NDL report

Frank Gerlings Justin Reniers Janneau Thijssen

April 2018

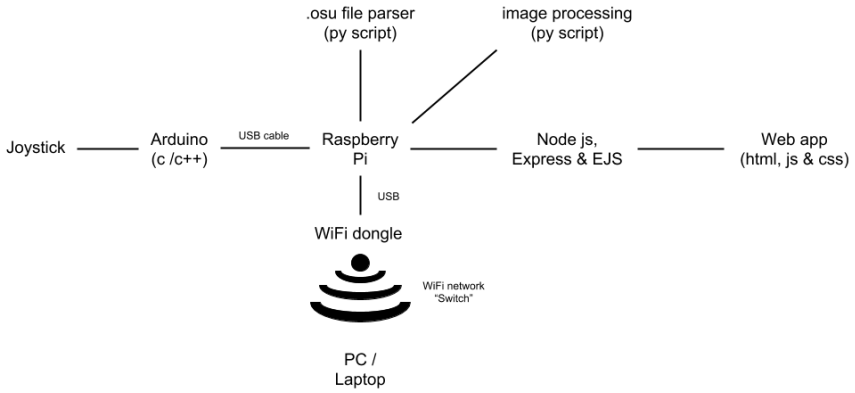
Contents

1 Introduction

We first started out this NDL project with the idea to program on the Nintendo Switch. In particular with the Nintendo Labo[1] in mind. This however turned out to be impossible so there were a lot of changes in our project along the way. In the end we built a ukulele/guitar using an Arduino and a Raspberry Pi, though it is good to know that some of our choices we're based on the idea of mimicing the Nintendo Labo project.

In our contact with Nintendo, we mailed them to ask if it would be possible to get a development kit for the Nintendo Switch for the project. They responded by asking if we could send them some concepts. We drafted a response, which can be found in the zip-file. They then replied to us by telling us they did not yet have any protocols for the Switch and Labo development kits and asked which one we were looking for. We responded that we wanted the Switch development kit since we expected that with the Labo kit there would probably be to much work already done for us. Nintendo responded by saying that they couldn't give out the development kits yet, thus after 3 weeks of mailcontact leaving us with nothing really.

Figure 1: Schematic displaying connections in our project



2 Product

Making the cardboard structure for the ukulele seemed a little harder than first expected. We had an idea of buttons that could be pressed down which would be pushed up again by a spring. The buttons would have a white mark on them, which would set them off to the black background, making them easy to read for the qualitatively bad IR camera of the Nintendo Switch.[2] We however found out that the spring mechanism would rip the cardboard, making it not very durable. Next we wanted to make a structure where you could put your finger in some sort of ring (made of paperclips) which are stuck to pieces of the cardboard that could be pushed down. We tried multiple instances of this idea, but those were all fruitless too. Finally after a lot of different prototypes making holes in the top of the structure with the the camera pointing to them from the bottom seemed to work the best. It now detects whether a button is pressed by checking which holes are blocked by a finger. This does however rely on the higher quality camera of the Raspberry Pi, rather than the IR Motion Camera of the Right Joy Con.

2.1 Arduino

The Arduino is supposed to mimic the Left Joy Con, which allows the user to tap. For this we used an Iteaduino BT, the Arduino IDE[3], our knowledge of the first NDL exercise, a joystick and a USB serial cable.

The code given in `Guitar/Left_Joy_Con/guitar_left_joy_con.ino` consists of three parts. In the first part, all variables are initialised. The second part executes the loop. The last part consists of some functions that where used to test a Bluetooth connection.

We tried to mimic the Left Joy Con of the Nintendo Switch, which has Bluetooth for communication with the Right Joy Con, in our case the Raspberry Pi. Therefore we used an Iteaduino BT, which is capable of sending a Bluetooth signal. However, using an Iteaduino is not as straight forward as we'd like.[4] The board has three different operating modes, which can be changed by switching some jumpers. One for uploading scripts, one for configuring the

Bluetooth settings and one for actually sending data over Bluetooth. This has two implication. First off it means that you cannot upload a Bluetooth based script and directly run it, since you still have to tediously change the jumpers. Secondly, in Bluetooth sending mode you cannot use the serial bus anymore. If for some reason your Bluetooth signal isn't caught, you'll have to do the debugging through the blinking LED. It even took us a while to figure out if our script actually ran.

Eventually we dropped the idea of using Bluetooth, since it had consumed quite some time and hadn't shown any results. Instead we used a USB serial cable to send the data.[5] This turned out to be easier to configure than the Bluetooth variant.

2.2 Raspberry Pi

For our project we are using a Raspberry pi 1 to mimic the Nintendo Switch console and right joy con. We installed Raspbian stretch with desktop as operating system. We use a cable connection with the Arduino and a LAN cable for internet. We also tried to run the python script that will be discussed in the next subsection at start up. We attempted this by editing the rc.local file in the etc map, but this didn't seem to work and gave some odd issues.

2.2.1 Image Processing

The python IDE that is automatically installed with the raspbian operating system comes with a useful library Picamera. You just need to initialize the camera and set the resolution.[6] Then we want to get input from the Arduino, we can easily get this by importing the serial library and opening the correct port. Finally we set the initial values of each button to 0 and make a general setup for the JSON file output. We use the simplejson package for this.[7]

The code given in `Guitar/Right_Joy_Con/Bright_spots.py` waits till it gets the first input, then it processes the first image. This image sets the locations for the "buttons". After this there is a continues while loop that waits for the Arduino's input. After it gets this it takes a picture and looks at the button locations to check if the buttons are pressed.

When the script receives the input form the Arduino it take a picture and save it in the same map as the script is in. Then we load this image in from the map, we first tried to directly take the picture as input, but we couldn't figure out a way to do this. Then we process the picture to be black/white, where the bright spaces become white. It then removes all the little bits and pieces of white that are too small. It then finds the contours of the white spots and orders them from left to right. Now it will find the enclosing rectangles of the white spots and takes these as the places of the buttons. This can be any number, so by pressing some buttons during the initial photo some buttons can be removed for this run of the program (or added if you make an extra hole). Then it sets the length of the button results list to the amount of buttons.

We then enter a continues while loop which listens to the Arduino input. When it gets this it takes a picture, makes it black/white where the bright spots are white. It then scans over the areas the buttons should be and if there is enough white, it is counted that that button isn't pressed. If there

isn't enough white it is counted as if the button is pressed. After all areas are scanned the results are passed to the web application.

2.2.2 Parsing

When making a game, especially a rhythm game, it is important to have a song that you want to play. We took the liberty of taking an already existing file format, .osu files, to parse and create a data structure for our song with[8]. The structure of these files is nicely documented [9], thus understanding what was in such a file and what we needed to parse out of it was doable. We decided to only look at the information contained under the [HitObjects] header, since these are the objects that represents the notes that are to be hit. The important information we needed here was the x coordinate, or which of the four keys to hit, and the time it should be hit.

The parser named `Guitar/Console/Website/views/ParserAndResult/Parser.py` is relatively small, as we only need a small portion of the whole file. It tries to open the specified "song". It then searches for the line containing the [HitObjects] header. Once found, for every HitObject (every line after the header), we take only the x coordinate and time. The x coordinate is mapped to the corresponding key, and all that is put in a dictionary. After all of that is done, it writes it to a JSON file.

Sadly, we haven't reached the point of using this JSON file in the web app. The parsing itself, however, does work and is easily used on other .osu files.

2.2.3 Hosting the web app

To host the web app so that not only the pi, but also those on the same network as the pi can access it, we needed several things. First off, we decided to make the pi into a standalone network. For this we needed the packages hostapd and dnsmasq, both available for the raspbian operating system. Following a tutorial online, we set up a bridge between the ethernet connection of the pi, and the network that the pi was broadcasting. Now we could all connect to our personal pi network.

To actually host the web app, we needed Node js: a JavaScript run-time environment that executes JavaScript code server-side. However, since the web app was separated into multiple files (.html, .js and .css) that referenced each other, Node js alone wasn't enough. We quickly found out that we would need 2 more frameworks to allow for the communication between these files, as well as communication between the json files on the pi and the web app. These frameworks are Express and EJS, or Embedded JavaScript. Express allows for a better integration of other templating engines, such as EJS, and creates an easier link between the front end and back end of the application. Since we have the parser and image processing running on the server end (back end) of our app, and the user interface is running on the front end, we needed this framework to connect the two. EJS is needed to add dynamic content to the web app. Since the notes we play on our guitar are constantly updated, we'd have two options. We could either refresh the app every time new notes are sent, which would be woefully inefficient, or use EJS to have dynamic content running. The choice was easy for us, and we chose to use EJS. So once we run the web app with Node js, we set up an IO socket connection. Thus, our server

can dynamically listen to the changing content and use it. The server-side part of the application receives information on what notes are pressed from the script that does the image processing, and then passes it on to the .ejs file that's loaded in where it is used to show the clients connected with our web app.

2.3 Web application

The web application's main goal is to provide a more interactive and engaging experience for the user of the guitar. Although the notes from the hardware could not be read yet, we already started working on the web app. In the end we weren't able to complete the data transfer, so our web app remains as a standalone project.

The JavaScript libraries we used are jQuery [10] and Howler [11]. We use these to read in the .mp3-files.

The HTML-file is replaced by an .ejs file for earlier mentioned reasons. The files can be found in `Guitar/Console/Website/views/guitar.*`.

The HTML-file consists of two main parts, the canvas and the horizontal menu. The bottom of the canvas shows the buttons when they are pressed. Above these buttons is a field that shows all strokes of the currently played track that will be played in the next three seconds. A note starts at the top of the screen and at the moment that note should be played, the note is positioned directly underneath the button that becomes visible when hit. This way, the user can see which notes he has to play at what time.

The second part of the HTML-file is the horizontal menu, which consists of three functionality categories. The first category is a dynamically generated dropdown menu that contains all currently tracks. Clicking on one of these should start playing the track and, for the one track you start off with, play the corresponding music. Sincerely, there are multiple bugs in this menu. First off, it shows all menu items on top off each other, meaning only last track is shown. Secondly, we couldn't dynamically give the buttons onClick events. The second part of the horizontal menu allows the user to create tracks on itself. The track can be given a name and the recording can be restarted. A button called *test* will show the buttons of the first saved track. The last part of the horizontal menu is dedicated to playing an .mp3-file. There are buttons to play, pause and stop the music and the volume can be turned up and down.

The CSS-file mimics the structure of the HTML-file. It's pretty straight forward, so we will not elaborate on them. The complicated stuff happens in the JavaScript file, which we will now discuss.

The JavaScript-file, guitar.js, is quite large to be honest. We used a Model-Controller-View approach. The first thing we do in the model is time normalisation. Since we are going to play tracks in a time-based game, we need some helping functions. Next we define two classes. The first is the class Stroke, which links a button to a time since the start of the loading of the page. The second is the class Track, that consists of a name and a list of strokes. Lastly, we a list of all available tracks, on which the dropdown menu will be generated. In it is one song that should be read from a .json-file and one test track. It should be noted that we did not finish the json reading.

The controller consists of three parts, two for the canvas and one for the horizontal menu. To test the canvas without being able to read the Raspberry Pi input, we made a temporary input mechanism. Pressing q, w, e or r would

correspond to a button on the guitar. A second way to control the canvas is by playing the guitar, this will result in incoming socket information. The handlers for the horizontal menu are more straight forward and influence the tracklist. The second part of the code uses jquery and the Howler library to manipulate the music, as mentioned earlier.

The first part of our view consists of printing methods for our object. Sincerely we could not figure out how to overwrite the toString function of our classes, so we defined the printing statements here. Next we have some code showing the strokes of the song that is about to be played. We regrettably couldn't finish this functionality, although the current version does show all the strokes pressed within a song. This can be tested using the *test* button. The last part of our view generates the dropdown menu containing all the playable songs.

3 Course goals

In this chapter we will reflect on the course goals. We will explain our process throughout the course and why it helped us understand and use the architecture of the given devices.

After some fidgeting around with both the Raspberry Pi and the Arduino, we already got some small scripts up and running on both of these devices. The script on the Arduino was fairly easy, especially after having done that before in the warm-up assignment. The Rpi script took a little more time, but luckily the obvious choice was python, with which all of us had some prior experience. However, it took us multiple methods to find out that the communication between these different types of hardware devices is not as easy and clear-cut as we might have initially thought. At first we tried setting up a bluetooth connection between the Rpi and the Arduino, but this proved tough. Eventually we chose to take a more direct approach, and connect them via a USB cable. This allowed us to put the input that came from the Arduino hardware onto the serial bus, and send it to the Rpi. However, since we wanted to make another communication step to the world online, we needed more. We found out that having a dynamic web app that takes shows input provided by a "client" device needed intricately structured software, that makes use of packages that have been developed specifically this all.

But having overcome this hurdle, we now have the Arduino that takes input from the user that comes from the actual hardware. The Arduino then sends those inputs to the Rpi via a serial bus. The Rpi reacts to those inputs with possibly taking a picture. It then processes that picture, and sends the results of that processing to the web app server side via sockets. And then finally, the web app server then sends this information to the client where the web app is loaded, so that the user has a graphic representation of their actions.

4 Conclusion

We have built a link between the shields of the Arduino that take real world input, the Raspberry Pi and even a web app for graphical presentation. In our efforts to find out how Nintendo might have programmed their Nintendo

Labo ideas, we found out that the link between software and hardware is not as easy as it looks, and that hardware (sometimes even less than software) doesn't always do what you want it to do. Hardware has more possibilities to fail, since it can be faulty or defect compared to software that is badly designed or buggy. The fact that the hardware doesn't specifically point out that it is defect also makes that you have to find this out for yourself. We built a product that tries to emphasize the communication between multiple devices, and designed an abstract script that uses image processing to recognize and process the number of "bright spots", to then pass this on to the web app. We underestimated how much time all of this would cost, but in the end, we were able to bring it to fruition and solidify the link between everything that we had in mind, and make it work.

References

[1] Nintendo, "Nintendo labo," https://labo.nintendo.com/, last accessed: 2018-04-15.

[2] ——, "Nintendo switch: Technical specs," https://www.nintendo.com/switch/features/tech-specs/, last accessed: 2018-04-15.

[3] T. A. Community, "Arduino ide download page," https://www.arduino.cc/en/Main/Software, last accessed: 2018-04-15.

[4] B. Bellamy, "How-to configure and use an iteaduino bt," http://techwatch.keeward.com/geeks-and-nerds/how-to-configure-and-use-an-iteaduino-bt/, last accessed: 2018-04-15.

[5] O. Liang, "Connect raspberry pi and arduino with serial usb cable," https://oscarliang.com/connect-raspberry-pi-and-arduino-usb-cable/, last accessed: 2018-04-15.

[6] R. P. Foundation, "How to plugin/setup and use the camera module," https://projects.raspberrypi.org/en/projects/getting-started-with-picamera, last accessed: 2018-04-15.

[7] B. Ippolito, "simplejson documentation," https://simplejson.readthedocs.io/en/latest/, last accessed: 2018-04-15.

[8] Feerum, "Alone - alan walker [feerum]," https://osu.ppy.sh/beatmapsets/637917/#mania/1353324/.

[9] D. Herbert, ".osu (file format)," https://osu.ppy.sh/help/wiki/osu!_File_Formats/Osu_(file_format)/, last accessed: 2018-04-15.

[10] T. jQuery Foundation, "jquery," https://jquery.com/, last accessed: 2018-04-15.

[11] J. Simpson, "howler.js," https://howlerjs.com/, last accessed: 2018-04-15.