LookSee Surveillance Robot

Stavros Avdella, Austin Pena,
Tyler Wallace, Jade Zsiros

University of Central Florida, Department of Electrical and Computer Engineering, Orlando, Florida, 32816, U.S.A.

Abstract - Current surveillance technology is growing rapidly more advanced as cameras improve in quality and computer vision algorithms become more reliable. However, variation within the field is relatively limited, and surveillance options do not exist for all use cases. As a low-cost alternative to a network of security cameras or a team of human security guards, we propose LookSee, a small robot that can wander an empty home or business, and call for help if an intruder is detected. LookSee navigates using the follow-the-gap behavioral navigation algorithm, and detects people using NIVIDA's built in deep learning libraries. All of this is managed by ROS (Robot Operating System).

Index terms: robotics, reactive paradigm, computer vision, follow the gap, surveillance

A photo of LookSee.

Introduction

LookSee is a low-cost, consumer-level surveillance robot intended to patrol an indoor area for the purposes of a small business. Many small grocers and offices would like the security of knowing that their building is safe during hours when regular employees are not there, but cannot afford to hire a night guard or install an expensive camera system. LookSee will be able to be purchased, removed from the box, and used with minimal setup time for less than 1000 dollars.

LookSee can autonomously patrol indoor facilities when no employees are present and detect human intruders. If an intruder is detected, LookSee begins a video call with an emergency contact, who can attempt to communicate with the intruder through the onboard microphone and speakers. A desktop interface will be available for use by the emergency contact.

Our group had two primary considerations in mind for the design of this project: a focus on our individual specific career interests, and achievability without physical contact with one another. Due to the COVID-19 pandemic, we were separated for almost the entirety of the project. We selected parts that were either easily attainable or already in our possession, as many of the parts we initially selected were out of stock or unable to be shipped. We designed the robot to have four discrete components: the electrical system, the computer vision system, the navigation system, and the user interface, so that each of us could develop independently and integrate our components late into the project.

Mechanical Construction

The construction of our robot consisted of 3D printed parts that were used to mount all of our components. The base plate is a filament printed DonkeyCar base plate made specifically for our RC car. Using a resin printer we made a mount for the Jeston Nano to be mounted on it as well as mount on the top for Lidar.

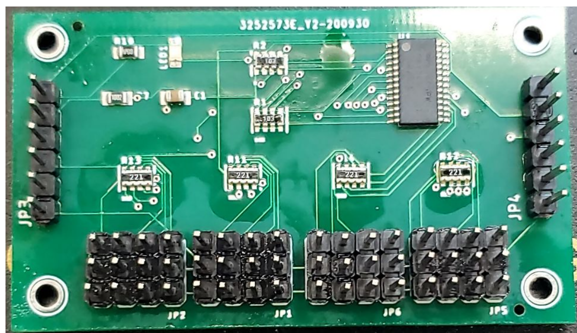
Electrical Design

A. PCB

For our PCB, we have designed a driver that has multiple inputs for all the motors and servos we have. The processor for this PCB is a PCA9685 chip that is an I2C-bus controlled, 16-channel controller. The chip is mainly used for LED applications, as it is capable of pulse-width modulation (PWM) to reduce the power delivered by an electrical signal. We use this technology to control the amount of power delivered to our motors and servos. This is an essential part of the project, since we want to have no delay in our motor output commands. If our microprocessor had handled this, there would be output delays.

We designed the PCB using Autodesk EAGLE. We laid out the parts and components we needed, and then put it on a board to lay out our components and wires. The manufacturer of our PCB was JLCPCB.

The primary processor of this robot is the Jeston Nano. The PCB we made acts as the Jetson's nervous system, responsible for all motor commands and functions.



B. Sensors

LookSee has a large variety of sensors to facilitate its several functions. All of them are connected directly to the Jetson.

The web camera on our robot is used by ROS, and has a dedicated topic that all other nodes can see. It is used by both the computer vision code, which detects human intruders, and the user interface camera, which shows the video stream.

Because our video call software. Jitsi Meet, requires a dedicated camera and cannot draw from the ROS topic, we have an additional webcam on the robot that is used only for the video call. It does not serve as a sensor otherwise.

The LiDAR we selected, the RPLIDAR A1M8, is also connected directly to the Jetson. We use it for navigation. We discussed using both the camera and LiDAR to navigate, but ultimately decided that since the intention of the robot is to navigate a space that should be largely devoid of obstacles, a reactive, LiDAR-only algorithm would suffice. The unit we selected takes scans only in two dimensions, one at each of 360 degrees. We selected this LiDAR unit primarily because of cost. It has served us immensely well.

The thermal camera we selected was initially picked for its compatibility with the Raspberry Pi, rather than its ability to integrate in with ROS. After the header was soldered (and tested for continuity) we proceeded onward. Though upon a shift to the NVIDIA Jetson Nano the library warned that the sensor may be potentially damaged by the given library. Though we searched for an alternative solution to this issue, it appeared that despite the few libraries that existed; the issue lay with the Melexis library and EEPROM corruption. (Which contains unique calibration parameters to the sensor.) So if the values are corrupted, it damages usability of the device. Additionally, the lack of integration with ROS discouraged us from pursuing it in the final iteration of our project.

Software Design

A. System Structure

The Jetson Nano hosts all the required software for operation of the platform. There are two main workspaces, the catkin_ws which hosts all the ROS code and the webpage_ws which hosts all the dashboard code.

The catkin_ws has a `src` folder where all the relevant code to our project is. The `src` folder is organized into packages, both third party and packages we wrote ourselves. The custom packages include robot_bringup (contains required files to launch the robot), robot_control (contains required files to control the robot's movement), robot_msgs (contains all custom msgs, actions, and services), laser_values (contains code to receive LIDAR data and follow the gap algorithm). The third party packages we used are RPLIDAR (used for receiving LIDAR data and publishing to a topic), i2cpwmboard (used for communicating with the servo driver), and ros_deep_learning (used for image detection).

The webpage_ws hosts all the code related to the dashboard. It contains a HTML and Javascript file responsible for creating the entire dashboard. The HTML file loads the roslibjs library, VueJS, and TailwindCSS all from their respective CDN’s. The Javascript file contains all the Vue code which makes the calls to the ROSBridge JSON API and controls the interactive joystick created in Vue.

On a normal run, several pieces of software must be started. The first is ROS itself, which allows all of the other launch files to be used. Next is the web server, which allows the robot to receive commands over a web page interface. After that, robot_bringup is called, which initializes the camera and LiDAR sensors, as well as the camera detection node. From there, the phone call software is turned on, as well as the navigation node, which utilizes the LiDAR that was initialized in robot_bringup.

Each of these nodes fall almost entirely into two separate systems. The intruder detection system starts with the camera, which publishes image data to ROS. The detectnet node subscribes to this image data and uses it to scan for recognizable objects each time the camera refreshes. If an object is detected, information about that object is written to the detectnet/detections topic. The alert node subscribes to the detectnet/detections topic, and scans each detection to determine whether the detected object was a person. If it is, the alert node sends a text message to the currently designated human user with an invite to a video call.

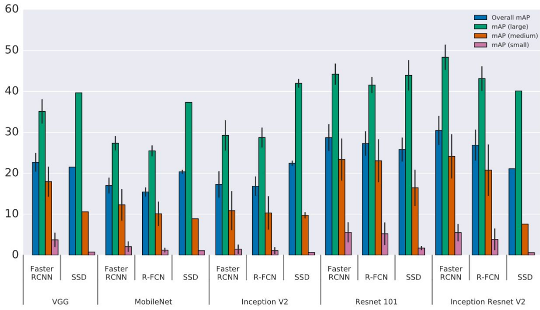
The other system is navigation. The navigation system consists first of the LiDAR, which publishes scan data that is read by the navigation node. The navigation node processes this data and uses it to make Twist messages, a type of movement command data utilized commonly in ROS applications. The other type of navigation is achieved through the web server, which contains a virtual joystick the user can use to drive the robot remotely.

B. ROS

The Robot Operating System, or ROS, is a free and open-source software that runs on Unix-based operating systems, primarily Linux. It uses a publisher-subscriber model to facilitate communication between user-written code, sensors, and output devices. LookSee is built to use ROS as the structure that ties the above modules together.

C. Computer Vision

Object detection using computer vision was done using NVIDIA’s DetectNet node which is built on TensorFlow and uses TensorRT for optimization. When looking for different solutions we used a Google research paper that compared and contrasted the different neural network architectures based on performance [1]. We found that the combination of the SSD and mobilenet architectures was the most lightweight and performant pre-trained model that fit our accuracy and performance requirements. The Single Shot Detector using the MobileNet for feature extraction performed the best when it came to the accuracy metric mean average precision (mAP) and GPU time in milliseconds for each model. We ended up using the Single Shot MobileNet v2 (ssd_mobilenet_v2) which has been trained on the COCO (Common Objects in Context) dataset. It was able to accurately detect a person just by their legs, arms, torso, head, etc along with additional objects.



Mean Average Precision (mAP) of Different CNN Architectures [1]

GPU Time Measurement of Different CNN Architectures [1]

The NVIDIA ROS integration of the detectnet software provided the detections and the output video as topics in ROS that other code could subscribe to. This allowed for a simple and seamless implementation.

D. Navigation

a. Reactive paradigm

LookSee is designed with the Reactive Navigation paradigm in mind. Reactive navigation uses immediate sensor input to decide the corresponding driving command given the environment and decision making parameters. These set cases should wholly manage the swath of conditions the robot may encounter in the surrounding world.

We considered using a more robust hybrid architecture such as HectorSLAM. However, we ultimately decided that as the robot is seeking intruders anywhere in a building, a “roaming” behavior pattern is completely adequate for this use case, and could improve the security of the system due to its semi-random drive pattern. Additionally, we implemented the Follow the Gap algorithm with our own code, while implementation of any kind of mapping algorithm would have likely involved mostly downloading prewritten code online.

b. Follow the Gap

LookSee was initially conceived with the concept that we should seek the largest gap (Find the Gap) between two objects, (eg. a wall, and a hallway) and by virtue of the sensor readouts having the largest discrepancy in range there existed some gap for us to proceed towards and navigate through. Although simple in concept it seemed fairly robust. This, however, has quite a few pitfalls for our particular use case. Chiefly, it doesn’t account for non-holonomic use cases. When controllable degrees of freedom are equal to the total degrees of freedom, it is perfectly acceptable to make a linear pursuit towards a point, and then adjust to the next desired position. To conceptualize, you can think of the difference between how a car might parallel park in comparison to a Roomba or Turtlebot would parallel park.

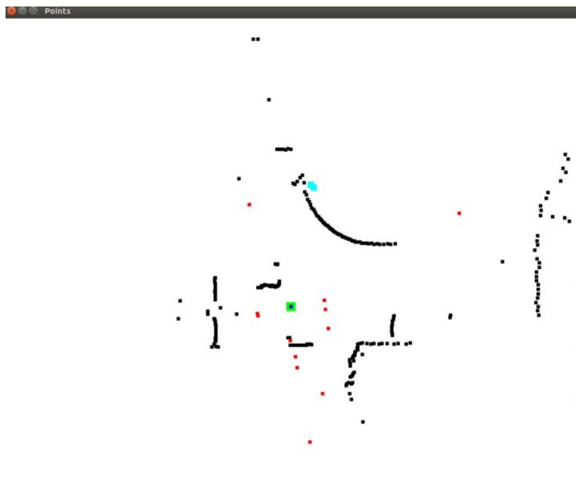
Intuitively, heading for the longest possible distance readout between the widest gap available seems great. However, this also fails to account for the width of the car, and how closely you may be cutting near a wall. In order to prevent the robot from crashing into the corners of nearby objects we initially thought that by simply extending the wall of the nearest point to by a bubble (an inflated size increased by the radius of the robot) would solve our issue, effectively preventing any bumps into the nearest object but this turned out to rather ineffective. Better yet, it was decided we should translate all readouts into the configuration space of the robot; all discrepancies would flag LookSee to overwrite the nearest distance value over the further distance value, effectively identifying walls (via the existence of a discrepancy) and also every obstacle would have a bubble and effectively have an equivalent representation where our robot is now just a point object. Meaning if our point robot representation can fit through two inflated obstacles, we are effectively sure that the car will pass the obstacle. In order to solve this for our particular case, a set amount of angle ranges (equal to roughly radius of the robot) would be zeroed in a safety bubble adjacent to any

discrepancy greater than half a meter as identified by our lidar. From this new filtered array the furthest point from the robot is chosen as the goal. Then, the steering angle is set towards that point and the vehicle is driven in that direction.

c. LiDAR Logic Visualization

The find-the-gap code classifies and moves points in several stages. In order to better understand the logic behind the algorithm, as well as debug, a visualizer was written to display the points at each stage of the algorithm, color-coded by their purpose to the robot.

The visualizer was written using GLUT, the OpenGL Utility Toolkit, which allowed the graphics to be published to the screen inline with the rest of the code.



This is an example visualization. It shows the points that have been detected by the LiDAR. The black points are walls, the red points are potential driving goals, and the blue point is the goal that the robot would select if it were driving at the time of this screenshot.

E. User Interface

ROSBridge is a ROS component that creates a JSON API for a ROS system so non-ROS programs can interact with it.

We used ROSBridge and utilized the exposed API using roslibjs, a javascript library for making ROS API calls.

A frontend dashboard was created using VueJS and TailwindCSS which made calls to the exposed ROSBridge API through roslibjs. A joystick was created using VueJS which published ROS Twist messages to the robot_driver node with a corresponding velocity and angle component. This can be used to control the motion of the robot.

The web page is hosted using a python SimpleHTTP server and is accessed by visiting the IP address of the robot and the exposed port on an external device connected to the local network. Then the UI connects to the ROSBridge API by providing it with the ROSBRidge address URL which is automatically filled in for you. Once connected the controls and video feed appear and allow you to control the robot.

The original plan was to use LIGHTTPD web server, but as responsibilities were shuffled around on the team, one member discovered ROSBridge and realized that the available virtual joystick it provides would be much preferred to the buttons that would be necessary if the other web server was chosen.

The idea with the web server interface is to allow the human user to take over driving of the robot if they see something that they would like to investigate on the camera feed. For example, if the human user is watching the robot over video, and notices that a piece of furniture has been knocked over, they can take over manual driving to find the source of the problem. If the user stops using the virtual joystick, autonomous driving will resume one minute later.

F. Phone Call Interface

Two pieces of proprietary software have been selected to facilitate our video call interface: Jitsi Meet and Twilio. Jitsi Meet is an open-source video call software that can run easily on a Raspberry Pi, which was the first computer we were using, and Twilio is a professional programmable phone autodialing service. LookSee has its own phone number through Twilio, (508)-452-6291, and when

an intruder is detected, it will text the user to invite them to a video call. It does this a maximum of one time per two minutes. This was achieved through a ROS node programmed in Python that listens to the image detection topic.

At this point, LookSee does not make phone calls. This was a tradeoff that ultimately required us to sacrifice some functionality. In order to send the user the link to the video meeting, a text message simply had to be used. A phone call from the robot would be more intrusive, and therefore more useful, but would not allow the ease of being able to click a link. The original plan for the robot was to use the Skype developer API, which allows direct video calling, but since its capability on Linux operating systems is limited, we decided to prioritize the stability offered by Jitsi Meet and Twilio.

Conclusion

A. Lessons Learned

We faced many challenges of various degrees over the course of the construction of this robot. The biggest challenges we believe were most informative are detailed below.

The most obvious challenge was that this robot was constructed amid the 2020 COVID-19 pandemic. The members of this team met exactly one time in person, and all other development was achieved by writing code on separate computers and sending over email, or passing the robot from member to member. Each member had possession of the robot one time, with the exception of Tyler Wallace, who had it twice. During this time with the robot, each member had to effectively make certain that their component was ready and complete, under the assumption that they would get no more development time.

We would absolutely never recommend that any other team do this. We were able to achieve the base version of the robot that we originally envisioned, but only because we carefully planned out exactly who would be responsible for each component, and designed the project with minimal codependency in mind. Even then, responsibilities shifted over the course of the project, and the order that components were added to the robot was not necessarily intuitive.

Unrelated to the pandemic, we learned that the Raspberry Pi 4 was inadequate for the processing requirements of this robot and had to switch to an Nvidia Jetson Nano. The Nano was a very good piece of equipment that served us well, but did cause some logistical issues because the entire team had been developing software on Raspberry Pi 4s up until that point. The change in Linux versions after the selection of components was not at all ideal. However, ultimately, the change was more than worth it. The robot is able to run several very intensive software processes simultaneously.

An original plan we had for the robot was to identify human intruders using a combination of computer vision and thermal vision. The idea was to identify potential locations where a human could be in an image, and seek the temperature reading at the corresponding location to improve the estimate of whether a human was there. This turned out to be completely unnecessary. The deep learning ROS node we are using had better than 95% accuracy at detecting people, and only had trouble if a person passed within one foot of the camera. There was no need to verify the camera results with thermal data.

B. Results

The robot performs extremely well under ideal conditions. However, not all conditions are ideal. This section details the actual performance of each component of the robot.

Computer Vision. The object detection on this robot is effectively perfect. During testing, it was only unable to recognize a person if that person was within one foot of the camera. However, it would occasionally recognize objects that are typically associated with people, such as shoes, and raise an intruder alert falsely. In a business setting, where shoes are unlikely to be laying around, this would probably not be a problem.

Computing Power. The Jetson's computing capacity is entirely adequate to run all of the software

components with no significant slowdowns or bottlenecks. However, running all software components will cause the robot to overheat within 30 minutes, usually around 20. Without running the computer vision nodes, the Jetson can run effectively indefinitely with no cooling problems. For a commercial version of LookSee, a more robust heat management system would be absolutely necessary to be able to run LookSee at its intended duration.

Navigation. The Follow-the-Gap algorithm was immensely successful. LookSee can drive around a room with a changing layout as long as its batteries allow without getting stuck or running into anything, and we haven't found a configuration of walls that causes it to be unable to continue driving. However, the navigation computation does have a little bit of lag. Due to the relatively slow speed of the robot, as well as its constant updating, this is unnoticeable under normal conditions, but if an obstacle were to be suddenly dropped in its path, the robot can take up to 3 seconds to react. There are not many situations that would cause this to be a problem, but those that would could indeed cause damage to LookSee.

User Interface. The user interface does not have any severe issues.

Phone Call Software. The ROS node that decodes the detection data and texts the user when a person is detected has no issues.

C. Future Development

The concept of LookSee certainly has greater potential than what is achievable in a single Senior Design course. We have a few features that, given more time, we would consider worthwhile to implement.

Video Backup. The most useful feature for potential commercial sale of the robot would be to back up key video footage to internal or external storage for later review. We didn't implement this because we decided to prioritize the components that did not lean heavily on full robot integration.

Tamper Detection. An original stretch goal for the project, the installation of a gyroscope would allow the robot to detect when unexpected movement happens, such as being picked up, and either begin recording video, call the user, or both.

Mapping and Route Planning. At this point, the robot's reactive behavior suits its use case perfectly well, but there are some benefits to preplanning a path. Implementing mapping would allow the robot to return to a set point at shutdown, and ensure that it scans all rooms, although it would sacrifice an element of unpredictability.

Smart Start. Mobile robots require a lot of battery power, and LookSee is no exception. To stay at its current price point, a significant increase in battery capacity is not feasible. To retain battery, a future iteration of LookSee could drive for a short period at a set time, and begin driving again either at intervals or when a noise is heard.

Team Members

Jade Zsiros is a senior Computer Engineering student at the University of Central Florida. She is a current Telemetry Engineering Intern at the National Aeronautics and Space Administration, where she has been offered a full-time position after her graduation. She previously worked as an intern at Aerojet Rocketdyne. At UCF, she was an officer and later president of the Robotics Club for three years, during which time she led an autonomous IGVC team and a prototyping team. She did LookSee's system design, phone call software, and worked on the navigation with Austin.

Austin Pena is a senior Computer Engineering student at the University of Central Florida. Though the opportunity to work as a CNC programmer is available to him. He's hoping to find a more degree oriented job somewhere sunny with good waves. He programmed LookSee's navigation.

Stavros Avdella is a senior Computer Engineering student at the University of Central Florida. He is further pursuing his education with a Masters in Engineering Management. His hopes are to become a

project manager for a hardware development company. He did the mechanical and electrical design and assembly for LookSee.

Tyler Wallace is a senior Computer Engineering student at the University of Central Florida. Tyler works part time for University of Central Florida as a full stack web developer and is pursuing jobs in the fields of software development and robotics. He programmed the web server, web site, and implemented the computer vision for LookSee.

References

[1] Jonathan Huang and Vivek Rathod and Chen Sun and Menglong Zhu and Anoop Korattikara and Alireza Fathi and Ian Fischer and Zbigniew Wojna and Yang Song and Sergio Guadarrama and Kevin Murphy, . "Speed/accuracy trade-offs for modern convolutional object detectors". CoRR abs/1611.10012. (2016).

[2] Nathan Otterness. “The ‘Disparity Extender’ Algortihm, and F1Tenth”. Web. 2019.