. On this masked image, we then run a standard computer vision algorithm, contour detection, to find all the contours in our image. This works by generating bounding boxes for all the continous regions that remain in our masked image. Filtering out the contours that cannot be dogs–boxes that are too big or too small–leaves a bounding box for the location of the dog. The vision module then finds the center of this most-likely bounding box and returns left if the center is found on the left side of the Kinect's field of vision, otherwise it returns right. This is then returned to our Robot's main run loop via a callback and is interpreted by the control module based on the current state of the Robot's statemachine.

Kinect Driver to work with Raspberry Pi
Luckily enough, there is a driver that can be found online on Github. (The driver is at https://github.com/xxorde/librekinect.) When installed, the Kinect's video will appear on /dev/video0 - the traditional location for all video devices that can be connected to the Pi. Installation was definitely not trivial. It was tricky, and took a lot of (uninterrupted) time (7 hours). This is due to the installation process of the driver.

```
def find_dog_contour(self):
  _, im = self.camera.read()
  COLOR_MIN = np.array([127, 127, 127],
                    np.uint8)
  COLOR_MAX = np.array([255, 255, 255],
                    np.uint8)
  mask = cv2.inRange(im, COLOR_MIN,
                   COLOR_MAX)
  output = cv2.bitwise_and(im, im,
                         mask = mask)
  ret,thresh = cv2.threshold(mask,0,255,0)
  contours, hierarchy =
  cv2.findContours(thresh,cv2.RETR_TREE,
              cv2.CHAIN_APPROX_SIMPLE)

  bounding_rect = None
  if len(contours) > 0:
    areas = [cv2.contourArea(c)
             for c in contours if
             cv2.contourArea(c) > MIN_AREA]
    if len(areas) > 0:
      max_index = np.argmax(areas)
      cnt=contours[max_index]

      bounding_rect = cv2.boundingRect(cnt)
      x,y,w,h = bounding_rect
      cv2.rectangle(im,(x,y),(x+w,y+h),
                   (0,255,0),2)

  return bounding_rect
```

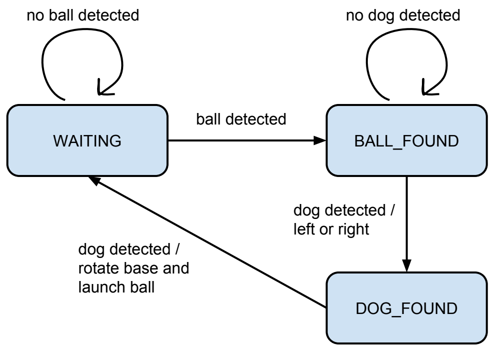
III. Actuation

In order to support hardware input and output from the Raspberry Pi, we had to use two servos and a microswitch.
Rotation Servos

```
while True:
    while self.should_spin[0] == 0:
        time.sleep(POLL_FREQUENCY)
    self.pwm = GPIO.PWM(self.pin, 1000)
    self.pwm.start(0)
    if self.should_spin[1] == "left":
        self.pwm.ChangeDutyCycle(4)
    elif self.should_spin[1] == "right":
        self.pwm.ChangeDutyCycle(72)
    else:
        self.pwm.ChangeDutyCycle(1)
    while self.should_spin[0] > 0:
        time.sleep(POLL_FREQUENCY)
        self.should_spin[0] -= 1
    self.pwm.stop()
```

This code fragment from servo.py, is explained below.
In order to use the rotation servos, we had to utilize several differnt techniques. First, we needed an external power source other than the Raspberry Pi to power the Servo. At first, when we hooked up the servo to the 5V provided by the Pi, the Pi would lose power for it's own processor and crash. After researching online, we decided to purchase a 4AA battery pack to power all of our actuators. After the power was hooked up to the servo, we hooked up the two servos to two different GPIO pins on the PI. GPIO pins are voltage output pins that serve to actuate. By using software found on the internet, we were able to manipulate the GPIO pins to serve PWM signals over to the two different servos. What was tricky was to get a servo to rotate both clockwise and counter clockwise– we were able to do this by stabilzing the frequency at 1000hz, and changing the duty cycles of the PWM signals. At low levels, the servo would rotate clockwise, and at high levels, it would rotate counter clockwise. Granular changes of the PWM signal would lead to the speed of rotation being changed. We used the servos to actuate two different things:
1. Base Rotation
The base rotation is required to orient the hammer in a certain direction. The servo was deliverd a 1000hz PWM signal with a duty cycle of 5.0 to rotate counter clockwise at a slow rate. This signal was sent for .5 seconds to the servo. A duty cycle of 72.0 was used to rotate the servo clockwise. Thus, after detection of the dog, the servo would rotate accordingly in different directions, and then the control module would trigger the hammer servo.
2. Hammer Rotation
Since the servos rotate too slowly, we came up with a method to deliver enough force to a ball so that the dog could chase it. The hammer is made out of plastic, and is in free swing mode on an axis. The servo, then rotates a wooden piece that carries the hammer in 180degrees, until the hammer then freely swings downwards to "kick" the ball. The servo was strong enough to lift the hammer. In order to rotate the hammer, we delivered a PWM signal of .5 to the hammer, since if the signal was too low/servo was rotating slowly, it would not have enough torque and force to carry the hammer.
Microswitch - Ball Detection We utilzied a microswitch under the ramp that should have the ball from the dog recieved, and launched from. We calculated the weight the ramp would place on the switch, and allowed it to be sensitive enough for various different kinds of balls. We also hooked up this microswitch to a external powersource. We utilized the same GPIO software that would power the servos, except changed it to be an input pin. When the switch was pressed, it would deliver a high voltage to the pin, allowing the control module to change states from WAITING to BALL_FOUND. The code, switch.py, for this was very simple. At recurring frequencies, the code would poll the pin to detect voltage inputs.

IV. Statemachine


no ball detected
no dog detected
WAITING
ball detected
BALL_FOUND
dog detected / left or right
dog detected / rotate base and launch ball
DOG_FOUND

The state machine contained three main states to govern the modal behavior: WAITING, BALL_FOUND, and DOG_FOUND. In the state WAITING, the only action that can move the state into BALL_FOUND is the activation of the microswitch. Once the microswitch is activated and thus a ball is detected, the state machine will request the vision module to report back the location of the dog, if there is a dog present. If there is no dog located by the Kinect, the machine will remain in the BALL_FOUND state. Once receiving the dog location if a dog is present, the state machine will then transition into the DOG_FOUND state. Upon reaching this state, the base servo activates to rotate in the direction away from the dog. After the base servo completes its rotation, the device sleeps for a few seconds before activating the hammer servo and launching the ball. After the ball is launched, the base servo again rotates back to the starting direction, negating its previous rotation. This process brings the state machine back to the WAITING state, which it remains in until the next microswitch activation.

V. Simulation

Simulating our statemachine required triggering sample voltage inputs and outputs on our raspberry Pi. We were able to use software that could detect whether the pin was Hi or Low. We would then run our main.py, and by inserting print statements in the code, examine it's behavior based on what the pins were doing. We simulated the switch, ball detection, by putting the respective GPIO Pin on High, and then observed the statemachine to change to the BALL_FOUND state from waiting. Then we gave sample image data to our vision module, and observed the code until it switched to DOG_FOUND. Then we observed the two servo GPIO pins with software to make sure they were sending the PWM signals, and were high for a period of time. And then main.py would return to WAITING.

VI. Lessons Learned and Next Steps

I. Lessons Learned

We all learned an immense amount throughout the course of this project. A main lesson we learned is to be very specific about the hardware we choose, and to research thoroughly to make sure that the hardware is capable doing what we need it to do. For example, we tried to use a Beaglebone in the start instead of a Raspberry Pi, but had to use a Pi instead because interfacing the Kinect with a Beaglebone was too difficult. Also, we realized that the rotation servos that we chose were too slow to implement our previous model for ball launching (two rotating disks that would shoot the ball out), but we only realized this as we were building the device so we had to create a new method for ball launching halfway through the project using the servos that we had.

Another major lesson we learned is that the planning and design phase is always hard and never going to be perfect, so it's better to start working on the actual hardware early on so that iteration is possible. We thought through and worked out the design of the device very early on, however didn't start actually building and putting together all the parts until later. Once we began, we ran into difficulties such as not having the right resources to make the device (resources like the Invention lab or a 3D printer). If we had begun sooner, perhaps we could have tried to get access to such resources, or even just bought more pre-made parts using our budgeted money.

II. Next Steps

There are a lot of possibilities to improve DogeFetch in the future. First off, we would love to make a more stable physical housing for the device, since right now it is mostly made from scrapped pieces of old projects and a bunch of duct tape. We could get access to a 3D printer and print the parts for the device.

Secondly, the device only detects a dog in the visual range of the Kinect right now, which is less than 180 degrees. It would be fantastic to be able to allow 360 range for the dog detection, so that ball launching can actually be in the complete opposite direction of the dog (180 degrees from it). We could implement this by having multiple cameras in all directions instead of just one Kinect in one.

Lastly, our program is only written to detect dogs of white color right now. It would great if the vision module could be rewritten to detect any kind of dog. To implement this, we would have to rethink our dog detection implementation to probably incorporate depth or motion information as well. Another possibility would be to require the dog to wear a specific collar that displays an AR code for the cameras to track, or something like that.

VII. Conclusion

There were several key considerations that we made when making this device on both the software and the hardware side. We knew that for a device like this that is constantly in communication with external sensors and actuators, we needed to be able to create a system of abstractions that would allow us to control the robot's different hardware components independently while also keeping our model as well abstracted as possible to make reasoning about the system easier. After considering different approaches to this problem, we decided to deviate from the framework that we had used in lab, to a continous, multithreaded callback method that allowed us to interface with all of our hardware in realtime. This became very useful when using the Kinect, which we found had a very high latency and low framerate. On the hardware side, we had to make some tough design decisions to achieve our goal given actuators that were not as powerful as we had original hoped. By building our own hardware, we were able to be very flexible with the structure of our device, which ultimately led to our success. Using the Raspberry Pi, we were able to interface with sensors (the Microsoft Kinect and a microswitch) and actuators (Rotation Servos) in a concurrent system while still getting predictable, correct results and state transitions. In the end, we built a device that any dog owner will love!