Junior Design Final Project – DoodleBob

Lauren Duffitt

**Design Overview**

The DoodleBob is a personal drawing device used for recreational doodling. It is interfaced through a touchscreen and has a variety of drawing options such as changing the color of the pen, changing the size of the pen, changing the background color, and saving the drawing. The user may save as many drawings as they want, and additionally DoodleBob can be connected to a Kodak Step™ printer so that the user can print their drawing as a sticker.

The final DoodleBob product is very similar to the initial design proposal; I was able to implement all of the drawing features that I wanted and it can be used as originally intended. Originally, the user was supposed to print directly to the printer from the user interface and without any inputs from outside of the program. However, the mobile printer that I purchased did not implement direct Bluetooth printing, and instead utilized an app to process images before they were printed. To work around this, I wrote code to allow DoodleBob to connect to AirDrop; this way the user can AirDrop their drawing to their iPhone, and then print from there if they would like to.

The first two-three weeks of work were designing the code for the drawing program. Since I was utilizing packages I had never worked with before, I spent most of my time researching the most effective way to implement my code. Once I had a working prototype of the program on my personal computer, I transferred it to a Raspberry Pi 3 to begin working on the standalone product. The rest of the time was spent configuring the Raspberry Pi to be oriented to the user -meaning that the DoodleBob program will execute as soon as the Raspberry Pi powers on- and configuring a way for the DoodleBob to print to the printer. This stage took more time than expected, as I do not have a lot of experience with Raspberry Pi’s and configuring hardware through Bluetooth.

**Preliminary Design Verification**

The original prototype for the DoodleBob was solely code implemented on a laptop rather than on the Raspberry Pi. I decided to implement DoodleBob using Python, Qt Creator, and PyQt6, as I have experience working with these programs from ECE1140. My first step of the project was to set-up a GitHub repository to ensure I was properly backing up my code and could better track errors as they occurred. My second step was to begin coding the project; my original idea was to implement a grid through the QLayout widget in PyQt6 as shown below.

Calendar

Description automatically generated

The original prototype had buttons stored in each square of the Qt grid. When the user clicked on each button, its color would change and give the appearance of something being drawn. However, I quickly realized that if I were to use this design method, then I would have to make a button for every pixel I wanted to represent, and that would be extremely inefficient to code as I would have to make an attribute for every single QButton widget and assign it a place on the grid. Instead of using a button implemented approach, I decided to track the mouse moving across the screen, and use the cursor coordinates to directly modify the pixels of an image. I used the pyautogui package to track the cursor coordinates and implemented a rudimentary PyQt user interface to allow the user to draw or change the color of the pixels; the code is shown below.

Text

Description automatically generated

This version of code was to test if the image -titled newDrawing- was being created and if the mouse was accurately being tracked, so it did not actually draw anything on the generated image. Once I confirmed that these two elements were functional (demonstration of the pyautogui tracking can be seen below) I moved onto implementing the drawing function.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

As shown in the prototype code, I orginailly implemented the QThreading function so that the user could operate the menu and draw on the image at the same time. I kept this functionallity when developing the drawing function, with the hopes that both the drawing function and the menu could operation simultanesoly. The code below shows the changes to the Worker class – which is a crucial part of the thread- and how the drawing function operates seperatley from the menu function. When I implemented the drawing function it was unsuccessful, as the user could not see real time updates of their drawing. While I could see a few pixels had been changed when looking at the saved image, shown below, the drawing was not consistant, and the functionally of the code was not practical enough to be used by the user.

Text

Description automatically generatedA screenshot of a computer

Description automatically generated

**Design Implementation**

**Design Testing**  
  
**Summary, Conclusions and Future Work**