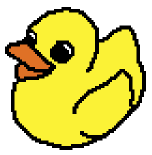
Technical Feasibility

11/05/2021



Team Code Duckies

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

Daniel Rydberg, Chris Cisneros, Jacob Heslop

Sponsor: Dr. Truong X. Nghiem

Mentor: Vova Saruta

Table of Content

1. Overview

In June 2021, the US Department of Transportation's National Highway Traffic Safety Administration released an estimate[1] of 38,680 people who had died in 2020 in motor vehicle traffic crashes. This does not even include non-fatal accidents! Many motor vehicle accidents are caused by factors such as delayed reaction times, distracted driving, falling asleep at the wheel, or driving drunk. One of the most prevalent branches in automotive science fiction, and a challenge that many automotive developers are trying to tackle is the creation of self-driving cars that integrate well into society; ideally removing these human factors. Unfortunately, the creation of such vehicles has a high cost of development and production including the time and knowledge needed to program an artificial intelligence to drive the car, and is a difficult field to get into!

Our client, Dr. Truong X. Nghiem, works at Northern Arizona University as an Assistant Professor at the School of Informatics, Computing and Cyber Systems. He currently fulfills many roles in that title, including teaching classes centered around the development of autonomous vehicles to university students. These classes include basics of general robotics, computer vision, control and planning, and requires the ability to program in languages such as Python or C++. Although helpful, these classes still pose problems since a vast majority of his students (who are paying for classes) will probably never work in the field of self-driving cars (again limiting the number of those creating autonomous vehicles). The high technical skills needed for this class would be difficult for most middle or high school students to be able to pick up as well.

Dr. Nghiem wants to be able to expand his reach to younger students in order to get more of the younger generation interested in the field of autonomous vehicles. His proposed solution is to build a system that removes the technical skills needed for kids to program using Python, C++, or running on the command line by instead encapsulating the running code into basic commands (seen as blocks) that can be dragged and dropped into a run environment. These commands can then be on small, cheap autonomous robots such as MIT's currently existing Duckietown mobile robots. Examples of commands that could be run might be: "move forward at X speed" (general robots), "check for pedestrians" (computer vision) or "determine location and move towards destination" (control and planning).

[1] U.S. N.H.T.S.A. (2021, June 3). 2020 fatality data show increased traffic fatalities during pandemic. NHTSA.

2. Technological Challenges

In the following section we describe the various challenges any solution to this problem will be required to overcome. Ideally we will leverage a pre-existing visual programming language to reduce the development time required to build a new one from the ground up. This will also give us a pre-existing user experience and set of user documentation to ease the user in. Whether we use a pre-existing language or build our own, it will have to meet the following requirements.

2.1 Ease of Use

Since we are targeting a younger audience that may have little to no prior programming experience, any solution we implement must be simple and intuitive for them to use. That is the entire point of this project afterall. To that end, we must ensure that our tool is easy enough to use so that the interaction with and understanding of the robot's functioning can take precedence over learning the semantics of the tool.

2.2 Ability to Interface with the Duckiebot

The Duckiebots run their code in Docker containers and make use of a framework called Robotic Operating System or ROS to facilitate communication between software and hardware. These are technology decisions that were made for us. We have to use ROS, Docker, and the Duckiebots, so our language must be able to interact with these tools. There are multiple issues to consider when thinking about interfacing with the bots via these tools.

2.1.1 Back-end Language

Visual Programming Languages execute on the backend using more traditional programming languages (e.g. JavaScript, Python, etc.). The backend execution of the visual programming language will need to be able to control the bots. The bots are typically controlled using Python, and all of the documentation related specifically to controlling the Duckiebots uses Python. We do not necessarily need our backend language to be Python though due to how ROS works. ROS works by constructing a graph of nodes that intercommunicate. The roscore command is used to create a ROS session. Nodes can then be added to the ROS session in several ways. Rosrun specifies a single script that creates a single node to add to the session. Roslaunch takes an XML file that specifies a list of scripts to use to create multiple nodes. The scripts themselves tell ROS how to create a given node and how that node should connect to the others. Each node controls a different part of the system. For instance, one node can be taking input from the camera, another node can be sending commands to the motors, etc. It is possible to create ROS nodes in many different languages; these nodes communicate in a language agnostic manner. This means that while Python is ideal, any language that ROS nodes can be written in should work. As an example of how this might work, we could write the ROS nodes that interact directly with the hardware in Python ourselves. The visual programming language can then create nodes in its backend language that receive data from the camera node, process that data, then send some commands to the motor control node, making the bot move in a particular way.

2.1.2 Network Communication

We will need to be able to transfer the commands from the visual programming language to the Duckiebot in some way. This needs to be done with minimal effort from the user. Ideally they will be able to press a single button to start executing their code on the robot and a single other button to halt the robot. ROS solves the backend portion of this problem for us in the form of rosbridge and roslibjs. Rosbridge provides a JSON API for non-ROS programs (our tool) to communicate with ROS. Roslibjs allows for communication with rosbridge from a web browser. This means we only need to worry about the front-end that will interact with these tools.

2.3 Documentation

If we use an already existing visual programming language, it must have solid developer documentation describing how to extend it for our purposes. If we cannot figure out how to start adding features to an existing tool to make it do what we want, it is not the right tool for the job. This is largely separate from the user documentation. Unless we can find a tool that natively supports what we are trying to do (unlikely), we will need to know how to modify and extend the tool, not just use it.

3. Technology Analysis

3.1 Intro the Issue

Duckietown is an ecosystem for robotics and AI that allows college students to learn and for general research. Due to its nature of using Python and C++ to interact with the Duckiebots, car robots, it is difficult for anyone without prior programming experience to use Duckietown effectively. The goal is to develop a visual programming tool that allows middle to high school aged students to interact with the Duckiebots with little to no programming experience.

3.2 Desired Characteristics

The ideal visual programming tool will need several important features. First off, the overall design and user interface of the tool will need to be simple and intuitive for a younger audience to use. We don’t want our target audience to be worried about how to use the tool and to be focused more on how to interact with the Duckiebots. Second, it will need to be able to interface with the Duckiebots when the play button is pressed. The tool will need to be able to at least execute Python on the backside since that is one of the main languages that Duckietown uses. It should be able to transfer blocks of code from the visual programming tool to the Duckiebots using ROS nodes. Lastly, our ideal visual programming tool will need clear and solid developer documentation. This will allow the team to be able to easily modify and extend an existing tool to develop it to work for our purposes.

3.3 Alternatives

3.3.1 Scratch

Scratch is a well known visual programming language used by children everywhere to start learning the basics of programming without getting deep into the semantics. It is made by MIT, and it is very well supported and documented. It is probably the premier visual programming tool that currently exists.

3.3.2 Snap

Snap is an offshoot from Scratch that is made by UC Berkeley. It is designed to be a more rigorous version of Scratch that is better capable of representing some more complicated programming concepts.

3.3.3 Ryven

Ryven is a smaller and more obscure language that is designed to be a visual representation of a Python program. Where Scratch and by extension Snap look like connected LEGO blocks, Ryven looks more like a flowchart.

3.3.4 Blockly

Blockly, developed by Google and MIT, adds an editor to your application that allows representation of code as interlocking blocks. The output will be correct code in the chosen programming language of the user’s choice. Its overall goal is to allow programmers to create a simple visual programming editor that can interface with an existing application.

3.3.5 FlowCanvas

FlowCanvas is a visual programming tool that can be used with the game engine Unity. Like other visual programming tools, FlowCanvas represents blocks of C# or JavaScript code as visual “nodes” that can be linked together in order to program a video game. While FlowCanvas is not directly applicable to this project, it can serve as inspiration if we need to create a visual programming tool of our own, rather than using one of our listed alternatives.

3.4 Analysis

3.4.1 Scratch Analysis

Ease of Use:

Scratch is probably the most easily accessible option we have. It was designed for children, and it is the language children are most likely to be familiar with. It has a pleasant and inviting user-interface, and its user documentation is thorough.

Ability to Interface with the Duckiebot:

Scratch was not designed for this sort of application. Some people, namely LEGO, have gotten Scratch to control physical robots, but they were able to build the robots with the ability to interface with Scratch as an end goal. We did not design the Duckiebots, so we must work within their constraints.

Back-end Language:

Scratch executes JavaScript on the backend which is potentially problematic because it is unclear exactly what challenges we will face getting the JavaScript Scratch backend to communicate with the Duckiebots. There is a library that allows for creation of ROS nodes in JavaScript though, so we should be able to use the JavaScript Scratch back-end to create ROS nodes that communicate with the Python nodes controlling the hardware in the manner described in section 2.1.1.

Network Communication:

Scratch has a start button that starts program execution and a pause button that halts execution in place while maintaining state. If the project is edited while it is running, the modifications show up in real time in the Scratch project. We may be able to use the start button to create our ROS node(s) and get the project started while using the pause button to send a halt command to the robot. Since Scratch is open source, we should be able to change the function of the start and pause buttons to change states in ROS in ways that we want. . We may also be able to make blocks that take care of this functionality. Whether we will be able to update the ROS project the robot is executing in real time as the Scratch project changes is less clear, but we should at least be able to get the code into the ROS project to communicate with the robot.

Documentation:

Since Scratch is a well established and maintained language, it is unsurprising that it supports reasonably comprehensive documentation. We have determined the necessary steps to start adding our own blocks to Scratch. The question remains of how to make those blocks do what we want them to do.

Summary:

An overall summary of our ideas on Scratch is illustrated in figure 1. The user experience and documentation are excellent, but there are some questions about how well it will be able to interact with the Duckiebot.

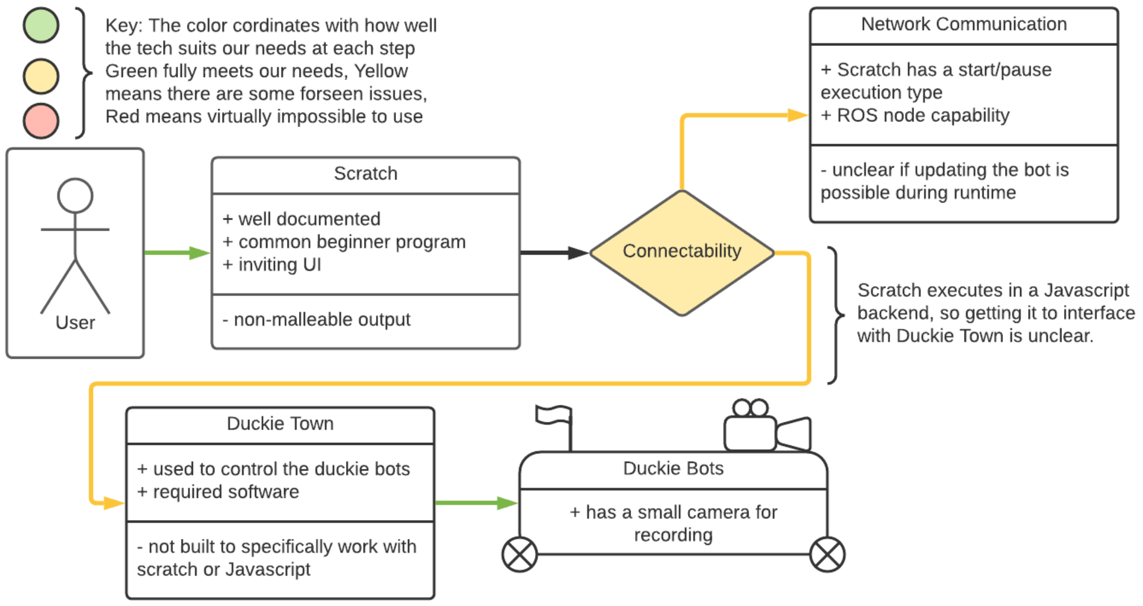

Key: The color cordinates with how well the tech suits our needs at each step Green fully meets our needs, Yellow means there are some forseen issues, Red means virtually impossible to use
User
Scratch
+ well documented
+ common beginner program
+ inviting UI
- non-malleable output
Connectability
Network Communication
+ Scratch has a start/pause execution type
+ ROS node capability
- unclear if updating the bot is possible during runtime
Scratch executes in a Javascript backend, so getting it to interface with Duckie Town is unclear.
Duckie Town
+ used to control the duckie bots
+ required software
- not built to specifically work with scratch or Javascript
Duckie Bots
+ has a small camera for recording

Figure 1. Diagram of Technological Feasibility: Scratch

3.4.2 Snap Analysis

Ease of Use:

Snap is not as intuitive or kid friendly as Scratch. It is explicitly designed to be a bit deeper than Scratch introducing children to programming concepts more directly.

Ability to Interface with the Duckiebot:

Since Snap is based directly on Scratch, much of its inner workings are similar to or even identical to Scratch. As such, there will be similar benefits and challenges to using Snap as with Scratch.

Back-end Language:

Snap executes JavaScript on the back end the same as Scratch.

Network Communication:

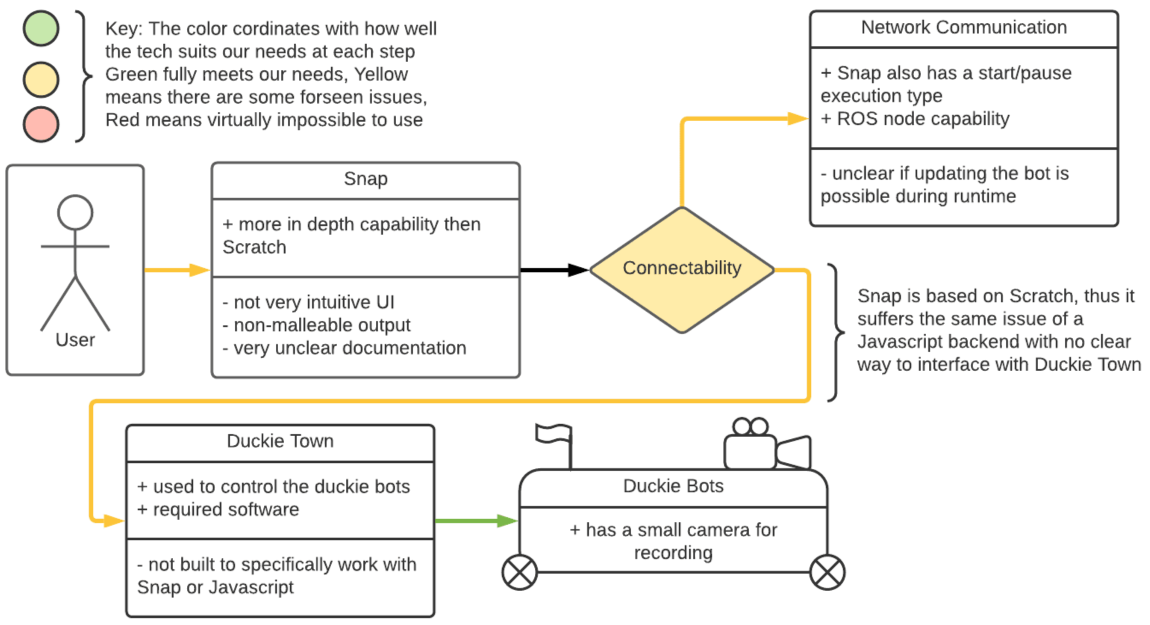
Snap is for all intents and purposes identical to Scratch in this respect as well.

Documentation:

One of the primary failures of Snap for this project is its lack of adequate documentation. It is less well supported, and it is less obvious how one would go about creating custom Snap blocks than custom Scratch blocks. This makes it a very unlikely candidate for our purposes as it is just a less malleable and less intuitive version of Scratch.

Summary:

As figure 2 suggests, Snap is probably not a good choice for this project. It fails to do anything better than Scratch for our purposes, and it is substantially worse in usability and documentation.



Key: The color cordinates with how well the tech suits our needs at each step Green fully meets our needs, Yellow means there are some forseen issues, Red means virtually impossible to use
User
Snap
+ more in depth capability then Scratch
- not very intuitive UI
- non-malleable output
- very unclear documentation
Connectability
Network Communication
+ Snap also has a start/pause execution type
+ ROS node capability
- unclear if updating the bot is possible during runtime
Snap is based on Scratch, thus it suffers the same issue of a Javascript backend with no clear way to interface with Duckie Town
Duckie Town
+ used to control the duckie bots
+ required software
- not built to specifically work with Snap or Javascript
Duckie Bots
+ has a small camera for recording

Figure 2. Diagram of Technological Feasibility: Snap

3.4.3 Ryven Analysis

Ease of Use:

Ryven is not as intuitive and easy to grasp as the others without understanding the semantics of actually writing code. It has some unusual design decisions, particularly surrounding the management of variables that make it harder to learn to use Ryven than some of the alternatives.

Ability to Interface with the Duckiebot:

Ryven was designed for creating visual dataflows of pure Python programs. It was not designed to interface with external devices like our Duckiebots.

Back-end Language:

Ryven is a visual representation of a Python program. It executes Python code on the back-end. Based on some inspection of the source code, it looks like it executes the Python code directly in the Python interpreter without writing it to a file. This could be potentially problematic for our purposes because it does not give us a file to point roslaunch or rosrun at.

Network Communication:

Ryven allows for the dynamic creation of buttons that execute the script attached to them when clicked. It would be possible to create a button that attached our node to the ROS project running on the bot, assuming we were able to create a node in Ryven, which looks unlikely.

Documentation:

Ryven is a smaller project than the others, as it is made by a smaller team. It is not as well supported nor is it as well documented. There is reasonable documentation explaining how to use it, but there is little to no documentation explaining how it works and how to expand upon it meaningfully. In order to use a tool for our project, we need to understand how the tool actually executes the visual program on the back-end.

Summary:

After analyzing Ryven, it can be concluded that the tool would not be ideal to use. Looking at figure 3, the majority of the arrows are red and there are a lot of flaws. Ryven is not user friendly and the ability to connect to Duckiebots is unclear.

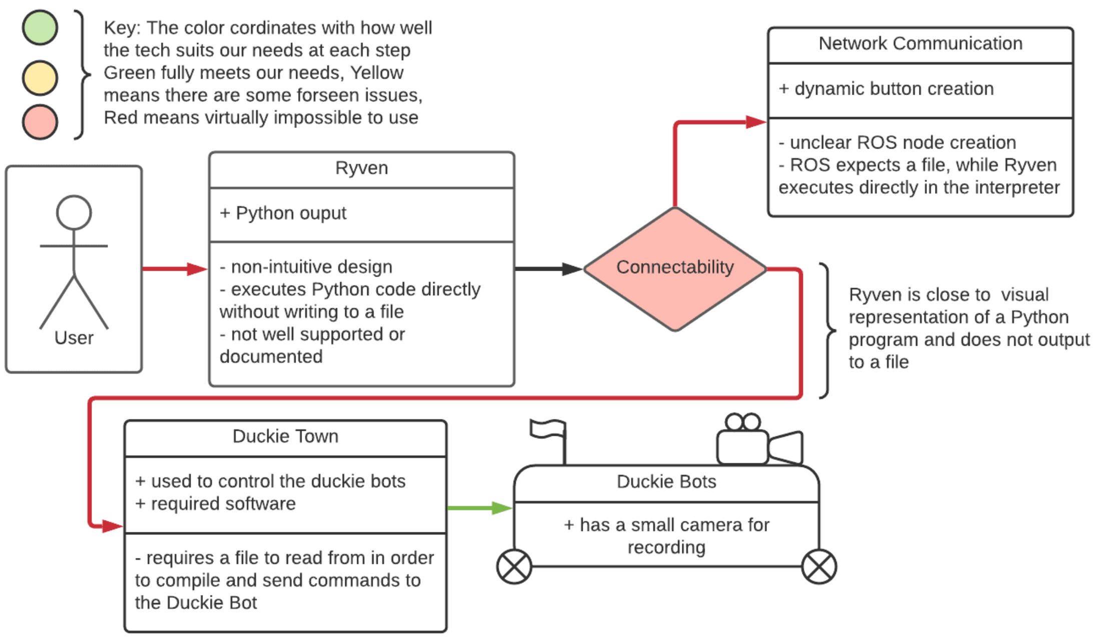

Key: The color cordinates with how well the tech suits our needs at each step Green fully meets our needs, Yellow means there are some forseen issues, Red means virtually impossible to use
User
Ryven
+ Python ouput
- non-intuitive design
- executes Python code directly without writing to a file
- not well supported or documented
Connectability
Network Communication
+ dynamic button creation
- unclear ROS node creation
- ROS expects a file, while Ryven executes directly in the interpreter
Ryven is close to visual representation of a Python program and does not output to a file
Duckie Town
+ used to control the duckie bots
+ required software
- requires a file to read from in order to compile and send commands to the Duckie Bot
Duckie Bots
+ has a small camera for recording

Figure 3. Diagram of Technological Feasibility: Ryven

3.4.4 Blockly Analysis

Ease of Use:

Blockly is another tool that is a very accessible option. The visual design is similar to Scratch as one of its designers is MIT, who also developed Scratch. Children will be familiar with the layout of Blockly as it looks like Scratch. The simple yet appealing interface, thorough documentation, and multiple tutorials makes this tool seem easy to use.

Ability to Interface with the Duckiebot:

Blockly was designed to be completely generic and applicable to a wide variety of problems. In fact, there is solid precedent for using Blockly to control bots described in the paper[2] by Karaca & Yayan 2020. That paper uses Blockly to interface with a robot that uses ROS just like our Duckiebots.

Back-end Language:

Blockly can export blocks into different programming languages such as Javascript, Python, PHP, and others. This ability to choose the backend language is exactly what we need as Duckietown blocks of codes could be generated and simply sent to the robots.

[2] Karaca, M., & Yayan, U. (2020). ROS Based Visual Programming Tool for Mobile Robot Education and Applications (Preprint).

The visual programming tool created using Blockly would be able to do the process as described in 2.1.1.

Network Communication:

An important topic to analyze is if Blockly can establish a connection with the Duckiebot as described in 2.1.2. With the flexibility provided by Blockly, it would be possible and simple to establish a connection between the visual programming tool and Duckiebots. It could use Python code to create the ROS nodes needed for network communication. This implementation would theoretically allow the user to push the start button and send the blocks of code to the Duckiebot.

Documentation:

Blockly is a large project that has helped develop numerous visual programming languages or websites such as Code.org, OzoBlockly, Code Spells, and more. The documentation behind the tool is comprehensive and even has tutorials that explain how to add custom blocks and implement other features. There is no question that Blockly has some of the best documentation out of all the tools so far.

Summary:

Overall, it seems that Blockly is the ideal solution for our project. Figure 4 has all important aspects as green. That means Blockly is user friendly, can connect to the DuckieBots, and should have minimal difficulty to understand how to use it. Using Blockly would allow us to create custom blocks which would be similar to creating our own visual programming tool but with most of the groundwork laid down

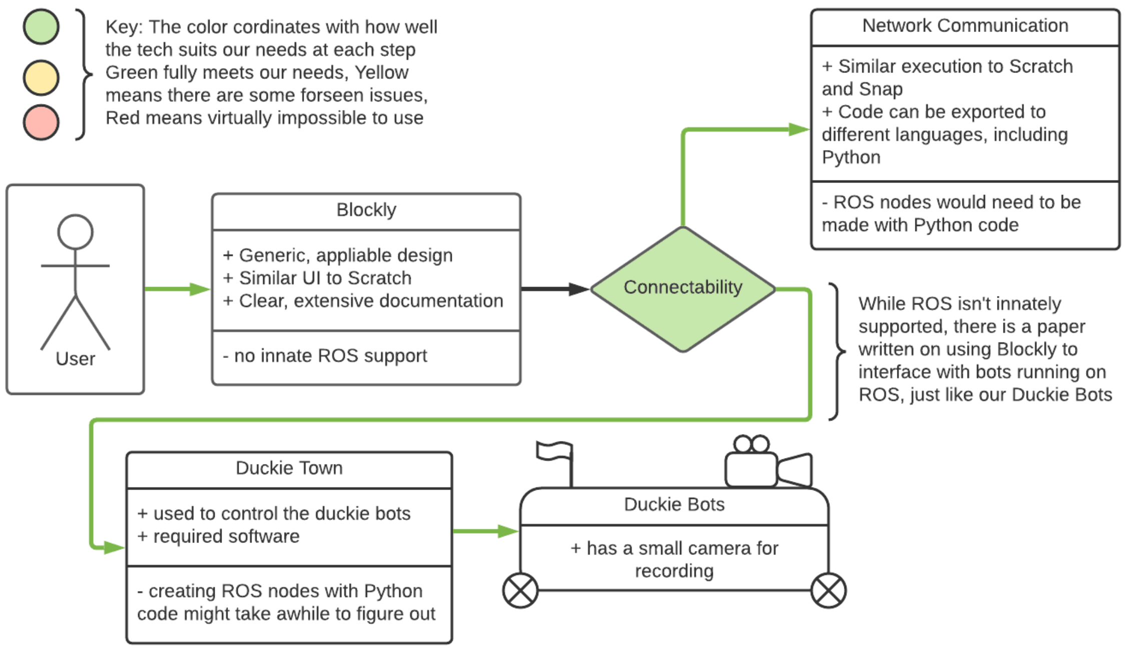

Key: The color cordinates with how well the tech suits our needs at each step Green fully meets our needs, Yellow means there are some forseen issues, Red means virtually impossible to use
User
Blockly
+ Generic, appliable design
+ Similar UI to Scratch
+ Clear, extensive documentation
- no innate ROS support
Connectability
Network Communication
+ Similar execution to Scratch and Snap
+ Code can be exported to different languages, including Python
- ROS nodes would need to be made with Python code
While ROS isn't innately supported, there is a paper written on using Blockly to interface with bots running on ROS, just like our Duckie Bots
Duckie Town
+ used to control the duckie bots
+ required software
- creating ROS nodes with Python code might take awhile to figure out
Duckie Bots
+ has a small camera for recording

Figure 4. Diagram of Technological Feasibility: Blockly

3.4.5 FlowCanvas Analysis

Ease of use:

FlowCanvas is highly intuitive. It is laid out similar to a circuit board. One can watch in real-time as variables “flow” like electricity between nodes, allowing one to see exactly what the program is doing at all times.

Ability to interface with Duckiebot:

FlowCanvas is specifically for the Unity engine, and unfortunately cannot be used with Duckiebots.

Back-end Language:

FlowCanvas uses C# and JavaScript, the two programming languages that Unity scripts can be written in.

Network Communication:

FlowCanvas cannot directly access networks, and instead utilizes Unity’s built-in networking features.

Documentation:

FlowCanvas provides about 28 pages of documentation, listing all of its features and how they are used. This will be useful in understanding how visual programming languages work and what features are expected from one. It was last updated on September 20th, 2021.

Summary:

Looking at FlowCanvas as a whole, it would not be an ideal solution for our project. Figure 5 below highlights all the flaws in the tool. While FlowCanvas is user friendly, it will not be able to connect to the Duckiebots which is extremely important for our purposes.

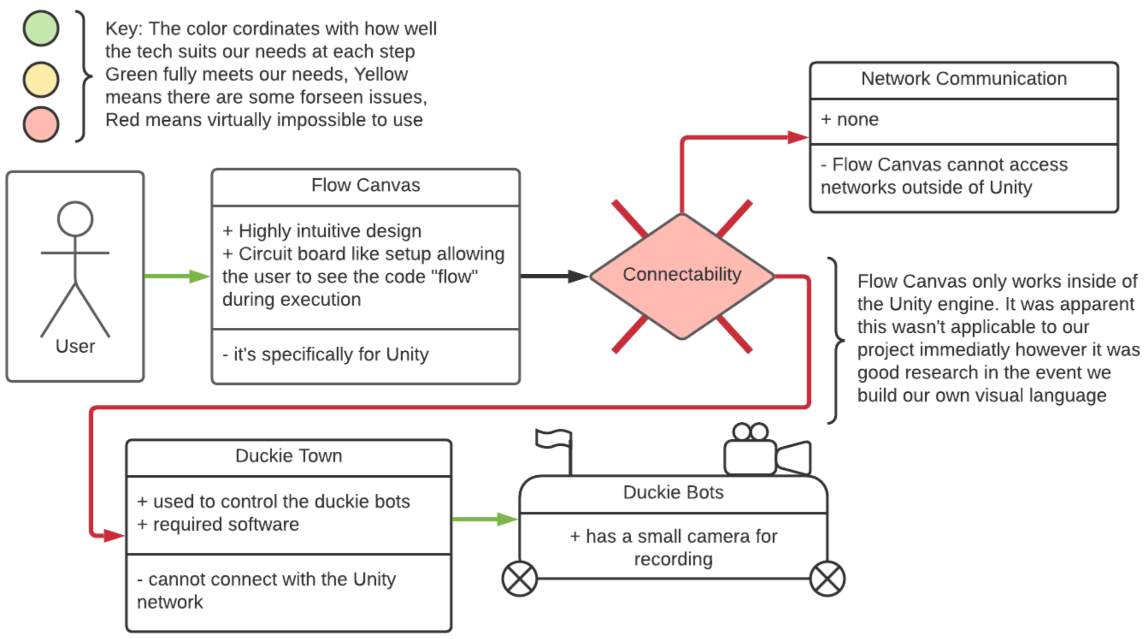

Key: The color cordinates with how well the tech suits our needs at each step Green fully meets our needs, Yellow means there are some forseen issues, Red means virtually impossible to use
User
Flow Canvas
+ Highly intuitive design
+ Circuit board like setup allowing the user to see the code "flow" during execution
- it's specifically for Unity
Connectability
Network Communication
+ none
- Flow Canvas cannot access networks outside of Unity
Flow Canvas only works inside of the Unity engine. It was apparent this wasn't applicable to our project immediatly however it was good research in the event we build our own visual language
Duckie Town
+ used to control the duckie bots
+ required software
- cannot connect with the Unity network
Duckie Bots
+ has a small camera for recording

Figure 5. Diagram of Technological Feasibility: FlowCanvas

3.4.6 Analysis Summary

This table provides a summary of how well it fits our requirements. A score of 5 is a great fit for our requirements and a score of 1 does not fit what we need at all.

Table 1. Summary of Technology Feasibility for Visual Programming Languages

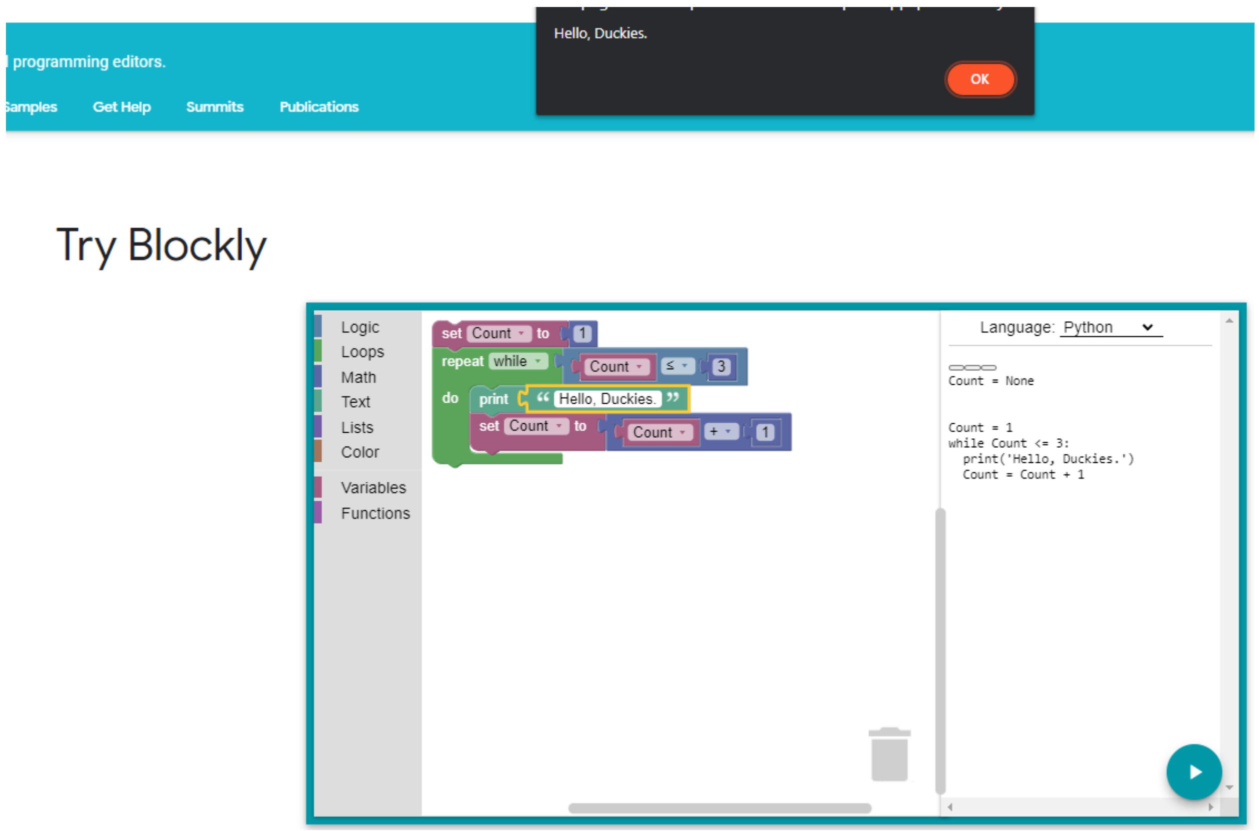
Scores 1-5	Scratch	Snap	Ryven	Blocky	Flow Canvas
Ease of use	5	4	3	5	5
Language interface with Duckiebot	3	3	2	5	0
Network interface with Duckiebot	4	4	3	4	0
Documentation	4	1	3	5	5
Total	16	12	13	19	10

3.5 Chosen approach

We plan to use Blockly for our project. Its ability to export blocks directly into Python or C++ elevates it above Scratch, which can only export to JavaScript. As Duckietown is written in Python and C++, this is crucial. Our main task will be creating an extension for Blockly that implements the Duckietown software ecosystem as visual blocks, allowing children with no prior programming experience to control Duckiebots with ease.

3.6 Proving feasibility

We will prove the feasibility of our chosen approach by writing an extension for Blockly that demonstrates the basic features of Duckietown, such as basic movement. Only once we have shown this is possible will we move on to more advanced features, such as movement within physical boundaries and certain environmental conditions. Overall, to write the extension we'll need the back-end language to be either Python or C++, figure 6 shows a demo of Python being used as the back-end language.



Hello, Duckies.

```
Count = None


Count = 1
while Count <= 3:
  print('Hello, Duckies.')
  Count = Count + 1
```

Figure 6. Blockly Demo

4. Technology Integration

Many of the pieces of technology we have to use for this project were already decided for us. We are required to use a Duckiebot, which requires us to use ROS and Docker. The only portion of this that was up to us to decide was the visual programming language that would best meet our needs. We have chosen Blockly which runs in a web application. Our final product is anticipated to look something like this.

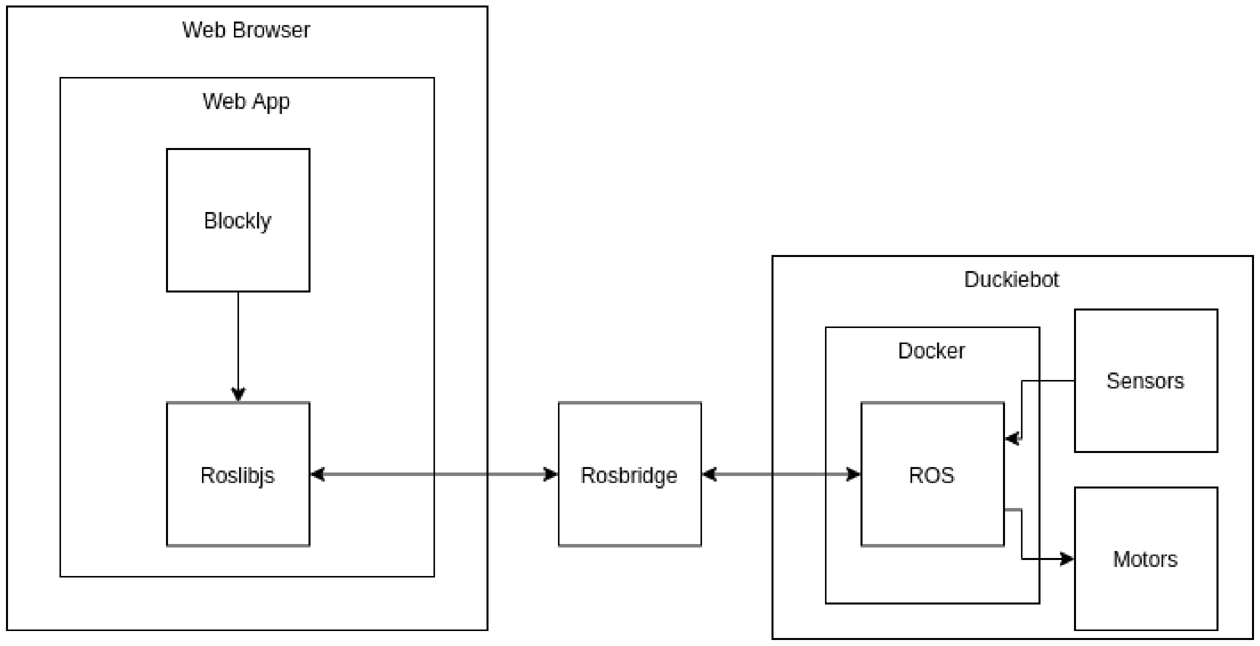


Figure 7. Diagram of System Using Blockly

Blockly and roslibjs run in the web app, which will run in the browser. The Blockly portion of the app contains a library of blocks that invoke roslibjs in the background. Roslibjs communicates with rosbridge. It sends commands to rosbridge which are communicated to the ROS instance running in the Duckiebot, which in turn controls the wheels of the bot. Roslibjs can also receive input back from the Duckiebot via rosbridge. This will mostly consist of data from the bot's camera. What data roslibjs receives, and what roslibjs does with that data, are determined by how roslibjs was invoked by the Blockly blocks.

5. Conclusion

In a world that continues to evolve technologically, it is often difficult to include youth in the programming world. This is especially true in terms of the research, development, and creation of self-driving vehicles. Our client, Dr. Nghiem, wants to create a software that allows kids to be able to learn and have access to easy programming tools to experience the creation and fun behind smaller, self-driving robots. Our product will utilize Blocky as a foundation to build software. This software will be centered towards a younger audience by creating a visual programming language to simplify the programming process needed for Duckietown robots. Our solution should create software that Dr. Nghiem can use during youth summer camps. In the future this would make STEM more available for youth, as well as build on the research done by others in the creation of autonomous vehicles.