Kobuki Kart

Grant Wang
grant.wang5@berkeley.edu

Nick Riasanovsky
njriasanovsky@berkeley.edu

Rehan Durrani
rdurrani@berkeley.edu

Sean Farhat
s.farhat-sabet@berkeley.edu

I. Overview

The goal of this project was to implement a live action version of the popular Nintendo game Mario Kart [1]. In doing so, we decided to use the Joy-Con [2], Nintendo's proprietary controller for the Nintendo Switch, as our controller, since the newest version of Mario Kart is played with this. Additionally, due to our experience using the Kobuki robots in lab, we opted to use them as our Karts. In order to create an experience resembling Mario Kart as much as possible, we needed to create a smooth driving experience, which differed greatly from the static movement from labs. Finally, we wanted to include characteristic gameplay features, such as hazards and power-ups that can impact yourself or others.

Our work this semester is likely beneficial to anyone trying to produce a live action racing game. This could be especially useful for anyone attempting to make an AR version of a racing game, which we did not attempt given the content of the class and time constraints of the project, but would enable much easier visual identifiers of in-game events.

II. Game Mechanics

The game we designed consists of a track and 1-3 Kobukis. Each Kobuki has a DWM1001C tag for localization, a WS2812B LED strip to display powerups and hazards, and is controlled by a JoyCon controller. The players use their Kobukis to navigate the track and compete to see who can finish first. The track has three elements - a red shell location, a mushroom location, and a banana location. The red shell is a peer-to-peer powerup. When a Kobuki passes by it, it recieves the red shell powerup. When it is used, it targets the nearest Kobuki and causes it to spin for 5 seconds. The mushroom is a normal powerup. Using it allows the Kobuki to speed up. The banana is a hazard - when a Kobuki passes by it, it spins for 5 seconds. After a powerup is taken, it cannot be received by another Kobuki for 5 seconds. After being triggered, the hazard is inactive for 10 seconds.

III. System Design

Our system consists of Joy-Cons for users to enter inputs, a single Raspberry Pi to serve as an aggregator and coordinator, and a series of Karts, consisting of the Kobuki, a Berkeley Buckler mounted on an NRF52832 (for the rest of this paper, we will refer to this combo as just the Buckler for brevity), a Decawave DWM1001C Tag, and a WS2812B LED strip. To interact with our system, the user manipulates their Joy-Con, which pushes its data to the Raspberry Pi over Bluetooth classic. The Raspberry Pi maintains Bluetooth Low-Energy (BLE) connections to each of the Bucklers to forward information about events such as hazards to the Kart. The connections also route information from the Buckler back to the Pi about powerup usage and location information. The Buckler maintains a SPI connection with the DWM1001C to obtain location information, a GPIO connection with the WS2812B to trigger the lights for visual effects, and a connection over an RS232 serial port to the Kobuki for setting the wheel speeds. A diagram of this architecture is shown in Figure 1. We will now discuss the Joy-Con, Raspberry Pi, and Kart in more detail.

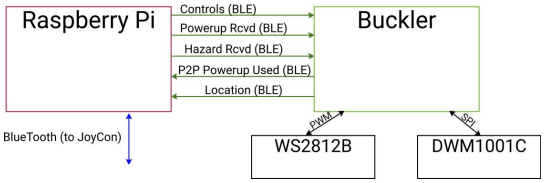

Fig. 1. Diagram of the system architecture, black boxing each component. For simplicity the Kobuki is not shown.

For visual diagrams of our system, please refer to our poster.

A. Joy-Con

The Joy-Con [2] is the primary controller for the Nintendo Switch. It connects to devices over the HID interface using Bluetooth classic. It packages information in 12 byte packets, which we converted into 2 byte packets containing only the information about which buttons were pressed.

B. Raspberry Pi

The Raspberry Pi is the coordinator and master node for our system. It serves two important purposes: providing a bridge between the Joy-Cons and the Bucklers, and working with a global view of the system.

To accomplish this, the Raspberry Pi runs three types of processes. The first type is as a Bluetooth Endpoint, which mediates between between the JoyCon and Bucklers, since the former communicates through Bluetooth Classic, while the latter uses BLE. Upon receiving button inputs, it has to forward this information to the Buckler via the second type of process.

The second type is as a BLE Endpoint which maintains one service per Buckler and keeps characteristics as specified below:

1) PI → Buckler
   Joycon button information
   Powerup received
   Hazard received

Fig. 2. Image of the Kobuki with all peripherals attached.

2) Buckler → PI
   Location information
   Use powerup

By receiving locations from the Bucklers, it serves as the provider for the last process, where all powerup/hazard logic can be done.

The last type, the Aggregator process, keeps track of where everything is, which it receives from the BLE Endpoint: powerups, hazards, and Karts. Upon a Buckler entering within a specified distance from a powerup/hazard, it will communicate that information to the BLE Endpoint, which forwards that to the appropriate characteristic (powerup or hazard received). When a Buckler wishes to use a peer-to-peer powerup, the Aggregator uses its location data to find the nearest neighbor to the initiating Buckler, and sends it a hazard. There is one Aggregator for the whole system whereas there is one Bluetooth and BLE endpoint per user. All communication between these types of processes is done through TCP sockets.

C. Kart

The Kobuki, as shown in Figure 2, serves as our Kart and is controlled by a Buckler mounted on top of it over an RS232 serial port. A DWM 1001C tag is connected to the Buckler over SPI, and a WS2812B LED strip is connected to the Buckler over GPIO. The physical SPI connection between the Buckler and DWM 1001C is formed via connection of the 3.3V power, GND, Chip Select, Clock, MISO, and MOSI pins on both the Buckler and DWM 1001C tag. The WS2812B LED strips come with 3 wires that are connected to the 5V power, GND, and GPIO Pin3 on the Buckler.

Our overall software architecture on the Buckler can be summarized as follows. We first initialize a BLE profile on the Buckler with a service containing the 5 characteristics listed in Figure 1 and Part III-B.

We then utilize the Simple BLE C library [3] to handle the initialization code on the Buckler, and the bluepy python module [4] to read and write characteristics onto the Raspberry Pi. Then, we initialize the DWM 10001C tag for localization and the WS2812B LED strip for visual feedback.

Once initialization is complete, the Buckler code will issue a request to the DWM1001C tag for location data of the Kobuki's current position, which is done every 0.5s. This is done by, via the Decawave Firmware API [5], the following sequence of messages over an initialized SPI connection:

1) Send 2 byte sequence [0x02, 0x00] to request location
2) DWM ACKs by responding with tuple (size per transmission, number of transmissions). These values will be $0x00$ until the return data is ready, upon which they will be updated.
3) Buckler polls DWM for (size, num_transmissions) tuple until they have non-zero values, and then requests size data from the DWM when the data is ready.
4) DWM sends over location data in the form of a 3-tuple (x, y, z) in millimeters w.r.t a fixed (0, 0, 0) setup by anchor network

We perform Step 1 at the start of our main loop before evaluating our FSM and Steps 3 and 4 after evaluation, so that the tag is given the time it takes to evaluate our FSM to produce the data, and so less time is spent busy looping in the polling stage.

The location information from the DWM is then later sent to the Raspberry Pi from the Buckler via the location BLE characteristic notification. The Raspberry Pi will infer whether there is a powerup, hazard, or nothing at the Buckler's current location, and write to the appropriate characteristic if either a powerup or hazard is present. Concurrently, the Buckler receives writes to the button characteristic from the Raspberry Pi, which will determine what velocity and acceleration to apply to both wheels of the Kobuki as it moves via a hierarchical FSM. When the Buckler receives a powerup or hazard from the Raspberry Pi, it will alter the model dynamics by triggering transitions to new states of the FSM.

Lastly, the WS2812B LED strips are used for visual feedback. Updating the colors on the LED requires a series of 24 high pulses on the data signal line that map to 24 bits. Each LED pixel color is encoded as 3 LED brightness values (red, green, blue) and each brightness value is encoded as a sequence of 8 bits in GRB order, hence 24 bits total. The specifications of the WS2812B [6] require a 1.25μs period, and each high pulse encodes one bit. A pulse of 0.9μs represents a 1 bit, while a pulse of 0.35μs represents a 0 bit. Following this specification, we send a PWM signal from the Buckler over GPIO to the data input of the LED strip using the Nordic SDK PWM Driver. We tuned the PWM signal to have a period of 1.25 μs by using a clock frequency of 0.8 kHz on the NRF52832, and use a duty cycle of 28% for a 1 bit, and 72% for a 0 bit.

IV. Course Objectives

A. Networking

Our project encapsulates two networking concepts - Bluetooth Low Energy and SPI. We use Bluetooth Low Energy to communicate information via characteristics between the Buckler and the Raspberry Pi. We use one service per Buckler. The Pi writes to the powerup recieved, hazard recieved, and control characteristics, which the Buckler reads from, and the Buckler writes to the location and powerup used characteristics, which the Pi reads from.

We also use Serial Peripheral Interface to communicate with the DWM1001C Tag for locationing. The Tag can be

communicated with over BLE, UART, and SPI, but we decided to utilize SPI since we were concerned about the lossiness of BLE connections, and the number of connections that the Buckler would have to maintain. The Buckler connects to the Tag as the Master, and sends Type-Length-Value Requests to the Tag. We specify the algorithm to query the Tag in Part III-C.

The process of setting up SPI, from making the physical connections from the pins of the Buckler to the pins of the Tag, as well as reading about and experimenting with the protocol to communicate encapsulates the Networking portion of the course.

B. Modeling Physical Dynamics

Making the Kart drive in a way analagous to Mario Kart's mechanics required a design that allowed for dynamic changes to each wheels' speed. Therefore, we modelled the dynamics via two FSMs, a velocity FSM (V-FSM) and a turning FSM (T-FSM), which, when composed, make up a hierarchical state machine that accepts button presses and powerup/hazard notifications as inputs and outputs wheel speeds, as seen in Figure 3.

The V-FSM centers around a continuous variable, v, representing the total velocity we wish to provide to the Kart. v is updated each step via the X-button providing a forward acceleration to $\dot{v}$, the B-button providing a backwards acceleration, or no button input allowing v to decay to 0. Therefore, the button presses can be utilized to speed, brake, and reverse from any direction. If the Kart receives a Mushroom powerup notification, then the FSM transitions to a 3-state sequence of v first becoming a constant high value, then decaying to the original max velocity, then returning to the normal logic.

The v from the V-FSM is fed into the T-FSM, which distributes the total v among the wheels, depending on which direction the analog stick is pointed in, given by the directional inputs. For example, if we want to turn a hard left, then we will put $0.6v$ into the right wheel and $0.4v$ into the left. If the Kart receives a hazard notification, either by running over a banana or being hit by a red shell, then the v input is completely ignored, and we transition to a Hazard state that will stop the Kart, have it rotate for 5 seconds in place, then return to whatever it was doing. The ouptut of the T-FSM are speeds for each wheel, which are passed into the KobukiDriveDirect(left, right) function.

To see powerups and hazards in action, please see this video.

C. Concurrency

Our project contains concurrency components on both the kart and the Raspberry Pi. On the kart because we do not have access to threading - our two main concurrency mechanisms are polling and interrupts. We use polling to regulate the frequency at which we send and wait for location information, and we use interrupts for receiving data from BLE characteristics. The Raspberry Pi does have access to a Linux based operating system so we have more traditional concurrency mechanisms. We use the fork syscall to spawn unique processes for each user. Additionally we use pthreads to spawn a new thread to a do a polling read for location information and a mutex to provide atomicity guarantees when the location data is used for determining power-up locations or the nearest neighbor.

D. Sensors and Actuators

By using a DWM 1001C as a sensor and WS2812B LED strip as an actuator, our system bridges the cyber and physical realms, a key theme introduced in this course. Key features of the DWM 1001C include a 3-axis accelerometer and UWB antenna. These hardware modules allow the DWM 1001C tag to interface with the real world by taking tilt measurements to determine its acceleration and send/receive ultrawide band signals from nearby anchors. Combining these sensor measurements, the DWM 1001C tag can localize itself within the anchor network and output a position value. Localization relies heavily on precise sensor data, and we demonstrate its usage as a key component of our project.

Driving the WS2812B LED strip via a PWM also addresses the class topic of bridging the cyber and physical realms. LEDs are introduced in this course as devices that require actuation. With PWM, we apply an electrical signal (a physical element) to control the RGB color bit sequence (a cyber element) and actuate the LED diodes. The PWM period and duty cycle values had to be precisely calibrated to match the LED specifications, a design issue that often has to be dealt with when working with sensors and actuators. Furthermore, actuating the LEDs provides a nice visual feedback in the physical world for users.

V. System Evaluation

To assess our system we opted to evaluate the scalability in terms of Raspberry Pi compute time. For all tests we ran our system but replaced our controllers with a simple socket server running on the Raspberry Pi, to ensure more consistent timing. If given more time we also would have evaluated the accuracy of our location data but were unable to do so because the anchor network was deconstructed.

Additionally while we did not conduct a formal user survey, the few individuals who tested our project did note that they were impressed with how smooth the driving of the Kobukis were. Users also noted how they enjoyed the dynamics of the powerups and hazards, along with the visual feedback from the LEDs. However, some users reported that the powerups were sometimes inconsistent, turning could be tricky, and that the lag between button presses was noticeable, but not meaningfully so, with two Kobukis but interfered with gameplay at three Kobukis.

A. Raspberry Pi Compute Time

We evaluated the amount of compute necessary to run our system using the Linux tool vmstat [7]. We tracked the percentage of CPU time spent in either kernel or user space without running our system, as well as with 1, 2, or 3 controllers connected. Then we repeated the experiment

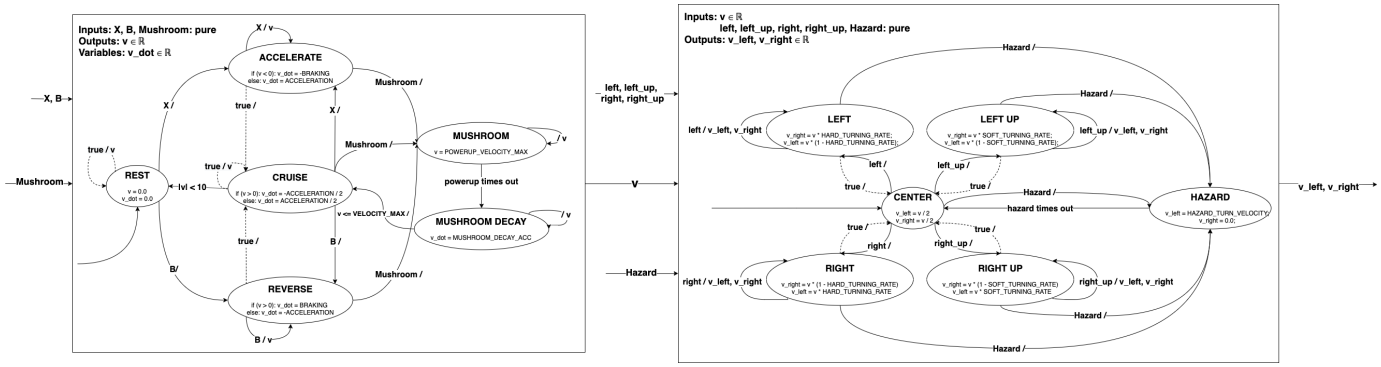

Fig. 3. Finite State Machine Modelling Movement Dynamics

# of Kobukis	Button Press	Mean % CPU	SD %
0	no	2.6949	1.3561
1	no	26.9831	.8334
1	yes	27.0678	.7561
2	no	51.6102	1.0053
2	yes	51.6102	.6643
3	no	75.8983	1.2715
3	yes	75.8305	.9234

TABLE I
RASPBERRY PI COMPUTE TIME

having the controller provide button presses every 2 seconds. Our results are shown in Table I.

Our results indicate a near linear scaling as we increase the number of users. It becomes clear that we would reach 100% utilization with 4 connected users. This means we cannot expect to have 4 users (which is the maximum set by bluetooth connections) and that the performance loss experienced at 3 users is likely due to the Raspberry Pi as a central bottleneck.

VI. Analysis of Success

Overall we believe that we succeeded with most of what we intended for this project. We were able to create a Kart that drives smoothly, place track events on the ground that could be triggered by location and introduce power-ups that work in a peer to peer context using location. Additionally, while we noticed increased lag as we integrated 3 karts, the lag was tolerable for 2 karts, and so we met our scalability target for this project. One of the biggest reasons for our success was the modular nature of our project. Although the Raspberry Pi was a central bottleneck for testing, we were able to easily integrate modular testing of new components. Some examples of this were adding individual characteristics where we could add a single component at a time, such as when we started sending location data, and because we had a lot of spare real estate on our control to implement testing features.

However despite our overall success, when we began this project we had some additional goals we were unable to accomplish. The first was that we initially intended to have more exact power-up and hazard markers, marking almost exactly where they could be acquired. Unfortunately we found that the location data provided by the DWM1001C was too noisy to be able to do this reliably and that our power-ups would always move slightly. This is not necessarily a bad game mechanic as it adds some randomness, but it was not what we originally intended. We also were unable to complete our stretch goal, which was maintaining a race leader board and a subsequent power-up. This proved to be too difficult in the time allotted because the location data could not be used to map out a track without a change of coordinate system. When coupled with the noise in the location data we could not construct an accurate enough model of the track to implement a leaderboard.

VII. Conclusion

In our project, we attempted to recreate the popular video game, Mario Kart. Fortunately, this allowed us to leverage multiple topics from what we learned in EECS 149: Networking by communicating with devices via interfaces such as BLE, SPI, and GPIO, Modelling Dynamics via a hierarchical FSM that allowed for smooth driving, Concurrency via several simultaneous Karts driving at the same time and the load balancing of communication lines required, Sensors by utilizing DWM tags to get the locations of Karts, powerups, and hazards that allowed for interactive gameplay, and Actuators by using LED strips to notify players of powerups and hazards. By touching on a myriad of topics, we were able to delve further into the subject matter and apply our knowledge to create a robust codebase for anyone to leverage to build their own improved Kobuki-racing system.

References

[1] Nintendo. (2019) Mario Kart 8 Deluxe for Nintendo Switch. [Online]. Available: https://www.nintendo.com/games/detail/mario-kart-8-deluxe-switch/

[2] ——. (2019) Technical Specs. [Online]. Available: https://www.nintendo.com/switch/tech-specs/

[3] N. Jackson. (2019-8-28) Simple BLE. [Online]. Available: https://github.com/lab11/nrf52x-base/blob/master/lib/simple_ble/README.md

[4] I. Harvey. (2014) bluepy - a Bluetooth LE Interface for Python. [Online]. Available: https://ianharvey.github.io/bluepy-doc/

[5] Decawave. (2019) DWM1001 Firmware Application Programming Interface (API) Guide. [Online]. Available: https://www.decawave.com/wp-content/uploads/2019/01/DWM1001-API-Guide-2.2.pdf

[6] Worldsemi. WS2812B Intelligent control LED integrated light source. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf

[7] H. Ware and F. Frédérick. (2011-9) VMSTAT(8) Linux Manual Page. [Online]. Available: http://man7.org/linux/man-pages/man8/vmstat.8.html