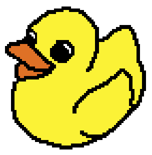
Requirements Specification

12/3/2021



Team Code Duckies

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

Daniel Rydberg, Chris Cisneros, Jacob Heslop

Sponsor: Dr. Truong X. Nghiem

Mentor: Vova Saruta

Accepted as baseline requirement for Visual Programming for Duckietown Autonomous Robots:

For client: ________________________ Date: __________

For the team: ______________________ Date: __________

Table of Content

1. Introduction **3**

2. Problem Statement **4**

3. Solution Vision **5**

4. Project Requirements **7**

4.1 Functional Requirements 7

4.2 Non-Functional Requirements 9

4.3 Environmental Requirements 10

5. Potential Risks **11**

5.1 Simulation Inaccuracy 11

5.2 Connection Loss 11

5.3 Damage to Vehicles 11

5.4 Damage to Environment 11

5.5 Injury to People 11

6. Project Plan **12**

7. Conclusion **13**

1. Introduction

The automotive industry is constantly trying to improve the safety features in their vehicles to reduce harm that comes to the passengers in the event of a car wreck. Even with these measures in place, the US Department of Transportation's National Highway Traffic Safety Administration said in 2020 there was a record high number of people who died in motor vehicle traffic crashes. Many deaths can be traced back to human error factors such as delayed reaction times, distracted driving, falling asleep at the wheel, or driving drunk. One solution that has been proposed is self-driving or autonomous vehicles, but such research is costly and takes a lot of professional work in order to produce cars that integrate well into society. Because of this, it is difficult for the average person, especially those of the younger generation, to experience what it might be like to create or develop such cars.

Dr. Truong X. Nghiem, an Assistant Professor for the School of Informatics, Computing and Cyber Systems at Northern Arizona University, fulfills many roles at NAU, including teaching university level classes centered around the development of self-driving cars. Classes about autonomous vehicles introduce students to basics of general robotics, computer vision, control and planning, but also require students to begin the classes with the ability to write and run programs in languages such as Python or C++. Unfortunately, even though these university students are putting money into taking these courses, many of them will not work in the field of self-driving cars. Dr. Nghiem wants to be able to expand the number of people who get the chance to experiment with this technology by simplifying the process of development and programming. His vision is to simplify the programming of cheaper, smaller robot vehicles by creating blocks of easy-to-read instructions for those who don't have programming experience to be able to use.

The system he intends to use is MIT Duckietown for their cheap Duckiebots. According to MIT, the original goal of the Duckietown project was "From a box of components, to a fleet of self-driving cars in just 3427 steps". The Duckiebots are designed to use the least hardware (eg. cheap cameras and small processors) in order to keep the cost at a minimum. Duckiebots run in a Duckietown environment. Duckietowns are built to be cheap and offer flexible driving experiences as roads and maps can vary. Duckiebots and the Duckietown software are built on an architecture of Linux, Dockers, Duckietown OS (or Duckietown libraries), ROS, and Python.

2. Problem Statement

The major issues Dr. Nghiem has been facing with getting students into the field of autonomous driving cars has been student interest, student accessibility and understanding of the concepts. He has found it incredibly difficult to teach students that are just starting out with coding how to interact with the Duckietown robots, due to the complex nature of the software environment that the bots use.

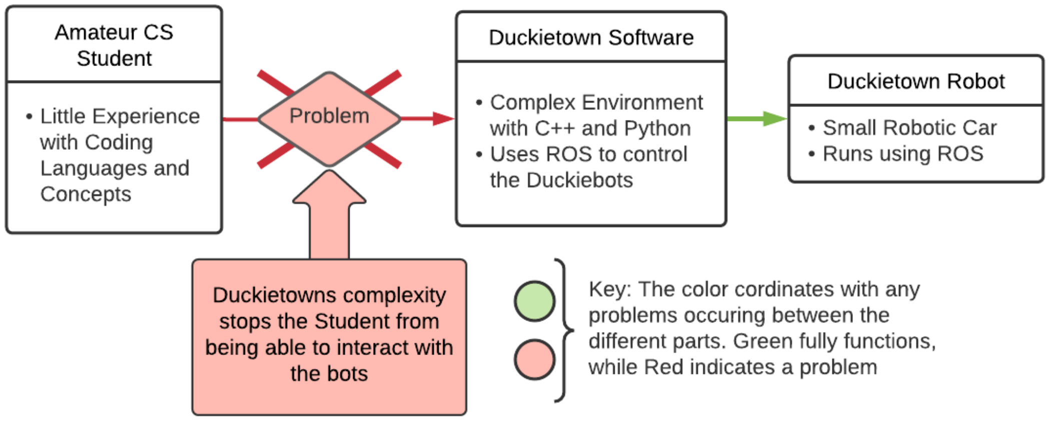

Amateur CS Student
• Little Experience with Coding Languages and Concepts
Problem
Duckietown Software
• Complex Environment with C++ and Python
• Uses ROS to control the Duckiebots
Duckietown Robot
• Small Robotic Car
• Runs using ROS
Duckietowns complexity stops the Student from being able to interact with the bots
Key: The color cordinates with any problems occuring between the different parts. Green fully functions, while Red indicates a problem

Figure 1. Diagram of Initial Client Workflow

Figure 1 above helps to visualize how the students interact with the Duckiebots currently, and where the problem with the current workflow lies. To use the Duckietown software, which is used to interact with the Duckiebots, the user needs to have a proficient level of knowledge in coding languages and concepts. Without that knowledge it becomes difficult to interact with the Duckiebots, which leads to a high barrier to entry into the field of autonomous driving cars. In order to remove this barrier, we need:

- A Visual Programming Language so students with little experience in coding can comprehend and interact with the workspace environment.
- An easy-to-use UI that students might be familiar with, allowing them to navigate the environment efficiently.
- The abstraction of the coding process of the Duckiebots into easy-to-understand connectable coding blocks that the students can use to make programs for the Duckiebots to follow.
- The chosen Visual Programming Language also needs to be able to link to the Duckietown software and create code in Python and C++ that the Duckiebots can understand and follow.

If all the bullet points above are resolved, it will significantly lessen the barrier to entry of the autonomous driving cars field. This allows amateur students to be exposed to and become interested in the field at a much earlier age with far less coding specific experience.

3. Solution Vision

Our team proposes to build a custom block library in the Visual Programming Language Blockly. This would allow students to combine our custom-made blocks together to create programs that the Duckiebots can then read and follow, without requiring students to ever have to actually write any code in Python or C++. Our custom blocks would accept parameters given by the students that would modify the performance of the Duckiebots. For example, things like a distance to travel or turn speed.

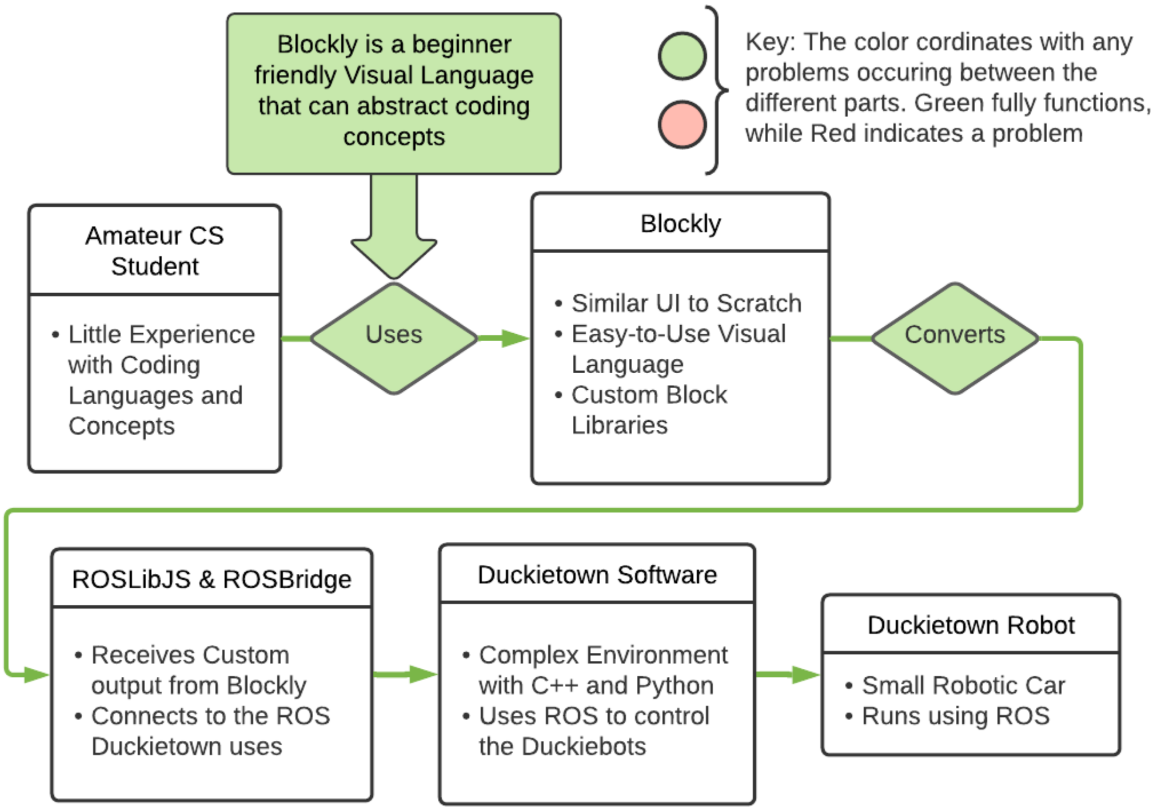

Blockly is a beginner friendly Visual Language that can abstract coding concepts
Key: The color cordinates with any problems occuring between the different parts. Green fully functions, while Red indicates a problem
Amateur CS Student
• Little Experience with Coding Languages and Concepts
Uses
Blockly
• Similar UI to Scratch
• Easy-to-Use Visual Language
• Custom Block Libraries
Converts
ROSLibJS & ROSBridge
• Receives Custom output from Blockly
• Connects to the ROS Duckietown uses
Duckietown Software
• Complex Environment with C++ and Python
• Uses ROS to control the Duckiebots
Duckietown Robot
• Small Robotic Car
• Runs using ROS

Figure 2. Diagram of Client Workflow After Solution Implementation

Figure 2 shows the technologies we plan to employ to create a connection between Blockly and the Duckietown software. The key points on why we chose this solution over our other options were:

- Blockly is based on and has a similar UI to Scratch, a beginner visual language used to introduce students to coding
- Blockly's block libraries allow for the complex code needed to run the Duckiebots to be abstracted into templates (the blocks) that students can easily use and understand

- Blockly’s flexible block style libraries allow for us to customize the coding language of the blocks and the output of the blocks
    - This lets us output to some ROS-related software that can then be used to connect to the ROS being used by the Duckietown software and bots

Our solution would allow our sponsor to present amateur/beginner level CS students with easy-to-use and easy-to-understand software, despite the complex nature of the initial software. This allows students an avenue to learn about and interact with autonomous driving cars at a much lower level of CS knowledge, inspiring interest in the field.

Through our research we discovered several possible alternative solutions. One of which was to build our own visual language from the ground up and use that as our abstraction tool. We decided against this due to the team having no previous experience developing a completely new visual language and the time needed to undergo the new languages construction. Another option was to use Scratch, however the flexibility of Scratch’s custom libraries is significantly less than that of Blockly. Scratch is also confined to blocks in Javascript.

Additionally, should our client, Dr. Nghiem, decide to make our finished software public, other courses that are being taught using Duckietown and the Duckiebots should be able to make full use of our software as well. This would allow even more students to get exposed to the field of autonomous driving cars at a young age and lead to more interest in the field overall.

4. Project Requirements

4.1 Functional Requirements

4.1.1 WebApp

Our tool will take the form of a webapp. This webapp will be locally hosted, so there will be no communication with any remote systems aside from the Duckiebot. A webapp is necessary because Blockly runs in the browser.

4.1.1.1 GUI

The webapp will require a GUI frontend for the user to interact with. This frontend will cover all parts of the tool the user will actually interact with. It will allow the user to assemble a visual program that will run on the Duckiebot without the user needing to understand how their code got to the Duckiebot or how it works on the Duckiebot.

4.1.1.1.1 Set of Blocks

A set of visual blocks will be created for the user to assemble programs out of. These blocks will cover all basic programming functionality as well as control over the Duckiebot. They will have basic and intuitive functions that should be understood by someone who has no prior programming experience, and they will be parameterizable enough for the user to control the behavior of the Duckiebot (exactly what this means will be described for each type of block individually). They will also visually fit together in an intuitive way.

4.1.1.1.1.1 Basic Control Blocks

The most basic form of control will be ordering the Duckiebot to move set amounts at a set speed. There will be a block that moves the Duckiebot forward a set distance at a set speed. There will be a similar block that reverses the Duckiebot. There will also be a block that turns the Duckiebot left or right to a set number of degrees at a set rotational speed. These speeds will be entered as a percentage of the Duckiebot’s maximum speed. An example block may tell the Duckiebot to move forward one meter at twenty-five percent speed. The block itself is “move forward” while the distance and speed are parameters in the block that are set by the user. The movement distances and angles specified will be achieved within the smallest margin of error the Duckiebot is capable of with the understanding that exact distances are infeasible in real life.

4.1.1.1.1.2 PID Controller block

A PID controller block will be needed to allow for more complex control over the Duckiebot. This block will allow the Duckiebot to follow a lane by sending commands to the Duckiebot's wheels that keep it centered in the lane. The block will be parameterizable with the weights for each of the proportional, integral, and derivative components set by the user.

4.1.1.1.1.3 Camera Control Block

For the PID controller to work, it will need to know the current position of the Duckiebot in the lane (how far it is from the lane edges). There will be a block that takes in the data from the Duckiebot's camera, identifies the lane edges, and estimates the current distance of the Duckiebot from said lane edges. The user will be able to use this block to feed distance data into the PID controller block.

4.1.1.1.1.4 Blockly Standard Library

The Blockly standard library of blocks contains many math and control flow operations. These will need to be included to allow for proper program creation. For instance, we want the camera block to constantly get new data from the camera and feed it to the PID controller in a loop. This requires a loop block in the program, which Blockly already provides.

4.1.1.1.2 Program Assembly Area

The GUI will need to include an area in which to assemble a program out of blocks. Ideally this area will allow for different levels of zoom, and it will grow to arbitrary sizes to support arbitrarily large programs. At minimum, The assembly area will take up 2/3rds of the width and height of a 1920x1080p display. It will provide enough space for the user to assemble a program that allows the Duckiebot to follow a lane and have the entire program visible on screen at once.

4.1.1.1.3 Start and Stop Buttons

The user will be able to instruct the Duckiebot to stop and start executing their program. Assuming ideal network conditions, it will take no more than five seconds for the robot to respond to a command to start a new program or to halt execution and stop moving.

4.1.2 Backend

The GUI will have a backend to get the user's visual program actually running on the Duckiebot. This backend will execute the code that connects to the Duckiebot, and it will send the instructions to the Duckiebot. The user will not need to interact with the backend at all.

4.1.2.1 Conversion to ROS Commands

The backend will convert the user's visual program into ROS commands that can control the Duckiebot. This will be done without the user needing to understand the underlying commands running on the backend. The user will only need to understand the high level logic that is controlling the Duckiebot. They will not need to worry about the semantics of the code execution at all. This conversion to ROS commands will happen using the JavaScript library roslibjs.

4.1.2.2 Real Time Connection to Duckiebot

We are targeting a real time connection to the Duckiebot. The user will be able to maintain an active connection from their computer to the Duckiebot that allows them to rapidly change what program the robot is executing in real time. This also makes it possible to rapidly stop the robot in real time. This connection will allow for the functionality described in 4.1.1.1.3. Rosbridge will be used to communicate ros commands from the webapp to the Duckiebot in real time.

4.2 Non-Functional Requirements

4.2.1 Minimal knowledge required

The visual programming tool will not require a certain amount of previous programming experience. Average people that are between the ages of 11 and 18 will be able to use the tool to build a program that moves the robot forward in about two hours or less.

4.2.2 Accurate deployment of code

The tool will accurately deploy the user's code to the duckiebot. For example, if the user wants the Duckiebot to move forward for a certain amount of time then the visual programming tool will send that command to the Duckiebot and it will move forward for the amount of time within a window of error of +/- 1 second.

4.2.3 Simplicity of UI

The user interface will be simple and have explicit folders for different types of blocks such as logic, math, loops, and movement. A user should be able to find a block for an if-else statement within two minutes.

4.2.4 User interface similar to Scratch

The user interface will look similar to Scratch so that it is familiar to users who have used/seen Scratch before so the learning curve will be much lower.

4.3 Environmental Requirements

4.3.1 Uses Duckietown

The visual programming tool will use Duckietown code in the backend for its custom blocks.

4.3.2 ROS Noetic

In the back-end, the tool will use a ROS node to establish a connection between the computer and a Duckiebot.

4.3.3 Ubuntu 20.04 Focal OS

The optimal operating system to run the visual programming tool will be Linux, Ubuntu 20.04 Focal. This is suggested because the Duckietown simulator[1] recommends using Ubuntu 20.04 as there might be compatibility issues if another operating system is used. The ability to do a simulation of a Duckietown program is just as important as programming the actual Duckiebot due to the possibility of users not having access to one.

[1] Chevalier-Boisvert, M., Golemo, F., Cao, Y., Mehta, B., & Paull, L. (2018). Duckietown Environments for OpenAI Gym. *GitHub repository*. Received from https://github.com/duckietown/gym-duckietown

5. Potential Risks

5.1 Simulation Inaccuracy

Duckietown includes a simulator that can be used to test code before running it on a real vehicle. However, this simulator is not always a perfect match for reality, as it cannot calculate every aspect of the physical world. One risk is that a user could create code that works perfectly well in the simulator, but not on the real vehicle. The effects of this could range from simply not moving, to crashing and causing damage to the vehicle, the test area, or nearby people.

5.2 Connection Loss

Duckietown connects to the autonomous vehicle wirelessly. If this connection was lost, it would render the user unable to control the vehicle, potentially causing a crash. This problem could be remedied by forcing the vehicle to stop in the event that the connection is lost.

5.3 Damage to Vehicles

If users are permitted to write any code they wish, they could accidentally or even intentionally damage the vehicle. This could be done by crashing the vehicle into some object in the environment, or crashing the computer that drives the vehicle. As autonomous vehicles can be quite expensive, and the hardware within them can be sensitive, any damage would be of great concern to the owners. This could be fixed by programming the vehicle to recognize other vehicles, and avoid crashing into them, regardless of what the user tells it to do.

5.4 Damage to Environment

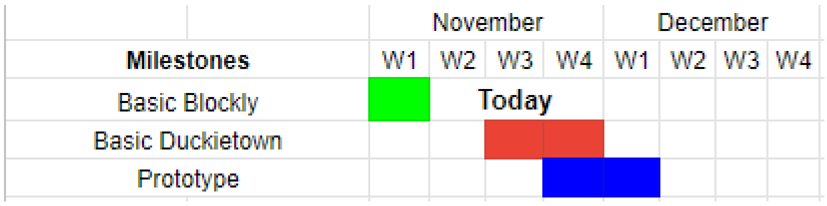
A vehicle can damage the object it crashes into as well. Autonomous vehicles are often tested in elaborate courses with intricate details such as traffic signs, decorative buildings, and model pedestrians. A crash could do a lot of damage to such a setup. Even more damage is possible if the vehicle goes off the course, potentially crashing into more expensive objects, such as computers. This could be remedied by shutting off the vehicle if it leaves the course.

5.5 Injury to People

A vehicle could also crash into a human being. While the vehicles we will be working with are quite small, one could still run over someone's foot or trip them. That alone can cause serious harm. If our software is ever applied to larger vehicles, the potential damage only increases, as does the need for a solution. This could be solved similarly to 5.3, by programming the vehicle to recognize pedestrians.

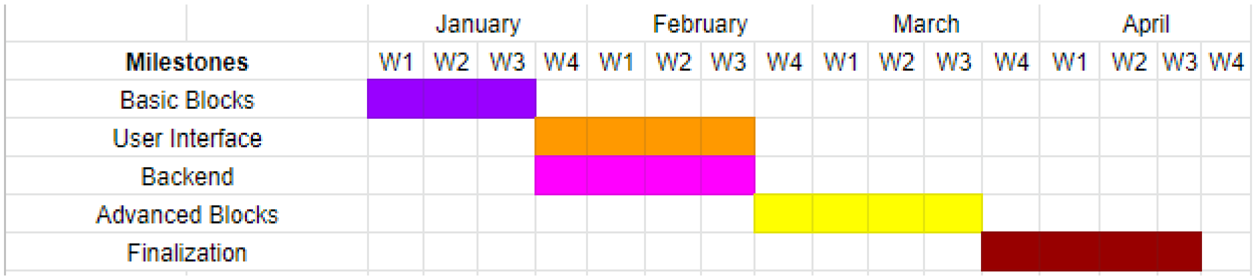
6. Project Plan

Fall 2021



	November				December			
Milestones	W1	W2	W3	W4	W1	W2	W3	W4
Basic Blockly	🟩		**Today**					
Basic Duckietown			🟥	🟥				
Prototype				🟦	🟦			

Spring 2022



	January				February				March				April			
Milestones	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
Basic Blocks	🟪	🟪	🟪													
User Interface				🟧	🟧	🟧	🟧									
Backend				🟪	🟪	🟪	🟪									
Advanced Blocks								🟨	🟨	🟨	🟨					
Finalization												🟥	🟥	🟥	🟥	

6.1 Basic Blockly 11/5

Learn how to program using Blocky, and demonstrate a basic program using Blockly.

6.2 Basic Duckietown 11/28

Learn how to program using Duckietown to control a vehicle in the simulator.

6.3 Blocky + Duckietown Prototype 12/3

Use Blockly to represent blocks of Duckietown code which control a vehicle in the simulator.

6.4 Basic Blocks 1/21

Create custom Blockly blocks that represent the blocks of Duckietown code that students would need in a visual programming for autonomous vehicles course.

6.5 User Interface 2/21

Create an intuitive visual user interface for our Blockly extension. This should include and start/stop button for the code running on the vehicle.

6.6 Backend 2/21

Write code to connect the webapp to the Duckiebot and convert blocks into ROS commands.

6.7 Advanced Blocks 3/21

Create custom Blockly blocks that represent the blocks of Duckietown code used for more advanced features such as PID and camera control.

6.8 Finalization 4/21

Throughout the development process, it is likely that we will encounter bugs from previous milestones we thought complete. Before the project is finished, it will be necessary to thoroughly test our work to find and fix these bugs.

7. Conclusion

The development of self-driving cars is a narrow field that is hard for people to get into or experience. Our client, Dr. Nghiem, wants to be able to expand the availability of experience to younger students but has found there is a gap in experience in students as well as the difficulty of handling Duckiebots he is using. Our group is tasked with creating a program that interacts well with the DuckyTown system that is also easy for inexperienced students to use. Our solution involves creating a locally run webpage that acts as an interactive GUI where visually basic commands can be arranged in a “programming area” which can then be sent with minimal delay to control the Duckiebots. We will be using the Blockly standard library for simplicity of user interaction and its ability to interact with current ROS and DuckyTown libraries.