Software Design

2/11/22

Team Code Duckies

Team Members: Anthony Simard, Ari Jaramillo (lead),

Daniel Rydberg, Chris Cisneros, Jacob Heslop

Sponsor: Dr. Truong X. Nghiem

Mentor: Vova Saruta

Table of Contents

1. Introduction

The automotive industry is constantly changing. There are advances in safety features that are installed and required in vehicles, computers installed to better diagnose engine trouble as well as recent improvements and installation of collision avoidance systems, or self braking. Each of these features helps reduce the damage caused when an accident occurs; accidents often caused by human factors such as distracted driving, delayed reaction times, falling asleep at the wheel, or driving drunk are the main causes of vehicle accidents. Collision avoidance systems are one of the first steps in autonomous, or self-driving vehicles which can ideally reduce the number of vehicle crashes from human factors. The research for these types of vehicles is costly and requires a lot of professional work in order to produce cars that integrate well into society. Because of this, for the average person, it is difficult to experience what it might be like to create or develop such cars, especially those in the younger generation.

Dr. Truong X. Nghiem, an Assistant Professor for the School of Informatics, Computing and Cypber Systems at Northern Arizona University (NAU), created a plan to use small, simple, and cheap robots developed at Massachusetts Institute for Technology (MIT) called Duckiebots to teach kids about developing self-driving vehicles. These Duckiebots were created to use software and an environment called Duckietown OS (or Duckietown libraries), autonomous vehicle software and a small, physical map that can mimic roads for the Duckiebots to run on. This software is still complicated, requiring understanding of programming languages such as Python or C++, robotic dynamics, computer vision, or other general robotics that may be difficult for younger students to enjoy and grasp who may want to experiment with these cheap robots. To help relieve this, Dr. Nghiem proposed further development of a system of coding using a visual block command system to simplify the coding process of the Duckiebots.

The visual block command system that has been proposed should allow for younger students in junior high or high school to be able to affect the code that will run on Duckiebots. The goal is to create a user interface of blocks that can be dragged from a library into an area for development. Each block will have a simple command shown on it (eg. “move forward” or “check for”) and can be connected together to create a string of commands. The simple commands seen on the blocks will actually have the complicated programming that is part of the Duckietown OS as well as ways to communicate with the instructions to Duckiebots on the backend. By rearranging, dragging and dropping new blocks, and running the commands and codes students can “develop” a working program that can run on Duckiebots. This working program will be sent to a specific Duckiebot connected either directly to the computer or over a wireless network so that students can see how their program affects a self-driving vehicle.

2. Implementation Overview

We are building a web application that utilizes the visual programming tool Blockly as a front-end for a roslibjs back-end that connects via rosbridge to a ROS (Robot Operating System) application running on a small autonomous duckiebot. This Blockly front-end allows the user to create a visual program that translates to commands that control the bot. This gives the user basic control over the bot without them ever having to write a single line of code.

Blockly is the primary front-end tool we are using for this project. It allows us to create blocks that snap together intuitively and execute pre-written scripts on the back-end. The blocks allow for basic if/else and looping constructs. They are also parameterizable with the values entered into the block being passed to the back-end script the block executes.

In our case these scripts will execute as JavaScript on the back-end. This is so we can make use of roslibjs which allows us to communicate with ROS via JavaScript in the web browser. Blockly allows us to easily wrap up chunks of JavaScript code that make use of the roslibjs library in the visual blocks the user will be interacting with.

This communication between JavaScript and ROS occurs using an app called rosbridge that packages the communications between roslibjs and ROS in json and facilitates the communication between the two. This happens invisibly in the background with no extra work needed on our part. We just need to ensure that rosbridge is running and that we are connected to it.

ROS stands for Robotic Operating System. This is the software that runs on the duckiebot and controls its various systems (camera, motor, etc.). It is designed to be highly modular and configurable which is why it is possible for us to send it messages from a web application running on a separate computer in the first place.

3. Architectural Overview

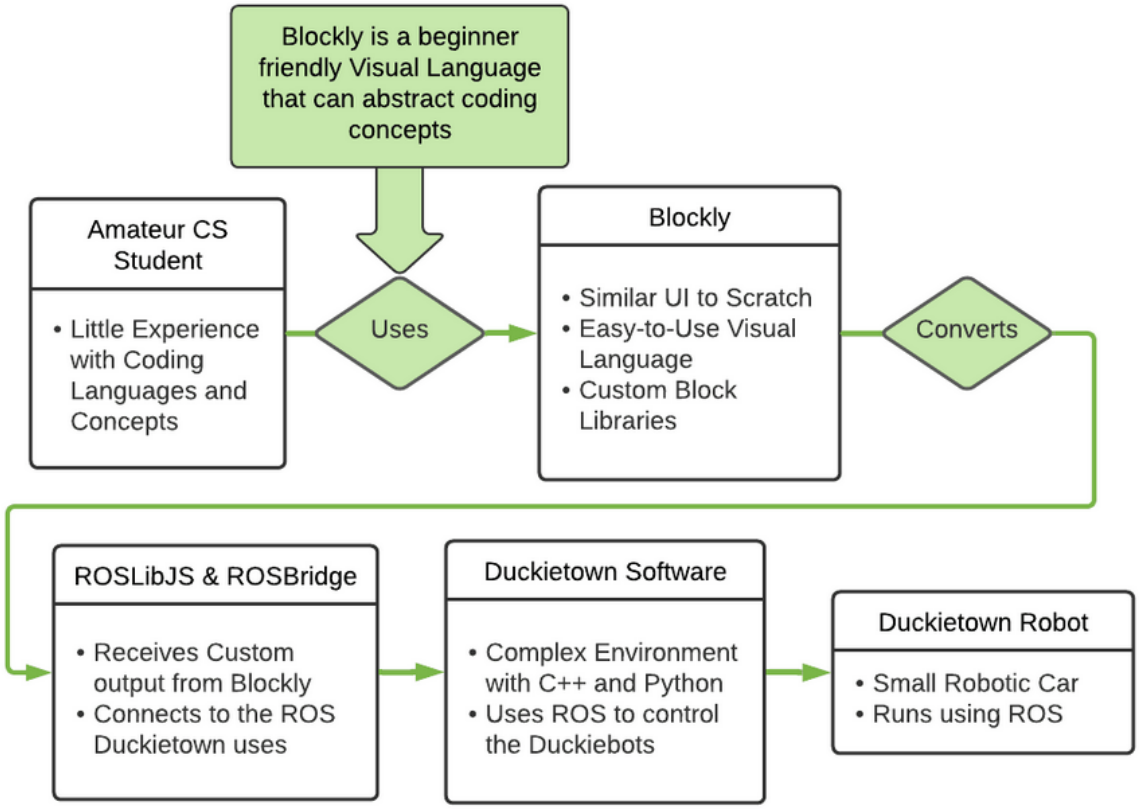

Figure 1: Basic architecture overview

As mentioned previously, the key components of our system are the Blockly based web app front-end, the roslibjs based JavaScript back-end, the ROS application running on the remote autonomous duckiebot, and the rosbridge link between the JavaScript web back-end and the ROS application.

In more detail, the Blockly front-end connects directly to the roslibjs back-end. This connection is handled by Blockly itself. Blockly facilitates the execution of arbitrary code on the back-end of a given block in a block-based program. Blockly natively supports execution of JavaScript on it's back-end making it trivial to make use of roslibjs library functions on the back-end of a block. This portion of the system runs on the web browser of the end user's computer.
Roslibjs must then communicate from the user's computer to the ROS application running on a remote duckiebot. This communication is facilitated by rosbridge. Rosbridge wraps ROS messages in json and transmits them over the network from a port on the back-end of a web browser to a ROS application and vice-versa. Rosbridge can be directly invoked by roslibjs on the user's end and ROS on the duckiebot's end.

The duckiebot itself runs a ROS application that will be receiving commands from the user's Blockly application over rosbridge. These commands will control the motors that turn the wheels on the duckiebot allowing it to move as instructed by the user. The ROS application will also be gathering data from the duckiebot's camera and sending it to the web app via rosbridge. This will allow the user to process the camera data in Blockly and gain greater control over the duckiebot allowing it to follow lanes in a road and, as a stretch goal, actively avoid obstacles.

4. Module and Interface Descriptions

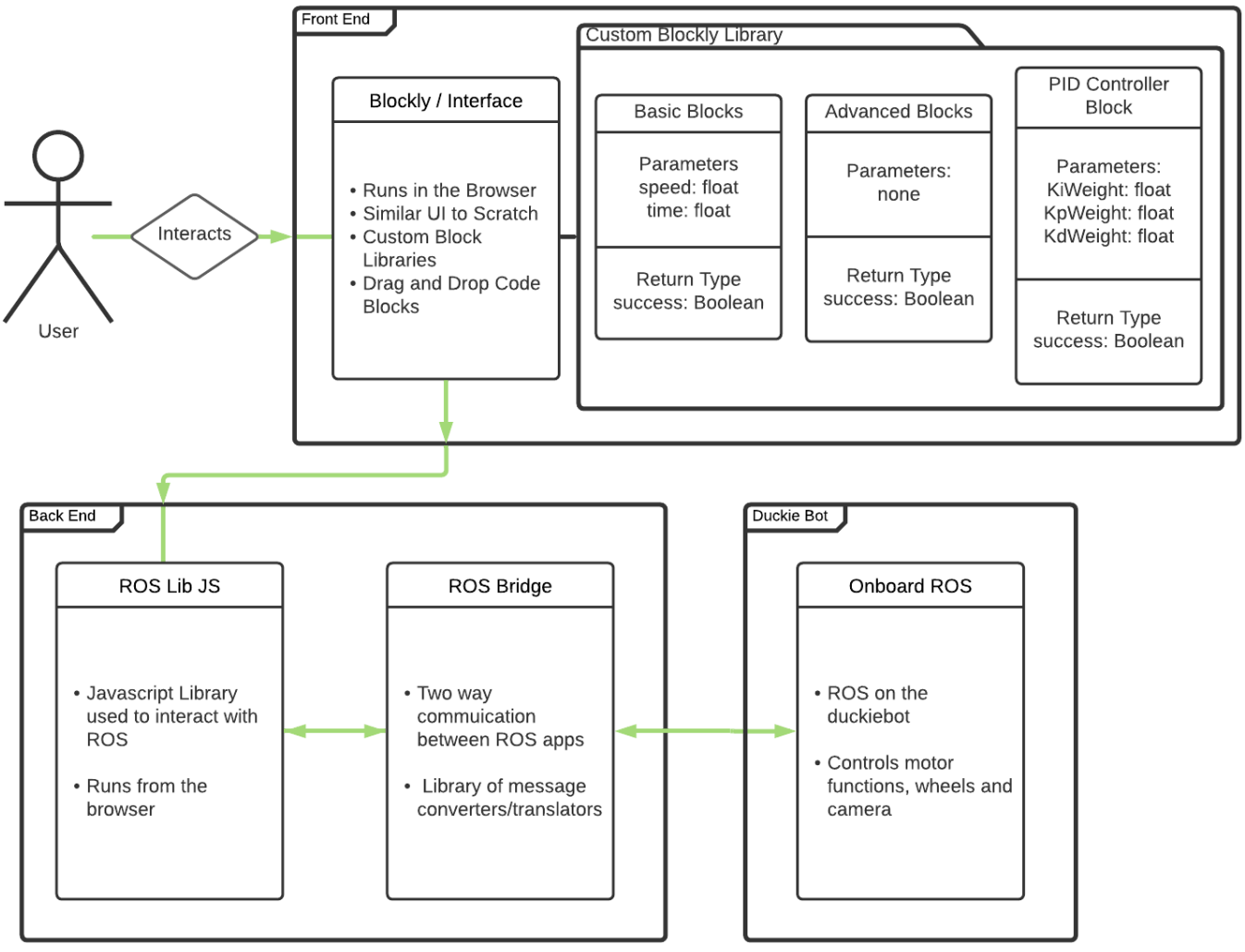

Figure 2: In-depth diagram of the sub programs and how they are related

Figure 2 visually demonstrates the full architecture we plan to implement for both our frontend and backend. The front end consists of all the programs we will utilize and programs we will build that the user will interact with. For this reason, the custom blocky library is included here since our user will interact directly with blockly and by extension the blocks. Using the blocks, the user will be able to build programs that the duckiebots can interpret. However the communication with the duckiebots is overseen by the programs that make up our backend.

The backend in the diagram consists of ROSLibJS and ROS Bridge. Since blockly runs in the web browser, it uses JavaScript for most of its functionality. To get this JavaScript code into a form that the duckiebots can recognize, we use ROSLibJS. ROSLibJS is a JavaScript library that allows for a web browser to interact with ROS, the program that duckiebots use to move and function. Now we can use ROSBridge, giving us a way to connect the ROS commands made in the web browser to the ROS instance on the duckiebot. Lastly the ROS on the duckiebot simply needs to follow the orders it receives. Now we can create simplistic programs in blocky with the custom library and send them to the bot for execution.

4.1 Description of Blockly frontend

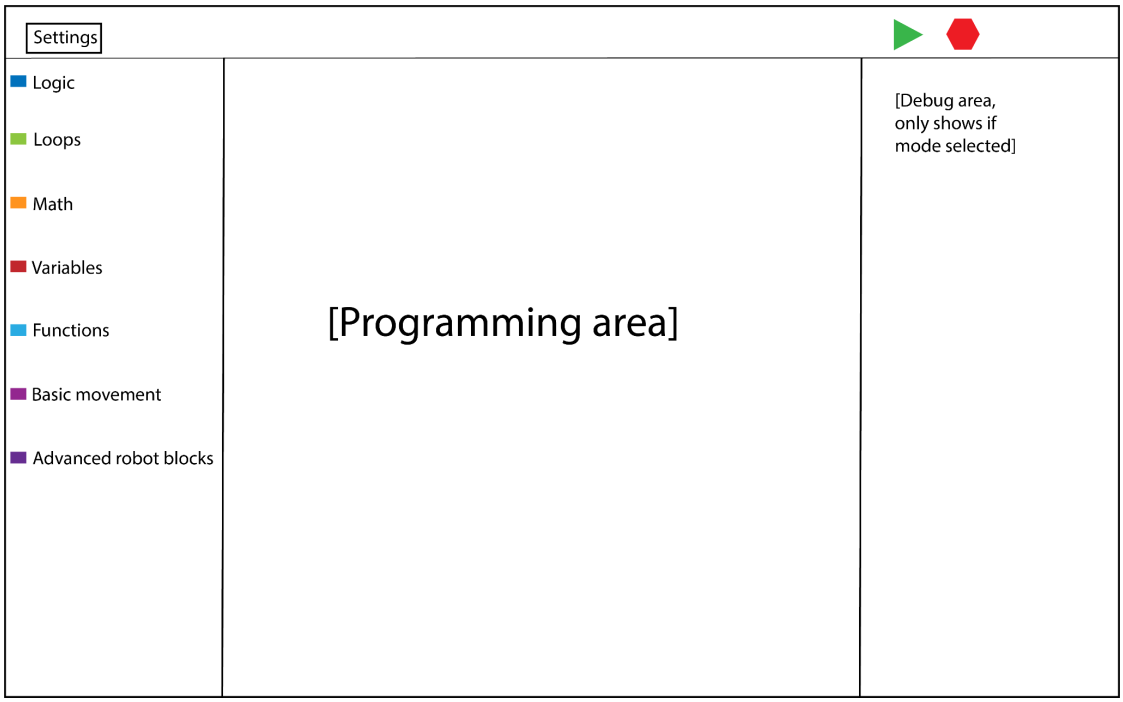

Figure 3: Diagram of user interface

There is only one user interface shown in figure 3 as the focus is the visual programming tool. When the user loads into the program, they will be greeted by a

blank programming area with a library of blocks on the left-hand side. On the top of the program is a toolbar where users can access settings and start/stop buttons.

On the left side of the tool, a library of blocks is presented by sections such as logic, loops, basic movement, etc. Programming blocks will be in related sections, for example: an if-else statement will be located in logic. Then, the programming area allows for users to drag and drop blocks into it in order to create a program. To send the program to the duckiebot and have it run, a user will need to click the start button on the right-hand side of the tool. If a user wants to stop the program mid-run, then they will press the stop button next to the start button which will send a command to the duckiebot to have it stop moving.

In the top toolbar, when a user clicks on the settings button, a dropdown menu will appear. That drop down menu will have an option to toggle the debug screen shown in figure 3 and configure duckiebot. The debug area, when toggled, will be on the right-hand side and show what sort of code is being produced by the blocks in the programming area. Configuring the duckiebot will involve inputting information required to connect the visual programming tool to a specific duckiebot.

4.1.1 Blockly builtins

The Blockly builtin blocks are categorized into logic, loops, math, variables, and functions. What sort of blocks are in each one should be self-explanatory. Logic will have blocks that provide logic statements such as if-else statements. Loops will have blocks that provide while and for loops. The type of input each builtin block takes will vary depending on the type and what is required for its functionality.

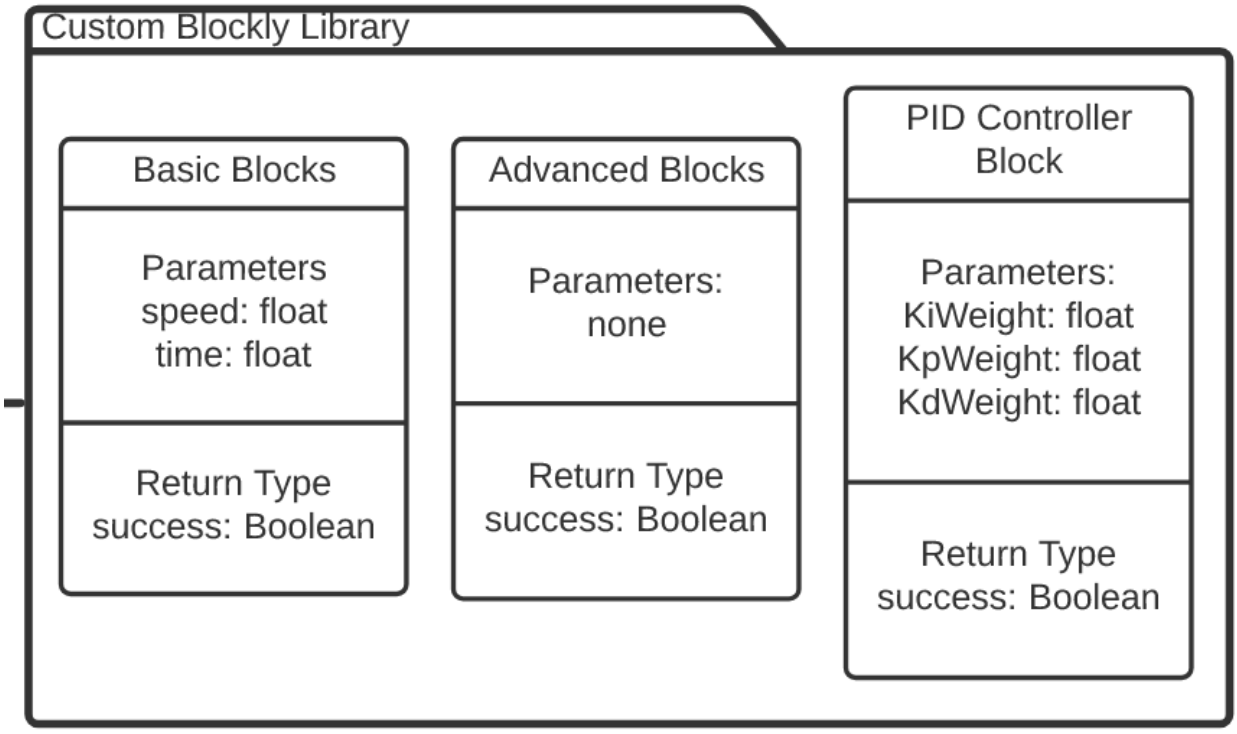

Figure 4: Diagram of the custom blockly library

4.1.2 Duckietown blocks

The Duckietown blocks can be broken up into 3 categories, basic blocks, advanced blocks and a PID controller block. Together they make up our custom blockly library. The basic blocks take parameters that correlate to speed and time, and usually consist of very simple movements of the bot. Things like moving forward or turning for x amount of time at y speed.

Our advanced blocks won't take parameters because they involve commands like stop at the next stop sign, or follow a lane. The fine tuning for these advanced blocks comes from the PID controller block, which takes 3 weights that correspond to the bots past recognition, present decision making and future predictions. These weights all fine tune how reactive the bot is when executing the advanced blocks.

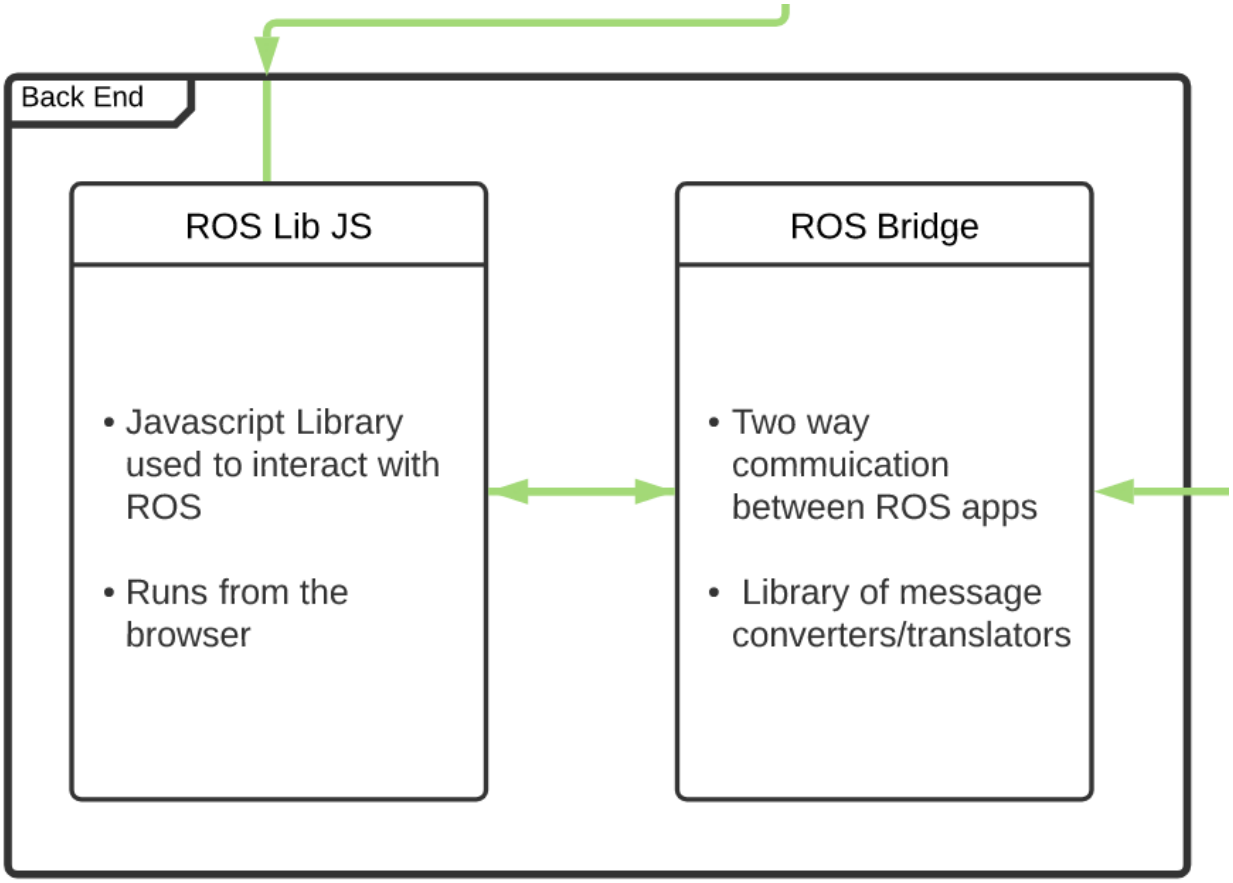

Figure 5: Diagram of roslibjs backend

4.2 Description of roslibjs backend

The roslibjs backend is what actually runs when the user executes their visual Blockly program. Their visual program will have templated out a JavaScript program that makes use of the roslibjs library which is capable of sending and receiving ROS messages over rosbridge. It is responsible for actually executing the commands the user indicates in their visual program.

When only making the bot do basic pre-defined movements, the backend will only be sending messages to the bot. It will not be receiving or interpreting the camera output or any other diagnostic output from the bot.

When performing lane following functions, the bot will be sending its camera feed to the backend via rosbridge where it will be interpreted to determine where the lanes are relative to the bot.

4.3 Description of rosbridge link between backend and ROS

Roslibjs is not capable of directly communicating with the ROS program running on the duckiebot, it requires a piece of middleware called rosbridge to facilitate the communication. This is because roslibjs is not really creating ROS nodes in the same way as a real ROS program. It is mimicking the behavior of ROS nodes and using a websocket on the backend of the browser it is running in to send and receive messages. Rosbridge is capable of translating roslibjs messages into a form ROS can understand and vice versa.

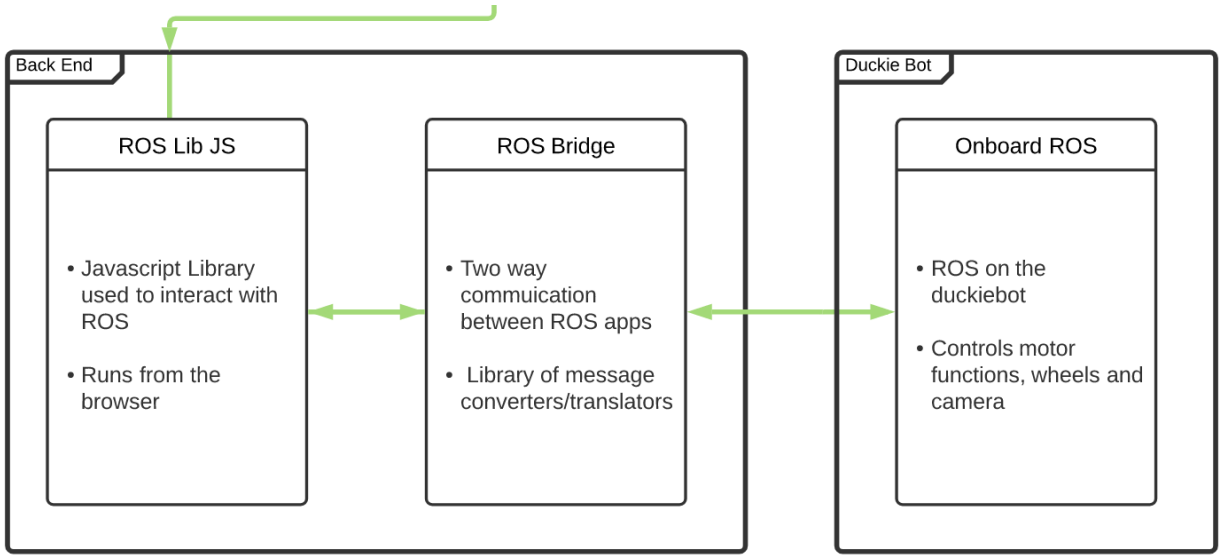

Figure 6: Diagram of roslibjs connecting to the duckiebot

4.4 Description of ROS running on Duckiebot (Duckietown software)

The duckiebot itself will be running basic control software that is capable of gathering footage from the camera and driving the motors that control the wheels of the robot. The robot itself is very simple with the only real form of control being how much power is going to each of the two wheels. This controls forward and backward motion as well as turning. The robot effectively steers like a tank with turning accomplished by sending more power to one wheel that the other.

The robot is entirely controlled by ROS. It is responsible for driving the motors and gathering data from the camera. As described above, the software on the robot will receive commands from the visual programming language and send the camera feed back to the visual programming language via rosbridge.

5. Implementation Plan

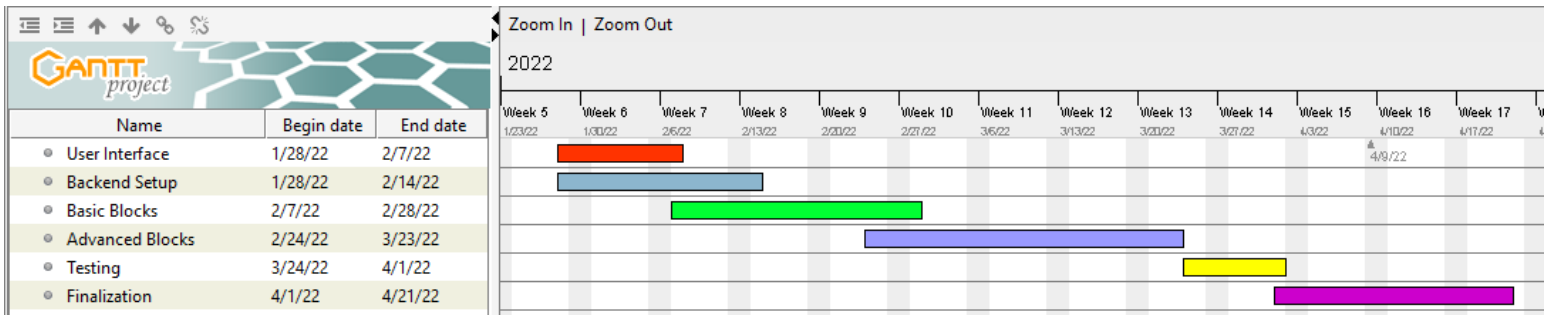

Figure 7: Code Duckies Spring 2022 Gantt Chart

5.1 Set up Blockly app

This step is represented by “User Interface” in our Gantt chart. We will create an extension to Blockly that allows users to manipulate visual blocks that represent blocks of code. The user interface will be as similar as possible to Scratch, which should be familiar to students who have used visual programming before. The team members working on this step are Daniel Rydberg and Jacob Heslop.

5.2 Connect Blockly app to actual bot

This step is represented by “Backend Setup” in our Gantt chart. We will create a backend which allows our Blockly extension to export code to the Duckiebot that the Duckiebot can execute. This step is being implemented simultaneously with the previous step. The team members working on this step are Anthony Simard, Ari Jaramillo and Chris Cisneros.

5.3 Communicate simple movement commands to bot using blocks

This step is represented by “Basic Blocks" in our Gantt chart. We will create custom Blockly blocks that represent blocks of code that the Duckiebot can execute. These will allow the user to give the Duckiebot simple commands such as move forward, move backward, turn right, turn left and stop without having to write any code themselves. The user will only need to use our visual user interface. The entire team will work together on this step.

5.4 Create blocks that can take in and process input from bot's camera

This step is represented by "Advanced Blocks" in our Gantt chart. These blocks will represent code that uses data from the Duckiebot's sensors, rather than naively obeying whatever movement commands the user gave them. This allows users to get a sense of how an autonomous vehicle operates autonomously. As blocks are completed, team members will branch off the main group to test them. Who tests which block will be decided at the time as needed.

5.5 Use camera input to follow lane using PID controller

This step is represented by "Testing" in our Gantt chart. We will use multiple of our own custom basic and advanced blocks to program the Duckiebot to drive within a lane autonomously. This will show that our custom blocks are sufficient to solve this problem when used together correctly. If not, they will need to be modified. Any lingering bugs will be fixed during our "Finalization" phase.

6. Conclusion

Through the development of a Blockly user interface will allow younger students to be able to experience the challenges and joy of developing instructions for self-driving cars. This project should decrease the difficulty of development of self-driving cars as well as interest students in areas of STEM (science, technology, engineering, and mathematics). Blockly is development friendly, allowing for creation of a library of blocks that can run more difficult code in an easy to use user interface. The Duckiebots that will be used in this project already have a well developed operating system that can be taken advantage of on the back end of Blockly. To get the commands from the web browser running Blockly, ROS can be used; and ROS already is well documented with a large library of commands that will benefit this project. By combining the already existing softwares together there is a way to simplify the handling of development for Duckiebots for a greater number of users.