SHaZam the Magic Lamp: IR-Based Gaze Tracking and Light Direction

Chaim Halbert, Dexter Scobee, and Edward Zhao
EE149A/EE249A Project Final Report

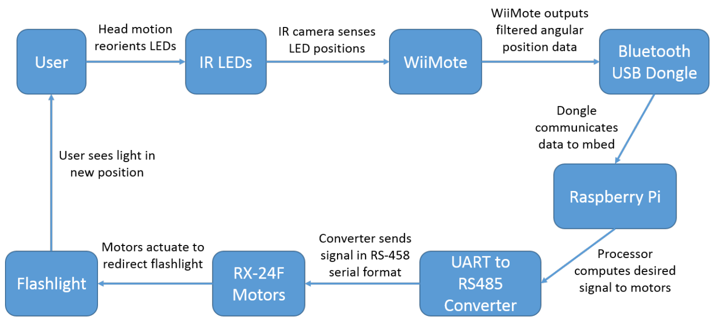

User
Head motion reorients LEDs
IR LEDs
IR camera senses LED positions
WiiMote
WiiMote outputs filtered angular position data
Bluetooth USB Dongle
Dongle communicates data to mbed
Raspberry Pi
Processor computes desired signal to motors
UART to RS485 Converter
Converter sends signal in RS-458 serial format
RX-24F Motors
Motors actuate to redirect flashlight
Flashlight
User sees light in new position

Fig. 1: SHaZam ConOps. This diagram represents the interconnections between components for the SHaZam system

I. Project Vision

The goal of this project was to design a lamp that will redirect its light to follow a user's gaze. The magic lamp will consist of a flashlight attached to a two-axis motor assembly, to be directed by an embedded microcontroller. Additionally, the lamp will utilize a Wiimote to track IR LEDs affixed to a specially designed hat. The Wiimote will communicate relative positioning data to the microcontroller via Bluetooth, which will in turn direct the motors to aim the flashlight. The controller will behave according to a state-machine that describes both automatic and manual modes of behvior. Figure 1 shows the logical connections between system components and how they are intended to interact, and Figure 4 depicts the governing state machines.

II. Hardware

The hardware consists of a gimbal-mounted flashlight connected to a system controller, in turn connected wirelessly to a Wiimote, which tracks sensor bars mounted on the user's hat.

The gimbal is a pair of Dynamixel RX-24F smart servos, which are daisy-chained and connected to the system controller via RS-485 serial link. To convert the RS-485 to 3.3V serial UART, we used an SP3485 breakout board.

The system controller is a Raspberry Pi running Raspbian Linux in "headless" configuration, without a monitor, keyboard or mouse. These can be added for development and debugging, or an ethernet connection can be used to log in remotely via SSH.

The system controller links to the Wiimote via a USB Bluetooth dongle. (Somehow, we had issues with newer Wiimotes that included MotionPlus, so we used an older Wiimote without it.) The WiiBrew web site [1] was a valuable resource in understanding the Wiimote hardware and in interfacing the Wiimote with the system controller.

[1]http://wiibrew.org/wiki/Wiimote

The hat has two Wii "sensor bars" mounted to it. These bars are not sensors at all, but actually contain a simple collection of IR LEDs at either end of each. The specially-designed camera in the Wiimote picks up and tracks these four points of invisible light. For more information about the sensor bars' asymmetrical configuration, see section IV.

Design changes

Although the overall structure and linkages of our system diagram have not changed since the beginning of our project, virtually every block was modified by the end of development.

1) Gimbal: Our original design called for traditional servo motors to drive the gimbal's motion, but we decided to change to advanced, serially-controlled servos. These Dynamixel RX-24F motors (see Figure 2) provided several advantages, the primary one being ease of control.

Unlike traditional servo motors which are commanded to set their position using PWM, our smart servos could be commanded via RS-485 serial link. This provides more timing flexibility in our control logic, since this makes PWM interrupt routines unnecessary. The smart servos also allowed us to set the speed of the movements over the serial link, offloading logic from our main control loop to the motors.

Also, the new servos were daisy-chainable, an impossibility with PWM. This allowed us to use a single serial output, as opposed to multiple, independent PWM outputs, each with their own interrupt routines. This resulted in a subtantial time savings in development, and greater reliability.

A disadvantage was that RS-485 is quite different from serial UART, in that it uses 200 mV differential signaling over two wires and is half-duplex, whereas our TTL UART operates at 3.3V and is full-duplex. We used a breakout board for the SP3485, which accomplishes both level conversion and protocol translation, with the aid of an RTS signal from our system controller's GPIO.

2) Bluetooth: Originally, we planned to use a BlueSMiRF Gold to connect to the Wiimote. However, the Gold only connects via serial data endpoints, while the Wiimote requires an HID interface. So, the BlueSMiRF Gold did not work.

To resolve this problem, we purchased a BlueSMiRF HID, which has the same hardware as the BlueSMiRF Gold, but has different factory-flashed firmware for HID capabilities. This also did not work, because the BlueSMiRF HID was designed to operate as an HID slave device, such as a mouse, keyboard or joystick; it could not operate as an HID host, to control such HID devices.

In the end, we used a USB-Bluetooth dongle.

3) System controller: Although we originally intended to use the ARM mbed FRDM-KL25Z Freedom board, we decided to change platforms to the Raspberry Pi model B.

We did not use the Pi initially because the lab already provided each of our group members with a free, personal ARM mbed

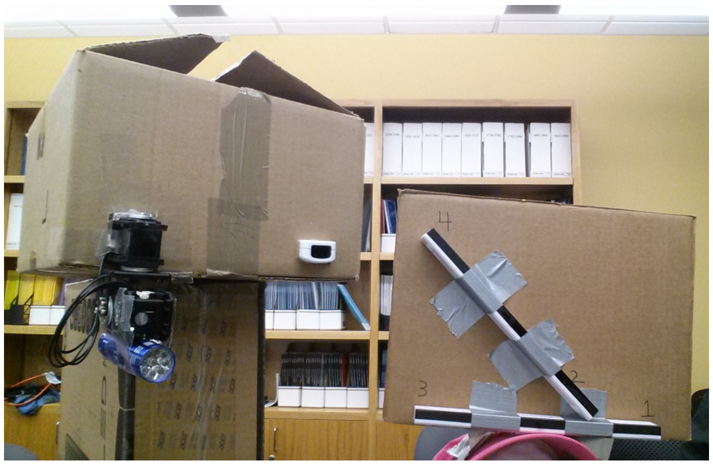


Fig. 2: SHaZam External Hardware. On the left, the main console with a protruding Wiimote camera and a Dynamixel 2-axis motor assembly with an attached flashlight. On the right, the head-mounted IR LED assembly for user wear.

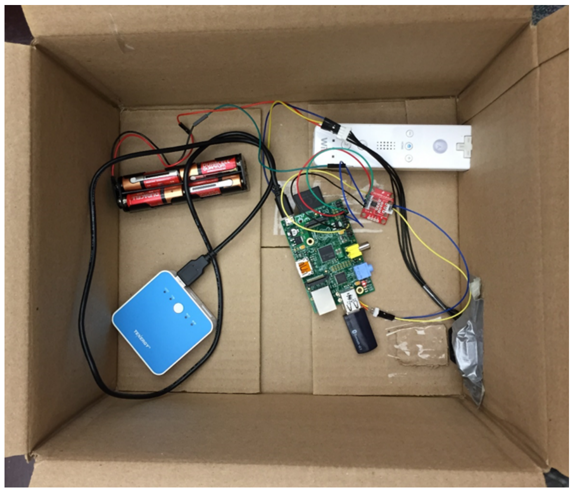


Fig. 3: SHaZam Internal Hardware. Clockwise, from top: Wiimote, Sparkfun RS485 breakout board, Raspberry Pi with attached Bluetooth dongle, USB power module for the Raspberry Pi, Battery assembly (8 AA) for powering motors.

processor. This allowed us to work independently, parallelizing development. In comparison, the Pi cost \$35. Also, PWM control requires Linux kernel programming on the Pi, or a separate daughterboard with its own microprocessor, such as the Arduino-compatible Pi Alamode (another \$35). We deemed this too complex and expensive.

However, circumstances changed during development. By switching to the serial servos, we eliminated the need for kernel programming or a daughterboard. Just like with the ARM mbed, we could now control all the hardware with a single system controller. Also, when we exchanged the BlueSMiRF for the Bluetooth dongle, the easily-installed support for the dongle in Linux gave the Pi a clear advantage over the ARM mbed, which in contrast required compiling and integrating C++ code. The Pi was simpler.

The Pi also offered new capabilities not available on ARM mbed. With the Pi, we could change languages from C++ to Python, and we could develop and test directly on the Pi itself. This eliminated the need for compilation and flashing every time we made a change. It also gave us an interactive Python shell to test out snippets of code prior to integration.

Next was the ethernet connection on the Pi. Connected to a LAN, multiple members could work simultaneously on the Pi via SSH. Also, with an internet connection, we could use package managers to quickly install and test pre-built third-party modules for new Linux and Python functionality. The internet also gave us tighter integration with GitHub for version tracking and merging our code modifications. The confidence that we could quickly revert our changes allowed rapid progress even as the deadline approached.

III. Software

On the Raspberry Pi platform, we opted to utilize Python as our language of choice. The reason for that is twofold; the first is that there is an extensive library called CWiiD[2], which provided a robust API to interface with the Wiimote. The second reason was that, once we had decided to use the Pi and its full Linux capability (see section II), we wanted to use a language with which we had more experience coding and debugging, cutting down development time by significant margins.

[2]CWiiD was developed by Donnie Smith of Georgia Tech - https://github.com/abstrakraft/cwiid

In the actual code, the main logic resides in statechart.py. The state machine logic, connection to the Wiimote via CWiiD, and sample logic all reside in this file. Essentially, we run a while loop, sampling every 0.2 seconds for IR position data. Based on this data, we used the modeling algorithms (described in section IV to find the appropriate pitch and yaw angle, then move the motors to that angle. However, periodically polling the Wiimote with the get_mesg() method yields only the oldest undelived data, which was not sufficient for our purposes. We had to introduce a wrinkle of asynchronous behavior, by taking advantage of callback functions. Essentially, once set up, CWiiD receives messages with positional data and button status data continuously. So, with a callback set, everytime a new message came in, we update our variables storing the latest data points, so the most up to date values are always available when we sample. We also keep track of what buttons are being pressed within the callback function. For instance, if we detect button A being pressed, we immediately go into Manual Mode, which would not be possible in a purely synchronous fashion (a button press might be missed if it occurs between sampling instances).

We import a file in statechart called motor_control. Like the name suggests, motor_control.py is responsible for controlling the motors. Once the desired pitch and yaw commands are calculated in statechart.py, and deemed to be within the designed range of motion, they are passed into motor_control.py. Motor_control.py then takes this data and determines what commands need to be sent to the motors in order to realize these commands. The program first calculates the difference between the desired pitch and yaw and its current pitch and yaw, and finds the necessary speed needed to reconcile that difference during a single sampling period. Once this is calculated, motor_control.py constructs a serial packet which contains an instruction to be sent to the motors. This packet will tell the motor to overwrite its “movement speed” variable with the new value. A second packet is constructed which tells the motor to update its “goal position" variable. These packets are prepared for both pitch and yaw motors, for a total of four packets. These packets are sent via the Raspberry Pi's UART to the RS485 converter, and from there to the motors themselves. Within the motors, the new data supplants the preexisting values in memory, causing the motors to behave appropriately and allowing them to "track" the user.

IV. Modeling and Algorithms

We developed a finite state machine that models the behavior we want for our project, as well as mathematical models detailing our tracking algorithm.

FSM System Model

The SHaZam system can be modeled as a hierarchical composition of state machines. At the highest level (shown in Figure 4a) the system starts by looking to pair via Bluetooth with a Wiimote (state BLUETOOTH_PAIRING), and continues doing so until it successfully pairs. This is the state first entered when the sytem is turned on.

Once paired, the system will transition to AUTO mode, which enables tracking of the user position. Within AUTO, there are two identical yet separate state machines for controlling the motors pitch and yaw. Figure 4b shows the FSM model for these machines. The system begins in the STAY state for both angular axes. For a given axis, if the motor are commanded to change its angle by a value which is within our accepted bounds (described in the figure as the range between minThresh and maxThresh) then the system transitions to TRACK and the lamp will be reoriented to point at the appropriate location. If the angular command falls outside of this range, that command is ignored and the system remains in STAY (in this case, the calculated command is never actually sent to the motors). Once in TRACK, if the change in angle is below minThresh (the desired lamp position has settled) or the change is above maxThresh (likely due to an errant measurement), then the system returns to STAY and holds its position.

If the user pushes the "A" button on the Wiimote, the system will transition into MANUAL (the manual mode is represented in Figure 4c). This sub-state machine follows a similar model to AUTO in that it also transitions back and forth between a stationary state (NotButtonPress) and a moving state (ButtonPressed). As the state names suggest, the transition is triggered by the pressing or releasing of buttons on the Wiimote. In ButtonPressed, the change in motor position is determined by which button is being pressed, and at every time increment (recall that our sample time is 0.2 seconds) the appropriate change occurs. Unlike AUTO, MANUAL has a third state, labeled Rumble. When a button is pressed that would have commanded the system to move beyond its designed range of motion, the system transitions to Rumble in lieu of ButtonPressed. In Rumble, the system remains stationary and vibrates to alert the user that they have reached the bounds of motion.

Finally, if the user presses "A" and "B" simultaneously on the Wiimote, the system will transition from MANUAL back to AUTO, allowing the user to change between these two modes of operation at their discretion.

User State Estimation

We have also derived a mathematical model that will allow us to reconstuct the state of the LED configuration (which reveals where the user is looking) by obtaining angular position measurements from the Wiimote.[3] The data provided are x- and y-angle measurements, corresponding to the estimated angular position of the LEDs in two orthogonal planes. Figure 5 shows the geometry of reconstructing the state in one of these measurement planes. Note that the state of the LEDs in a plane has three degrees of freedom (shown in Figure 5 as x_3, y_3, and ψ_{user}), so three angular measurements are needed to unambiguously reconstruct the state. Once the state in the first plane is known, however, the range data (x_3) is known for the second plane as well, so only two angular measurements are needed to reconstruct the state relative to the second plane.

Command Generation

Given the user's head position (x,y,z) and orientation (pitch θ and yaw ψ), we could then use a second algorithm to calculate where the user's line of sight would intersect with the surface of the table upon which the SHaZam system was placed. Once that location was known, it was then possible to calculate the required pitch and yaw of the lamp to shine a light such that it would also intersect the table at that same location. This process of calculating the intersection location based on user angles and then the required lamp angles based on the intersection location was accomplished using standard trigonometric and inverse trigonometric functions.

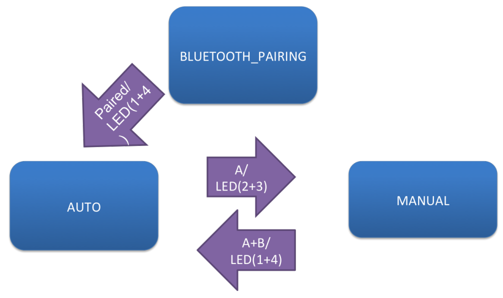
Verification and Testing

In order to verify the accuracy of these solutions, we created a simple model of the system in MATLAB that allowed us to specify user state data (angular and positional state), determine where the LEDs would appear relative to the Wiimote camera, and use that data to drive our state estimation and command generation algorithms. After successfully recreating our input state data based on expected measurements, the simulation indicated that we had derived an exact solution to the problem. However, upon running our algorithm using actual data collected by the Wiimote, we soon discovered that even in a static configuration (minimal motion of the Wiimote and the LEDs) the estimated angular state of the user exhibited large variations that would give erratic data to the command generation algorithm, and would have resulted in extraneous lamp motion. We confirmed this numerical instability in our MATLAB simulation, noticing that even a 1° error in a single measurement could result in nearly 50° of error in the user orientation estimate.

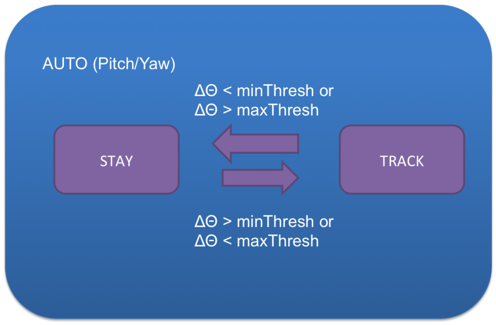
Managing Noisy Data

While the estimated angular state of the user was determined to be unusable (varying by tens of degrees), the estimated user position was notably more stable (varying on the order of one or two centimeters). It was therefore decided to re-scope our design to focus on directing the lamp at the user themselves. This change was accomplished without significant variation to our existing state estimation algorithm (we were already estimating positional data) or our command generation algorithm (we substituted the user's head position for the position of their gaze upon the table). To further ensure that the motors would not be commanded to move erratically, we tuned the angular change thresholds used in our state machine to allow the motors to respond to meaningful

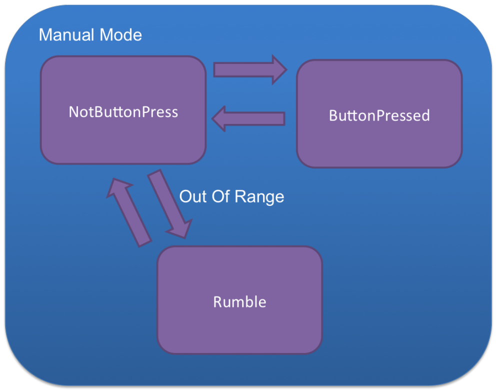
[3]This model is an expansion of work done by Johnny Chung Lee (Google, formerly CMU). See http://johnnylee.net/projects/wii/



(a) Top-level state machine model


AUTO (Pitch/Yaw)
ΔΘ < minThresh or ΔΘ > maxThresh
STAY
TRACK
ΔΘ > minThresh or ΔΘ < maxThresh

(b) Automated mode state machine model



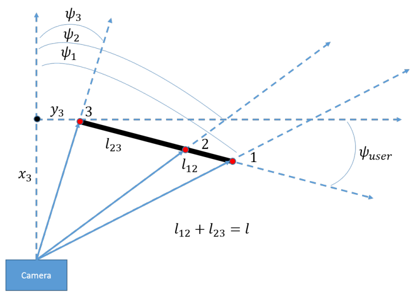
(c) Manual mode state machine model

Fig. 4: State Machines. The system is modeled as a hierarchical composition of state machines

small changes while ignoring extraneous large changes which may be caused by measurement anomalies. Future implementations may consider applying various filtering schemes to smooth out state estimates and acheive gaze tracking capability.

Manual Mode

Operating the system in manual mode proved to be more straightforward as it did not rely on camera measurements taken by the Wiimote. In this mode, directional input from the Wiimote D-pad was used to increment or decrement the pitch or yaw of the motor assembly. If the user tried to command either a pitch or yaw angle that was outside of the designed range of motion, the command would not be executed and the Wiimote would rumble



$$\tan(\psi_3) = \frac{y_3}{x_3}$$
$$\tan(\theta_2) = \frac{y_3 + l\cos(\psi)}{x_3 - l\sin(\psi)}$$
$$\tan(\theta_3) = \frac{y_3 + m\cos(\psi)}{x_3 - m\sin(\psi)}$$
$$c = \frac{l_{23}}{l}\frac{\tan(\psi_1) - \tan(\psi_3)}{\tan(\psi_2) - \tan(\psi_3)}$$
$$a = 1 - c$$
$$b = \tan(\psi_1) - c\tan(\psi_2)$$
$$\psi_{user} = \tan^{-1}(\frac{-a}{b})$$
$$x_3 = \frac{l(\cos(\psi_{user}) + \sin(\psi_{user})\tan(\psi_1))}{\tan(\psi_1) - \tan(\psi_3)}$$
$$y_3 = x_3\tan(\psi_3)$$

Fig. 5: State Reconstruction. By obtaining angular position measurements of three LEDs with known offsets, it is possible to fully reconstruct the state in a plane. Range data (x_3) obtained from one plane of measurement can be applied to aid state reconstruction in the other

to alert the user to the fact that they had reached the system bounds.

V. Path Forward

Originally, our project was designed such that the light would track the user's gaze, down to where he or she was looking at the desk. However, we found that the exact solution to this problem proved to be numerically unstable. Moving forward, we could address this problem in a few possible ways. One solution may be to implement data filtering within our control logic. While the Wiimote provides its own data filtering for its camera measurements, that logic may need to be augmented. Another approach would be to explore alternative algorithms to recreate the user state. While we were using an exact mathematical solution, perhaps a numerical algorithm could provide more robust estimates.

Moreover, we could explore other options of tracking the user's face. For instance, if we added an accelerometer to the hat, we could get further insight into what the user's head movements are, supplementing our existing algorithms with more data and reducing the effect of noise. Furthermore, we could investigate facial tracking or eyetracking, incorporating computer image or even video processing techniques to solve this problem. These other solutions could reinforce or replace our current tracking system, reducing the effect of input noise and possibly miniaturizing or entirely eliminating the headgear.

Another aspect that we could improve on is our physical design. Right now, the design is unpolished and unintuitive. The LED bars are mounted on cardboard taped to a hat, and our components are housed in a cardboard box. Moving forward, we would investigate the use of individual LED lights, so that the headgear is much more compact and usable. We would also strive to improve the user interface to SHaZam. Ideally, the user could start it up and enjoy its features with the push of a button. Ultimately, while there are many improvements yet to be made, we believe we have established a solid starting point with our work.