Memo

To: Professor Pisano

From: Linden Adamson, Jason Calalang, Tara Gill, Zhilang Gui, Cole Resurreccion, Raina Yin

Team: Team 7 - Bike for Blind

Date: 11/24/2024

Subject: First Prototype Testing Plan

**Boston University**

**Electrical & Computer Engineering**

**EC463 Senior Design Project**

**First Prototype Testing Plan Report**

**Blind Bike**

****

**By**

**Team 7**

**Bike for Blind**

**Team Members**

**Linden Adamson** [lindena@bu.edu](mailto:lindena@bu.edu)

**Jason Calalang** [jasonc16@bu.edu](mailto:jasonc16@bu.edu)

**Tara Gill** [taragill@bu.edu](mailto:taragill@bu.edu)

**Zhilang Gui** [zgui@bu.edu](mailto:zgui@bu.edu)

**Cole Resurreccion** [coler@bu.edu](mailto:coler@bu.edu)

**Raina Yin** [ryin2@bu.edu](mailto:ryin2@bu.edu)

**Required Materials/Equipment**

**Hardware:**

* Webcam (iC700)
* Servo Motor (MG 996H)
* Microcontroller (Arduino Uno)
* RGB-D Camera (Intel RealSense D415)
* IMU/GNSS (VectorNav VN-200)
* Computer (NVIDIA Jetson Nano)

**Software:**

* Python3 Scripts:
  + Run live interference on FastSAM
  + CARLA script
* Segmentation Models:
  + FastSAM model checkpoint
  + FastSAM git repository
* CARLA (Simulation software)
* ROS2
* GitHub Repository: <https://github.com/calsfu/AutonomousTrike>

**Set-Up**

Each of our sensors must be able to communicate with our onboard computer, the NVIDIA Jetson, through a ROS2 bridge. Our IMU/GNSS and RGB-D cameras must have a physical connection to the Jetson. One necessary feature of our bike is image processing for perception. For segmentation to improve our bike's performance, it must be accurate in real time. To create the segmentation model for the prototype, first, the FastSAM repository has to be cloned, and the files necessary from our repository (under ./Models/FastSAM) have to be downloaded. One of the model checkpoints must also be downloaded and placed in the appropriate folder. CARLA 0.9.15 and the proper .py files to run CARLA have to be downloaded. Within the folder CARLA was extracted within, make a directory “scripts” and within that another directory “data.” The script “view\_of\_vehicle.py” should be downloaded from our GitHub and placed into “scripts.” Additionally, any .png file should be placed within “data.” To set up the braking system, the Arduino, servo, and a button must be wired so that the button can trigger the movement in the servo. The servo must have sufficient weight (1-2 kg) on a cable attached to its level arm. This button represents an electrical signal that our bike will send to stop the bike. The weight represents the force needed to brake the vehicle, and the cable represents the electromechanical braking system that bikes typically use.

The set-up for the prototype is component-based, where all the individual tests comprise key features of the final product. They are comprised of:



**Pre-testing Setup Procedure:**

Computer:

1. Clone the AutonomousTrike repository from GitHub on Ubuntu computer.
2. Go into the repository with *cd AutonomousTrike*
3. Run *sudo docker buildx build -t ros2-humble-<date>:latest --load .* (This process will take a while).
4. To start the docker image, run *sudo docker run -it -v /home/user/AutonomousTrike/:/workspace/AutonomousTrike ros2-humble-<date>:latest*
5. Go to the ros2 workspace with the command *cd ~/AutonomousTrike/ros2\_ws*
6. Build the project with *colcon build*
7. Source the build with the command *. install/setup.bash*

IMU/GNSS:

1. Connect the VN-200 to the computer via USB.
2. Test connection with the command *ros2 run vectornav vectornav*

Intel Realsense D415:

1. Connect the D415 to the computer via USB
2. Install Realsense SDK with *sudo apt install ros2-humble-librealsense2\**
3. Test the connection with the command *ros2 run realsense2\_camera realsense2\_camera\_node*

Segmentation:

1. Connect webcam or video device to computer
2. Run liveinterference.py

CARLA:

1. Run CarlaUE4.sh to begin the simulation.
2. In a separate session, run view\_of\_vehicle.py.

Braking System

1. Attach a cable and weight to the servo motor arm
2. Upload Arduino code to the servo motor

**Testing Procedure:**

VN-200:

1. Run *ros2 run vectornav vn\_sensor\_msgs* to convert raw data into sensor messages
2. Run *ros2 topic list* and look for the topic /vectornav/imu
3. Run *ros2 topic hz /vectornav/imu*
4. Run *ros2 topic echo /vectornav/imu*
5. Leave the IMU/GNSS face flat on the table.
6. Test the linear acceleration reading by turning the IMU/GNSS 90 degrees in each Euler angle direction, then back to a neutral position.
7. Test the angular velocity by quickly turning the IMU/GNSS 90 degrees in each Euler angle direction, then quickly turn it back to a neutral position.

Intel Realsense D415:

1. Run *ros2 run realsense2\_camera realsense2\_camera\_node*
2. Run *rqt* in a to open rqt.
3. Select Plugins -> Image Viewer
4. Select /camera/color/raw\_image to view the RGB view
5. Select /camera/depth/image to view the depth view.

Segmentation:

1. Stand (or hold distinct objects) in front of the camera.
2. If multiple people are in front of the camera, write down the number of people in front of the camera, and count how many people the Segmentation Module recognizes (as separate people).
3. Repeating steps 1-2, measure from 1m, 3m, and 6m away from the camera.

CARLA:

1. Visually confirm the RGB and depth cameras are in sync, showing view from front and back of the moving vehicle.
2. Display data from in-game accelerometer, throttle, steering, and braking values in terminal.

Brakes:

1. Push button to tell servo to move into “brakes activated” position
2. Note down how quickly servo reacts to this signal and how smoothly it rotates
3. Repeatedly press the button, 10 times within 30 seconds, and see if the braking system works consistently

**Measurable Criteria**

IMU/GNSS:

* When *ros2 topic hz /vectornav/imu* is run, we are expecting to see 20 Hz for the sample rate.
* When the IMU/GNSS is flat on the table, the linear acceleration should read 9.81 ±.2 in the z-axis and 0 ± .2 in the other axes. The angular velocity should read 0 ± .2 in each axis.
* When the IMU/GNSS is turned 90 degrees in each Euler angle direction, the linear acceleration should read 9.81 ± .4 in the direction turned.
* When the angular velocity is tested, its absolute value should increase in the Euler angle at which it was turned. The opposite angular velocity should be present when turning it back.

Intel Realsense D415:

* The camera should run at 10-20 FPS, verified under the Hz ROS2 message.
* The depth data should be accurate up to 5m from the camera.

Segmentation:

* Every person (and ideally objects) should be recognized at long distances.
* When FastSAM is running on the camera, it should recognize three different people at short (1m), mid (3m), and long (6m) distances.

CARLA:

* Stability of the simulation:
  + Under a certain framerate, hugely dependent on the computer’s gpu, the simulation becomes unstable and the vehicle ceases driving.

Brakes:

* The servo motor reacts to the signal within 200 milliseconds
* The servo motor consistently reacts to the signal: it moves to brake applied position and back 10 times within 30 seconds

**Score Sheet**

Segmentation:

|  | Short (1m) | Mid (3m) | Long (6m) |
| --- | --- | --- | --- |
| People in frame (ground truth) | 3 | 5 | 5 |
| Distinct Segmentation masks (on people only) | 3 | 5 | 5 |

**Hardware Pinout/Wiring Diagram**

**Figure 1: Circuit diagram for the braking system prototype**

**Conclusion**

**IMU/GNSS:** The INS system will be used for localization and navigation. Since we are planning to use Google Maps, we will need to know the current location of the bike. The INS system will use GPS to get an initial location. If the GPS connection fails, possibly to a tall building or tree, it will switch to using dead reckoning with the IMU. For these reasons, we need an INS that samples fast and is accurate. Based on our findings, the VN-200 will work well. Our measured linear acceleration error was between 0.2, which is a very low error. It also samples at a rate of 20 Hz, which is more than fast enough for accurate data.

**Intel Realsense D415:** The depth data from the camera is necessary for localization, obstacle detection, and mapping which we need to control the steering/braking of the bike. Localization involves determining the bike’s precise position in its environment, while obstacle detection focuses on identifying and classifying potential hazards in the bike’s path.

**Segmentation**: We will be using depth information to estimate the distance to objects and people within the vicinity of the bike, allowing us to make timely decisions when controlling the steering and braking of the bike. By incorporating object recognition, we will be able to distinguish between people and various other types of obstacles. This will allow us to improve the safety of our autonomous bike by responding differently based on the type of obstacle.

**CARLA:** CARLA will be our primary method for designing and iterating our computer vision module, which is the heart of our system. Setting up the simulation and running it concurrently with other scripts and software can be a challenge of balancing packages and python versions. Ensuring that we can interface with the simulation using scripts of our own is important as it pushes us through this step, allowing focus on the building and training models in the near future. Additionally, stress testing computers’ GPUs is important both for their ability to run CARLA smoothly and with stability, but also as an illustration of their ability to train and run machine learning models. By running our CARLA script on it, we ruled out the Boston University supplied computer due to it having too weak of a GPU. We found multiple other computers we have SSH access to that are powerful enough for our purposes after they ran the script with real-time accuracy.

**Brakes:** To automate the braking system, we decided that using a servo motor to pull on the bike's cable brakes is practical and effective. The selected servo motor responds to signals within 0.2 seconds and provides smooth, controlled rotation, ensuring safe and reliable braking. During our prototype testing, we determined that implementing a real-time operating system (RTOS) is essential for the braking system, which can be achieved using Arduino. Since one of our biggest project requirements is user safety, RTOS is crucial because it ensures that tasks are executed within precise timeframes and prioritizes critical functions. For example, applying the brakes to avoid accidents must take precedence over less urgent tasks like updating navigation or user interface features. This prioritization ensures safety by triggering braking commands promptly and consistently within milliseconds. Overall, we plan to integrate the prototype braking system into the final design because it is efficient, safe, and straightforward to implement, aligning with the project's safety goals.