Harvey

An Autonomous Robot Utilizing Vision Recognition

Jacob Pitzer, Barry Huey, Jaqua Starr, Mounir Barjaoui

California University of Pennsylvania

CSC 492 – Professor Jeffrey Sumey

**Table of Contents**

**Document Description** …………………………………………………….... Page 4

**Implementation Details**….………………………………….…………...….. Page 6

* **Project Overview and Objectives** ……….…………………..…………............... Page 6
* **System Block Diagram** ……….………………………………………..................Page 8
* **Project Implementation** ……….……………………..……..…………………….Page 9
* **Vision Recognition** …………………….…………………………………………. Page 9
* **Arm & Drive System** .…………………………………………………………… Page 10
* **Path Stack** ……………………………………………………………………….. Page 11
* **Implementation Conclusions**…………………………………………………..... Page 12
* **Harvey Testing Overview** ………………..…………………………………...… Page 14
* **Vision Recognition Testing** …………………………………………………...… Page 16
* **Arm Testing** ……………………………………………………………………... Page 16
* **Path Stack Testing** …………….……………..…………………..……………… Page 17

**Software Engineering Principles** ………………………………………..... Page 19

**User Guide & Manual** ………………………………...……………..……. Page 25

**Project Management: Gantt Chart Evaluation**……….....………………. Page 36

**Results/Conclusions: Assessment of Objectives** ………………………..... Page 39

**Appendix:** ………………………………………………………………………………Page 41

* **Appendix I: Team Project Website** ……………………………………….…… Page 41
* **Appendix II: Test Modules**………………………………………………...…… Page 42
* **Appendix III: Code Listings** ……………………………………………..……… Page 48

**Document Description**

The intent of this document is to provide a technical description of the overall design and process of the Harvey Project. This includes a lot of software-specific terminologies and therefore is intended for a more technically inclined audience. Though due to the organization and readability of this document it can also be useful for non-technical readers as well.

This document is the fourth installation in an ongoing series of artifacts related to the Harvey Project. As such, much of the information is derived, in part, from the earlier documentation. This document is intended to build upon the previous concepts proposed in earlier documents, but present what concepts were implemented. It will do this by providing a more technical breakdown of the hardware and software that is used in the Harvey Project.

One of the goals of this document is to provide a Functional Description of the Harvey Project that could be presented to potential clients, investors, or co-developers. It allows a more technical view of the software and hardware involved in the project, and how those individual components interact with each other as well as the physical environment.

This document will also endeavor to address more in-depth aspects of the project, such as the bill of materials. With a Bill of Materials, it is easier to develop a potential business model by allowing clients or investors to quantify the potential financial risks involved. It will also cover more technical aspects like the initialization and configuration of the system components. This section of the document relies heavily on a prior understanding of programming terminology but offers an insight into some of the actual code that was implemented later in the project.

While anyone who has read the previous artifacts for the Harvey Project may recognize many of these concepts from the earlier documentation, this document is intended to present a more technical overview of the system. This document will also be used in the future by engineers charged with maintaining this system. Having the original design document will allow these developers to conceptualize the intent of the project, allowing them to make sense of the corresponding source code. The following document will also go through the extensive testing phases such as the arm, video recognition, etc.

**Project Overview and Objectives**

The Harvey platform was created with the idea of artificial intelligence in mind at the start of the project. Instead of merely directing a rover via human input with a remote controller, Harvey was envisioned to be fully autonomous once it is powered on. This removes the need for people to constantly control the rover and allows for Harvey to be able to work on its own to help accomplish tasks such as moving objects that could prove difficult for people on their own, without the help of a machine.

This project is a small proof of concept model based on that idea, which could be up-scaled to a much larger model in the future that would allow for lifting much heavier objects than this model could handle. This in turn makes Harvey a highly independent and efficient machine which could benefit our society as a whole in a very positive way. Harvey’s core concept is made possible by the following main key aspects to the project.

1.) Being equipped with vision recognition via an onboard camera along with the appropriate algorithms in order to recognize the correct objects that it will be picking up and moving. 2.) The capability of traversing on its own to reach objects in order to pick them up via a path finding algorithm that controls the chassis’ motor functional operations. 3.) Having an articulated mechanical arm mounted to the chassis, which has a gripper attached to it that is able to move up and down and grasp an object. Through the appropriate algorithms in order to issue the arm commands, this allows Harvey to pick up objects that it locates and move around while holding the objects with the gripper.

These key aspects and core concepts form the basis and overview for project Harvey. The objectives that have been established for the project are applied through these fundamental concepts and aspects to the Harvey platform, those objectives are thus follows.

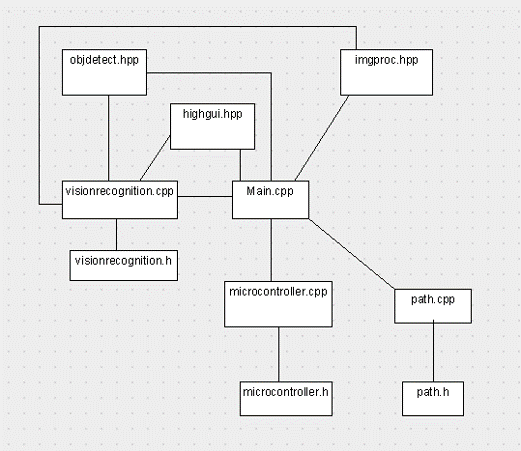
1.) The Harvey platform should be able to correctly locate and identify objects with its camera and vision recognition algorithms. It must be able to identify which objects are the correct ones for it to navigate to, based on classifiers contained within the vision recognition component of the rover’s source code.

2.) Harvey should be able to traverse and navigate to the correct located objects on its’ own via the chassis, wheel servos, and drive motor. It should also be able to return back to its original location after picking up the object. The path finding algorithm must allow for these hardware components to operate properly in order to travel to the given objects.

3.) Harvey should be able to grasp and pick up the selected object it has in its sights once it reaches the object’s location. The articulated arm must be able to move up and down in order to make this possible. Along with that, the gripper must be able to grasp the object properly and not drop the object while carrying it and while Harvey is moving around. The algorithms that issue the proper commands to the arm and the gripper are what make this possible.

With these objectives in place and implemented properly, Harvey will succeed as a highly innovative, efficient, and helpful creation that can benefit the productivity of companies and the physical well-being of numerous workers for professional use, along with society in general for many people to help for personal use as well.

**System Block Diagram**

****

**Project Implementation**

The Following sections outline the implementation of Harvey. These outlines contain the process details of implementation, development issues, and resolution of issues. While some issues were greater and often more potentially fatal than others, there was not a single module that had a seamless implementation. The solutions developed and implemented often remedied the issue in its entirety, or less often, made the fault of issue non-existent or simply found a way around the problem. Development occurred from the beginning of February to the beginning of April, approximately a two-month period total.

**Vision Recognition**

Initially the vision recognition module was to be developed using Intel RealSense and ROS (robot operating system). The team acquired a robotics development kit from Intel during the design phase in the fall semester. However, this process proved to be the of the most folly out of any module.

In a development accident, the original up-board was damaged beyond repair by giving it too much input voltage and current, resulting in irreparable damage. The solution was a simple, but costly one. Two additional up-boards were purchased, in case another one fell the same fate. In order to stop a potential project bottleneck, the up-board core was downloaded to each team member’s personal machines. Essentially, the up-board would be simulated until the software was ready to be moved on board.

Unfortunately, our team did not find similar remedies with Intel RealSense or ROS. While the cause of the faults is still undetermined, it was likely a combination of faulty equipment and broken software. This did not change the fact that the project still needed to be completed. From there, the team decided to move forward and use another Intel product, OpenCV.

The OpenCV implementation on the other hand, was far more successful than RealSense and ROS. The documentation on native libraries provided allowed for our team to learn the system rather quickly. We were able to quickly prototype a vision system and perform adequate initial tests. Using this original prototype, expansion to include several objects and distinctly colored covering shapes was developed and tested rather quickly.

**Arm and Drive System**

During the design portion of this project, An Arduino microcontroller was envisioned to control and call functions relating to the robotic arm and drive system. Our plan centered around using several Adafruit shields to connect the servos and drive motors to the Arduino. This implementation plan ultimately failed because of power issues relating to implementing the arm and the drive system.

To adapt to this problem, the Pololu servo controller was used to communicate with the arm and drive servos. This proved initially successful, as the native script and GUI application allowed ease of use. To communicate with the Pololu board, TTL and XMT lines were built between the Arduino Uno and the Pololu board. This communication did not prove to be fruitful, as communication was not clearly established.

On the third attempt to solve the communication issue, a C++ module was created to communicate with the Pololu Board via serial port. In this case, the third time happened to be the charm, as the servos immediately responded to the commands given in the microcontroller module. From this initial setup, the rest of the functionality for the microcontroller class was able to be developed and tested.

The microcontroller module, now functioning was able to be expanded to achieve maximum functionality. Once basic functionality of movement was achieved, our team was then able to focus on writing and focusing algorithms to smooth out movement, achieve maximum efficiency, and develop further functionality.

**Path Stack**

The last module to be developed, integrated, and tested was the path stack. This decision made logical sense as it was the least complicated, has the least potential for development issues, and could be modified the easiest. The class itself consists of the stack object and no other algorithm. However, the path setting and retrieval algorithm that is found in main, was developed in tandem with the path stack, and as such, is more closely associated with the stack than any other module.

The Path setting algorithm was developed to follow a certain pattern. The basic idea was to simulate a real-life scenario where the robot would travel from a starting point, to an ending point, and capture an object if it is detected. During development, a number of different algorithms and processes were used. The process we chose, highlighted in the Appendix, simplifies the process of location and retrieval while still giving the rover the freedom to move and locate objects.

The Path class contains the path setting and retrieval stack, and member functions that associate with the stack. Stack development occurred rather quickly, as well as integration and testing. One of the original structure idea was to use an array for ease of development. However, this was not a wise design decision as the path had to be dynamic in size. The stack structure of using a struct and pointers was then implemented. From the end of that development phase, integration and testing with the other two modules could then begin. At the end of the development phase for the path stack, the working stack structure combined with the control algorithms created a dynamic module capable of robust searching and retrieval.

**Implementation Conclusions**

Throughout the implementation phase, our team experienced many issues and setbacks. Through our skills and determination, we were able to create solutions and new implementations to defeat these setbacks. From these experiences we can draw the following conclusions over our implementation experience.

One of our greatest strengths was the development of contingency plans. We learned that often enough, initial development plans may go by the wayside. In these situations, it is vital to develop remedies to achieve the end goal despite these setbacks. In our project, we had many issues that could have been fatal if not resolved. The skills that we have developed throughout our University career prepared us to deal with these situations and to create creative or straight-forward fixes to solve the problem at hand.

The strength of initial planning is also a key lesson from the implementation phase. Looking back at the design document, our team developed a solid plan to work in the implementation phase but left out key details that would have been helpful during implementation. We did not know it at the time but gaining a further understanding of the technology we were learning to develop the project would have been essential. Not of how it works, we understood that process. Going back to that point of time, more attention should have been paid to how this implementation and library could fail, and what are strategies and methods to work through such a failure.

The last implementation conclusion is that of simplicity. Our team eventually discovered that the simplest solution to achieve 100% functionality is often the best one. Less complicated modules and functions are easier to develop and contain less faults than more complicated modules and functions. RealSense and ROS are not simple applications to say the least. OpenCV, a very sophisticated library in of itself, is easier to understand and develop. That knowledge would have greatly assisted our team during the design and implementation phase.

**Testing Overview**

The testing process for project Harvey was carried out through the entire implementation phase of the project. It was not simply at the end of the process after all of the source code is written, as that was not an effective way to test any large-scale project. Each specific hardware component of the Harvey platform has an appropriate section of source code dedicated to it divided into modules and subroutines throughout the program’s code. For every subroutine and module that is created during the implementation phase, the team members tested their specific section of the compiled source code to make sure there was no errors of any sort. Once the source code was fully completed, the entire program was compiled and thoroughly tested before finalizing the program to ensure the source code is as correct and accurate as possible.

The individual hardware components each had their own dedicated scripts within their specific modules and subroutines. These were planned and designed once the board was connected to the individual parts of Harvey. This document established the beginning of this process in the class/function description section and the installation and configuration section.

The Intel IoT board has the object libraries stored on it and has the bulk of the source scripts associated with it. It will relay the program’s instructions to the other hardware components of the Harvey platform. The Arduino microcontroller interfaced with the Intel IoT board and relay basic command scripts to the light sensors along with the wheel and arm servos. These two boards were thoroughly tested to make sure they are functioning properly and ensure that the program’s source code is interacting with the hardware components as they are intended to.

The hardware components to the Harvey platform were also assembled together to fully complete the implementation phase and to begin the final testing phase of the project. As with the software components of the project (primarily the source code), the hardware components were thoroughly tested after the Harvey platform was assembled to ensure that there is no defective mechanical components or faulty wiring. The connections between the individual wires and each hardware component were tested and verified to ensure that no wires come loose or disconnect during the final testing phase.

When the source code was fully completed and the hardware components to Harvey were fully assembled properly, the team members of the project began to setup and establish the designated final testing location for the Harvey platform. The room in which the final testing location was executed was Jaqua’s living room. It had the appropriate operating space setup in a block layout design. The team members placed a strip of black tape around the edges of a large square space in order to create the boundaries for the testing zone.

Inside the square space, various small objects were placed within the boundaries for Harvey to attempt to locate, pickup and move to different locations during the final testing phase. At this point, the Harvey rover was placed inside the testing zone and powered on by a team member to let it operate.

Once Harvey was powered on, there will not be any more need for human interaction during the live testing stage, as all of the instructions and commands built into the Intel and Arduino microcontrollers allowed Harvey to operate autonomously. Once the rover finished its designated tasks, a team member powered off Harvey and the final testing phase was complete.

**Testing Vision Recognition**

Jake’s house was the testing location for vision recognition. While testing the vision recognition on the camera a reoccurring problem was noticed. When searching for objects, the camera would work fine if the back round was a bright and plain color. However, when Jake was going further with the tests he noticed the camera would pick up the decorative wallpaper as “object found”. A solution to adapt around this problem was made. Something simple like putting a plain white medium sized cardboard square behind the object or keeping the rover in a room with bright plain colors should be enough for the camera to depict the object and back round.

The custom objects trained to be used in the program had a high fail rate. It was decided during the design phase that any object failure rate outside of the tolerance of 1% (+/-) 1% would constitute a failure. The Yoda and R2-D2 Figures had a failure rate of 20% and 25% respectively. To combat this problem, our team decided to use objects trained in the OpenCV library. Not only would these objects be reliable, but they were more generic than the objects we initially used.

Once these objects were implemented from OpenCV, the issues surrounding vision recognition vanished. From there, it was easy to finish final testing and development

**Testing the Arm**

Testing the arm was done over Jaqua’s and Jake’s houses. At Jaqua’s house, the arm was assembled. It was quickly realized that the arm was not going to have as large of a degree of movement as previously thought. It now has 4 degrees of movement, a rotator at the base, an elbow, a wrist, and a gripper. Originally, code was going to be written for the arm to move on its own. Instead of code, scripts were made. The scripts were a group of instructions to individual parts of the arm. To test the scripts, Jake and Jaqua had the arm pick up an empty box for a deck of cards. While the arm successfully picked up the box, its movements were rough and sporadic. This was later improved at Jake’s house along with combining the script with a test program. The test program for the arm is Unix friendly and was able to be implemented with the script. The program is a part of the vision recognition and once an object gets within distance, the test program will call the script. The movement speed of the arm was also reduced for smooth transitions. Finalizing the path settings is the last thing that needed to be done.

A test module was developed in C++ to test serial communication to the arm and drive system. This test module used movements to approach and capture objects. Using this module, which was implemented into the final software package, the algorithms for object retrieval and return were tested and finalized.

**Path Stack Testing**

One of the last modules developed, integrated, and tested was the path stack. The test procedure was a practical one. Simulate the rover moving by implementing the path setting and return algorithm by adding a delay and add a trigger for object found at various points of the program.

To accomplish this, test software was developed, most of which was integrated into the final software package. One of the advantages of the nature of this module, was that it can be implemented on any Unix machine. Essentially, this was the only module that can be copied and pasted from Windows to Ubuntu and compile without error, and vice-versa.

Once testing was sufficiently completed, and full functionality was achieved, certain parts from the test program were implemented into the main software package. This process proved to be ideal, as integration and full system test proved to seamless after this process. The path class was by far the easiest class to test but was not without its issues. Many of the class functions needed to be corrected as the destructor was crashing the program when initially implemented. This was corrected and adapted to the rest of the module.

**Software Engineering Principles**

Throughout the course of the implementation portion, our team employed numerous software engineering principles to realize the platforms end goals. The principles used gave our team a greater understanding not just in developing a software system, but how forces in industry use these techniques to maximize projects in a professional setting. The following section highlights in detail the software engineering principle, its description, and how it is related to the development of our project.

**Object-Oriented Design**

Object-Oriented Design (OOD) is a common software engineering principle used in the technology industry. Harvey was developed entirely in the C++ language, which is not exclusively object-oriented, like Java, but was invented for the purpose of realizing object oriented principles. In OOD, entity models are encapsulated to classes, also known as objects, that define attribute variables and action functions. During development, our team extracted entities from previous documentation and created classes encapsulated from those entities.

In total, three classes were written using OOD principles, each being a third of the components of the system. Each class defined the attributes and functions for vision recognition, hardware control, and path setting/retrieval. While acting together as a system, each class performs an entirely separate function and use. Functions and attributes unique to each entity, i.e. vision recognition, are only found within that object. These objects communicate within the main program, where the control algorithms exist. These algorithms then make decisions based upon input and pass appropriate information to each class when needed.

Within the OOD framework, a top-down design pattern was followed. Each module was developed as a subsystem to the entire project. This development pattern allowed for functionality to be built and tested from the highest level of functionality to the lowest. While this was not the only process that could have been implemented; our team felt that this was the best design pattern for our project because it allowed for major functionality pieces and issues to be brought to light immediately during development. This also created an easier implementation of lower-level functionality, as many potential issues were likely solved previously.

**Agile Software Development**

Agile Software Development is the process of rapid prototyping with small development increments. One of the advantages of this process is that it creates opportunities to test increments in small pieces during development, which is more likely to uncover small issues in software which may lead to larger issues or create functionality problems. Another key strength of this method, is that while it is not ideal for large development teams, it is very popular and effective for teams with few members. The project had four members that often could not meet up outside of class very often, and the agile development process was able to accommodate that nicely.

The Agile method was used between February and April, practically for the duration of the entire development portion of the implementation semester. The four main components: main, vision recognition, microcontroller, path followed this process extensively. While there were times where software needed to be written extensively to stay on schedule, the agile process still was the focal strategy even during rushed times.

The main.cpp module was developed along with the other three proprietary classes. When a small bit of functionality was added in one module, its corresponding function was also developed and tested. This not only allowed for testing and development on multiple modules but allowed for cohesion and coupling to be tested on a much smaller scale. At the very end of the development process once each module was tested, the entire main module could be tested as a system.

The Vision Recognition class, vision recognition, is small in size but is the most technical and complicated in nature. Because this class had many different components, staying with the agile process was vital. Each component associated with the object had to be developed on its own (i.e. opening the camera, opening cascade files etc.). After each increment, the vision recognition class was able to obtain more of the target functionality until eventually, target objects were identified correctly.

Arm and drive control was implemented into the microcontroller class. The agile process worked very smoothly with the microcontroller class. The first increment targeted returning servo position and moving the servo to a desired target. From there, small increments of building algorithms and action routines were more easily written. Action routines such as captureObject, and retrieveObject are built up of other functions, essentially making them larger increments of smaller pieces.

Lastly, the path class which contained the path setting and retrieval stack had a majority of its development through the agile process. The stack itself was written and tested very quickly, as a simple stack is not conceptually difficult to build. However, the path setting algorithm, which was implemented in the main class, has its roots in the agile process. Pushing and popping off the stack was the first increment, followed by a more complicated algorithm, and then the final path setting algorithm.

**Portability**

Software portability was an essential principle for the completion of this project. The software had to be developed onto a single computer board, but that meant that only one team member could write software for the platform. Taking this issue into account, the software was developed on each team members computers using the Ubuntu operating system and OpenCV library. Another issue taken into consideration, was the limitation of the board. It is not a full-size computer and only has 32 GB of storage, 2 GB of ram, and a 1.92 GHz processor. Those limitations were taken into consideration whilst developing on personal machines.

The core backend packages, such as OpenCV, the Pololu GUI, and the proprietary software written for the platform were each moved to the up-board and tested for functionality. Since no software development occurred on the board, there were no major development issues. However, there were issues with installing and configuring software, but these did not prove to be fatal in anyway.

Essentially, this means that the software package can be moved and implemented on a variety of other platforms if they meet the requirements to run the system. This creates many possibilities of future development past this course. The portability factor also allows for the software to be run on a personal computer and be connected via Bluetooth or Wi-Fi to run the system.

**White Box Testing**

In this course, our team was exposed to two types of testing during lecture. White Box Testing and Black Box Testing. Our team implemented the white box testing methodology. This meant that during testing the individual conducting the test had access to the code and was able to make changes or add test code to the module or function. This proved to be ideal as it allowed for fatal flaws when uncovered they could be corrected by the person running the test.

During testing of the components, scenarios and code was introduced to simulate events such as finding an object or encountering an error to see what the module or function did. The simulation of events also proved to be handy as many of the components from other modules were not ready to interact with the module or function being tested.

When modules were ready to be brought together, they were tested together for optimal functionality. At initial testing, only two modules were tested together. The purpose of this was to 1. Identify faults and issues between connecting modules and 2. To determine coupling and cohesion level and make improvements where necessary.

At the last level of testing the entire system was tested and prepared for the demo. This stage was not unlike any other part of the testing phase in that it was again tested for system faults and issues as well as cohesion and coupling level.

**Sandwich Integration**

This integration method takes modules and functions and integrates them from a bottom-up approach, meaning lower level functions and modules are integrated together before higher level modules. This process simplified the integration process as less complicated features were integrated together before more technical and heavier components. This process also was implemented because the bottom up integration approach made sense, many of the modules were more easily implemented and integrated in this fashion.

Integration and testing occurred nearly simultaneously. After one part of a module was integrated, it began some light testing, which was highlighted in the previous section. Integration always preceded testing; this is because after integration occurred testing needed to be completed to measure functionality.

For future development, this integration method is highly encouraged as Harvey is very complex for a small robot. Lower level functions and modules will need to be integrated first, as more complex modules may not work without correct implementation of lower modules. This issue was encountered during the integration process, and the remedy was simply leaving the highest modules for last.

**Harvey User Guide & Manual**



**Installed on Chassis Exterior Installed in Chassis Interior**

1. Arm Servos 4) 5V Battery Pack
2. Hand 5) Intel UpCore
3. Camera 6) Pololu Microcontroller

# 1

#2

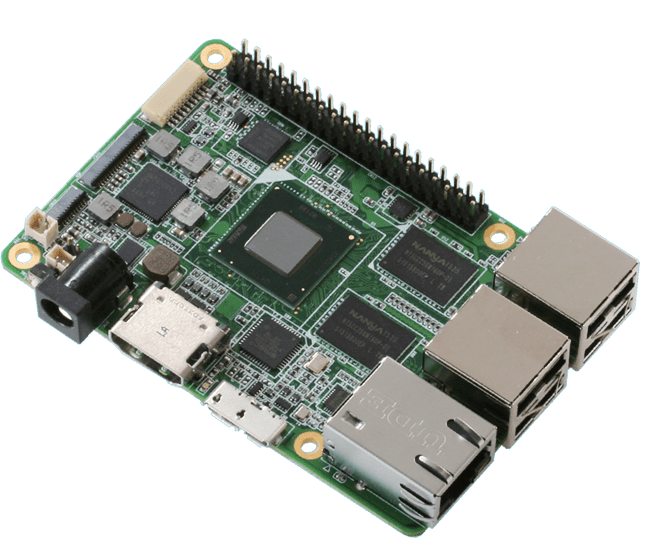
#3

#4, # 5, # 6

#5

Welcome to the Harvey Project. The intent of the project was to provide a proof of concept for an autonomous robot with integrated vision recognition. This robot can be scaled up for industrial use in factories, effectively replacing the need for forklifts and their drivers. Since the system doesn’t include any Graphical User Interface for the end user, this user manual will focus instead on the hardware of the system. The intent of this document is to familiarize the audience with the system as well as provide information for future troubleshooting.

If you think of the robot in biological terms the Intel UpCore is the “brains” of the operation. It is responsible for the overall control of the system. All the individual components are controlled either directly or indirectly by the UpCore. The operating system installed on the UpCore is the Linux based Ubuntu 16.04. This operating system was chosen for several reasons. First is the portability of the Linux based operating system. Since we are using a lot of individual components we needed an operating system that could “get along” with all of them. Another reason is the inherent advantages of using an open source operating system as opposed to a proprietary one such as Windows.



Intel UpCore

Image Via: www.up-board.org/upcore

The “heart” of the system undoubtedly is the Pololu microcontroller. This is the controller responsible for all servo movements. The Pololu microcontroller is controlled directly by the UpCore board. All the physical movements, including drive train as well as arm movements, are all controlled by the microcontroller.



Pololu Microcontroller

Image Via: www.pololu.com/product/207

The camera is the eyes of the unit. It is what is responsible for identifying the object, then determining a pathway to the object. The camera is running inside of an open source library called OpenCv. OpenCv is a collection of libraries that are used for many vision recognition projects. Using these libraries in conjunction with our own code, the system can operate autonomously using the camera for all the data collection that is needed to traverse the physical world.



Logitech c615

Image Via: www.logitech.com/c615

The arm assembly is one of the systems that is used to directly interact with the physical world. It contains a collection of servos that are used to control the arm. All the servos are controlled by the Pololu microcontroller, which in turn is controlled by the UpCore. Once an object is identified with the camera, the drive system will maneuver Harvey into position to retrieve the object. Then the arm will descend to allow the hand to grasp the object. Once the object is firmly in hand the system will return to its place of origin to release the object.



Lynxmotion Gipper, Base, and Arm

http://www.lynxmotion.com/

The other system that interacts with the physical world is the drive system. The drive system consists of the chassis, 4 wheels (each with its own servo), and the main power supply that is located inside of the chassis. The chassis is the component that all other components mount to. Therefore, it can be thought of as the skeleton of the system.



Lynx Motion A4WD1 v2 Robot

Image Via: lynxmotion.com

**Harvey Component Setup: Hardware Configuration**

This section gives an explanation on 1.) How to properly assemble Harvey’s hardware components, and 2.) The correct order to power on the different components of the rover for proper setup and operation. If your Harvey model has already been pre-assembled, please disregard the hardware assembly section and go straight to the powering order setup section.

Hardware Assembly & Configuration: In order to properly configure the Harvey platform, you must first have all of the necessary components that should be included. For visual reference, a diagram of the different hardware components to Harvey and how they are connected is shown in the above section of the user manual. After locating and confirming that all hardware components are on hand, start the assembly process with the mobile chassis, as it is the central hub of the rover’s entire platform. From there, the Intel UpCore board and the Pololu microcontroller should both be mounted to the chassis and connected to each other via the proper wiring. It is important that the wires that connect them to the other components can reach each other. So it is recommended to keep the UpCore and the Pololu microcontroller nearby each other on the chassis.

The drive system, including its 4 wheels and respective servos, all attach to the chassis. The camera is mounted to the base of the chassis, which connects to the UpCore. The arm is mounted on the top of the chassis and is attached to the arm servos, which is connected to the Pololu microcontroller as well. Finally, the power supply is mounted onto the chassis and directly connected to the UpCore. Be sure to remember to connect all of these components together in the prior way shown with the proper wiring. Without the correct wiring, Harvey will not power on or operate at all.

Component Power on Setup: After following all of the steps in the hardware assembly and configuration section, you are now ready to power on Harvey. In order to properly power on the rover, you will first need to connect the 7.4V cable to the motor controller. Next, connect the 6V cable to the Pololu microcontroller. After this, screw on the chassis top with the proper given screws. Next, connect the USB mini B from the Intel UpCore board to the Pololu microcontroller. Lastly, turn on the UpCore by pressing the power button for the power supply. This should activate the UpCore, which in turn should also activate the Pololu microcontroller, along with the camera as well. The articulated arm and its gripper should also now be activated at this point, along with the arm servos. Finally, the drive system wheels and servos should now both be activated and operational at this point.

Signs to look for in order to make sure Harvey is properly powered on correctly are checking to see if all of the lights on the UpCore, Pololu microcontroller, and the camera become illuminated. This shows that there is power flowing from the power supply to the rover’s other key hardware components. There should also be sounds coming from the servos that indicate that they are now functioning and have power flowing to them. The other most noticeable sign of proper startup is once all of the different components are activated, Harvey should start to operate autonomously. This means that there is no need for any more human interaction with the rover until its given tasks are completed.

At this point after Harvey has completed its given tasks, you may feel free to power down the rover by pressing the same power button for the power supply. Harvey should then return to a dormant and non-operational state. If any of these steps produce abnormal results not described here, please see the following troubleshooting section which helps solve possible issues that can result while operating Harvey.

**Troubleshooting Q&A: Resolving Known Problems**

This section provides helpful tips and advice on how to resolve possible issues that could arise while operating Harvey. The following are some of the known problems that can occur:

1.) UpCore Board/Pololu microcontroller not powering on or shutting off without warning: If this occurs, be sure to double check and make sure the power supply is properly charged, and that all of the wiring connecting to the UpCore or the Pololu microcontroller are connected in their respective ports or pins. Particularly important is making sure the wires connecting the power supply to the UpCore are properly in place. If the UpCore or Pololu are still not powering on or behaving abnormally after this, you might have a faulty or dead power supply, or additionally a torn wire in one of the connection cables. There is also the possibility that the UpCore or the Pololu microcontroller themselves could be faulty or dead. Any of these components would need to be replaced in order to allow Harvey to function properly again.

2.) Arm servos lose power during operation: If this occurs, be sure to double check and make sure the power supply is properly charged, and that all of the wires connecting to the Arm Servos are connected in their respective ports or pins. Particularly important is making sure the wires connecting the Servos to the Pololu microcontroller are properly in place. If the Arm Servos are still losing power after this, you might have a faulty or dead power supply, or additionally a torn wire in one of the connection cables. There is also the possibility that the Servos or the Pololu themselves could be faulty or malfunctioning. Any of these components would need to be replaced in order to allow Harvey to function properly again.

3.) Camera not working properly or detecting any objects: If this occurs, be sure to double check and make sure that all of the wires connecting the Camera to the UpCore are connected in their respective ports or pins. If the Camera is still not working properly after this, you might have a torn wire in one of the connection cables. If it is working but not detecting objects as intended, there could be a possible malfunction with the vision recognition software aspect, instead of the camera hardware itself. In this case, try powering off Harvey entirely and power it back on, this should solve an issue like this the majority of the time. There is also the possibility that the UpCore or the Camera themselves could be faulty or malfunctioning. Any of these components would need to be replaced in order to allow Harvey to function properly again.

4.) Arm and/or gripper won’t move or function as intended: If this occurs, be sure to double check and make sure that all of the wires connecting the Arm to the Pololu microcontroller and the Arm Servos are connected in their respective ports or pins. If the Arm and/or gripper are still not moving or functioning properly after this, you might have a torn wire in one of the connection cables. There is also the possibility that the Servos or the Arm themselves could be faulty or malfunctioning. Any of these components would need to be replaced in order to allow Harvey to function properly again.

5.) Harvey becomes unresponsive during operation: If this occurs, be sure to double check and make sure the power supply is properly charged, and that all of the wires connecting to all of Harvey’s hardware components are connected in their respective ports or pins. If the rover is still unresponsive after this, you might have a faulty or dead power supply, or additionally a torn wire in one of the connection cables. Any of these components would need to be replaced in order to allow Harvey to function properly again.

6.) Additional questions or concerns about Harvey: If you have any other questions or concerns about project Harvey that are not listed here, please feel free to contact one of the 4 members of team Harvey. They are: Jacob Pitzer (Team Leader), Barry Huey, Mounir Barjaoui, and Jaqua Starr. The team members can be reached via their California University of PA email addresses, which are accessible upon request from the universities’ academic staff members.

**Expansion of Features: Adding New Functionality to Harvey**

This section provides instructions for if you would have any interest in adding additional features or functionality to Harvey. There are two main ways of adding new features to Harvey, those are hardware and software based aspects. The following will provide further detailed explanations on adding new features for each of these 2 component categories.

Adding new Hardware Based Features: In order to add new hardware based features and functionality to Harvey, you would need to first purchase the hardware components that you are interested in adding to the platform. Once you have acquired those parts, you will need to connect them to Harvey. Depending on what the type of component is, it would vary where you would need to attach the new parts to the rover. The central hub of Harvey’s hardware is its mobile chassis, this is most likely where you would want to attach the new parts.

The other major section that new components could be attached to would be the articulated arm. For example, if you wanted to enable the arm to rotate at 360 degrees instead of being stationary, a rotating base could be attached to the chassis, which then the arm could be mounted onto. Any additional components would also need to be connected with the proper wiring to either the Intel UpCore board or the Pololu microcontroller, depending on the type of component it is. Additional wiring would also need to be purchased in order to connect the hardware together properly.

Adding new Software Based Features: In order to add new software based features and functionality to Harvey, you would need to have access to a personal computer and have a C++ compatible compiler installed onto it that is Linux friendly. A Windows based compiler such as Microsoft Visual Studio would not work for Harvey, a program such as Code Blocks or Crimson Editor would be perfectly capable of this task instead. C++ is the programming language that Harvey’s source code is written in, so no other programming language would be usable to add new software based features to the rover. It goes to say that you would also need to have knowledge of and some experience in computer programming (specifically in C++) in order to be able to write the proper source code for adding new software features.

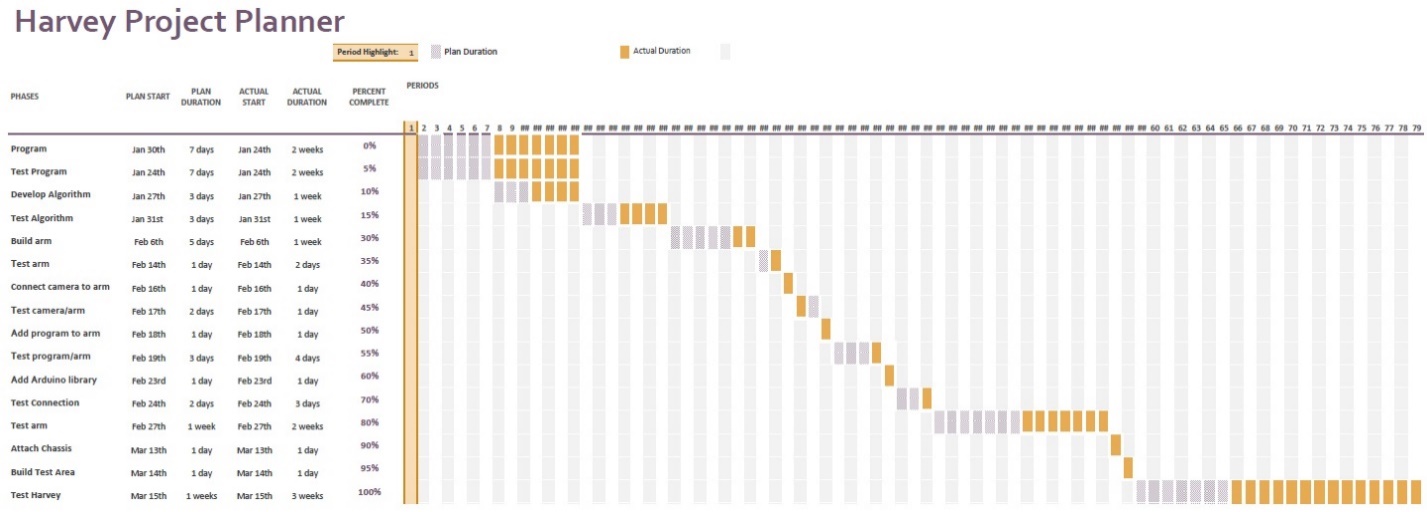
The final preparation step before being able to write new source code to Harvey, is to download the already existing code that comes stored onto the Intel UpCore board onto your personal computer. This can be done by connecting the included USB cable that comes with Harvey into both the UpCore and your personal computer. You should then be able to copy the source code from the UpCore onto your computer through the Windows Explorer program for Windows users, or the File Explorer program for Linux users. With all of these preparations made in advance, you can now begin to add new features to Harvey’s core existing source code.

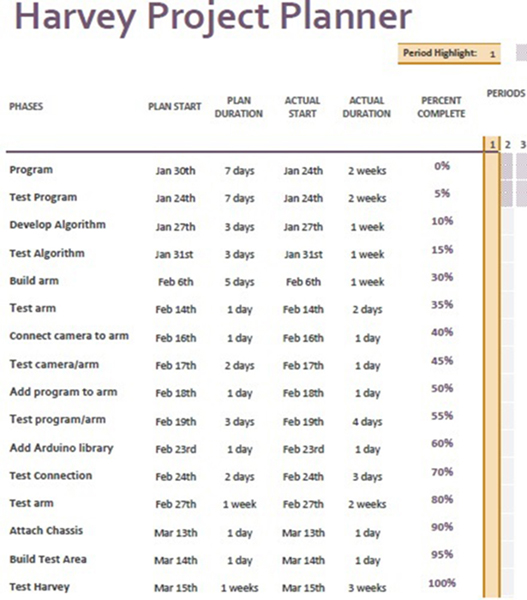
In order to properly add new software functionality to Harvey’s source code, you would want to first create a new project in your compiler of choice and add all of the existing files that you downloaded from the UpCore into the project. Then you would write new class types and their respective modules into either the existing files, or create new additional “.cpp” and “.h” files based on the type of features you would want to add to the platform. The “main” file included with the source code should be kept as the sole front end file for the source code and no additional “main” files should be created, only expanded upon the original one. Additionally, you could also add new features to the already existing source code by expanding on the current modules and classes that are included with Harvey by default.

After you have finished adding the new source code, you would need to compile all of the code in the entire project, and debug for any compiler errors that might occur. Once you have finished debugging and the code compiles successfully, double check your new code to make sure you did not make any logical based errors with the structuring of your code. These are not as easy to catch as syntax based errors, and the compiler will not check for those types of errors.

Once you have finished doing all of these steps, you should now be ready to transfer all of the source code files back to the UpCore by copying them back from your computer and over writing the ones already on the UpCore. Make sure to create a backup on your computer of the original source code that comes included with Harvey, in case the new code does not function properly or as intended. From this point, you should be able to power up and operate the rover as per usual, making sure the respective additional hardware is in place before hand if needed.

**Project Management: Gantt Chart Evaluation**



  
This is the original Gantt chart that was created to provide optimal management of the workflow process for project Harvey, preceding the start of the implementation phase for the project.

Throughout the course of project Harvey’s implementation phase, there were some set-backs that changed the way the project was fully developed. The original goals and objectives for the project that were laid out in the Gantt chart were still met in the end, but the process in which those goals and objectives were completed was not exactly as the team originally intended.

One of the set-backs the team encountered early on was not having access to the senior project lab to work on the development of the project. This made it cumbersome to meet as a group with the included hardware without any secure location to store the rover on campus. The team worked around this issue by taking turns borrowing the respective hardware when not able to meet as a group.

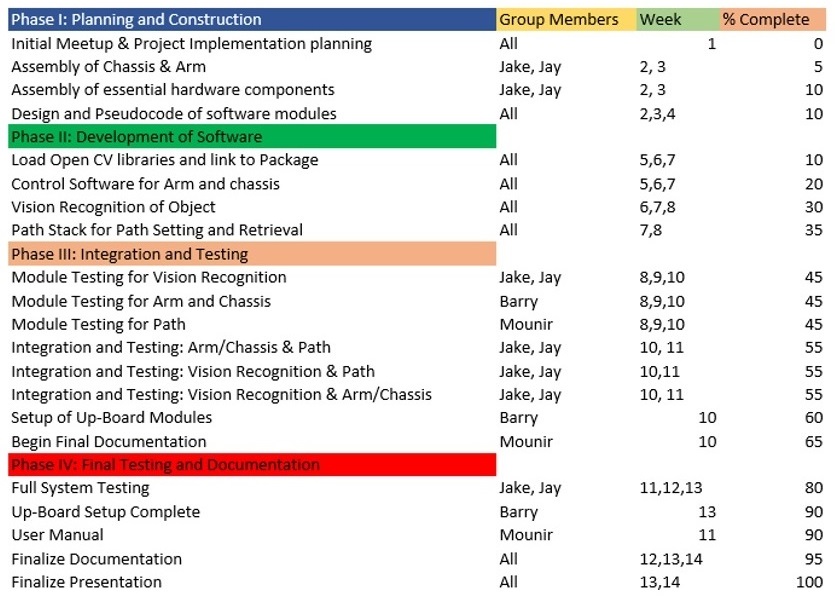
Another set-back was that the original Intel IoT board was fried due to having too high of a voltage sent to the device from a power source. This prevented work from being done with the majority of the hardware for the Harvey platform since the Intel IoT board is main control unit that contains the source code for the project. The team resolved this issue by ordering another Intel board which got the team back on track in this regard.

The other set-back was the switch from the originally planned Intel based software for the vision recognition part of the project to using OpenCV instead as the main software to implement and test the vision recognition. This switch was due to the Intel software not providing the level of effectiveness that was desired in order to complete the project as intended. While this did require some time to learn how to use OpenCV properly, it did not prevent the project from continuing. This issue was resolved and the vision recognition is fully functional with OpenCV for the Harvey platform.

These set-backs resulted in the shifting of the original planned phases within the Gantt chart. This changed how the team went about completing the project and led astray from the original planned implementation workflow that was laid out by the Gantt chart. The biggest change was shifting the writing of the source code for the project from the early phases of the implementation workflow to the much later stages, near the mid to late end of the project development.

This outcome was not in a negative light though, as it gave the team members a new and positive resolve on how the project still succeeded in the end despite having several set-backs along the way. The Gantt chart was still a very useful tool in assisting with keeping the team members on track for the workflow process of the project’s implementation and testing phases. The team members of project Harvey would consider this to be a good learning experience for handling set-backs and workflow management in future projects.

Additionally, this is the revised Gantt chart that team Harvey created towards the end of project development:



**Results/Conclusions: Assessment of Objectives**

As of the time of this writing, project Harvey has finished its final testing phase. The rover has successfully completed all of the objectives it had been assigned at the beginning of the project. All of the modules for Harvey’s source code have been completed, integrated together into a fully functioning system, and loaded into the Intel Up-board in order to interface with Harvey’s hardware components. All of Harvey’s hardware was successfully assembled together and is fully functional. Once both the software and hardware components of Harvey were completed and integrated together, the rover was thoroughly tested to ensure there wasn’t any errors or issues with either its software or hardware.

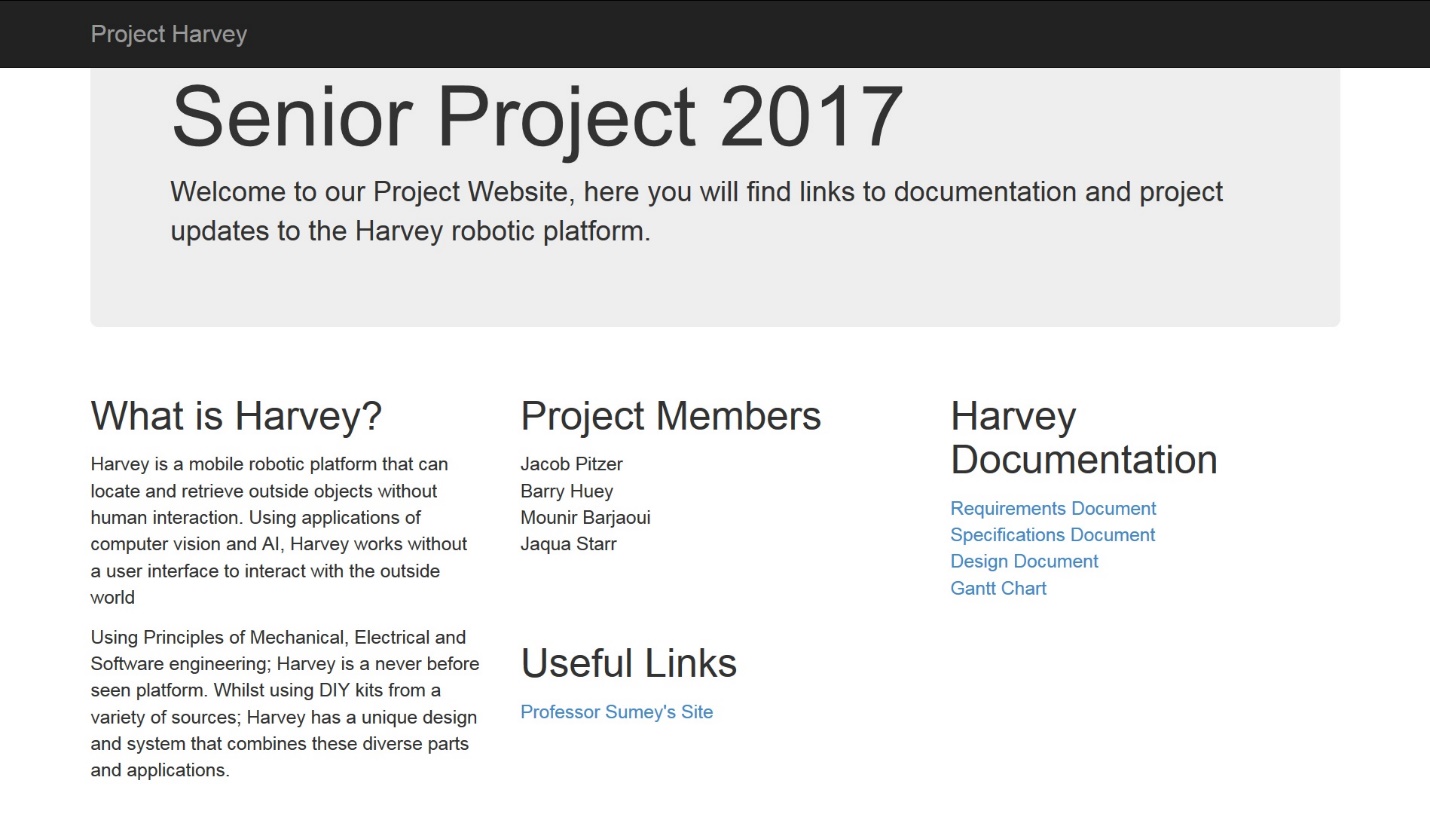
Further details regarding the completion of project Harvey’s objectives can be broken down into the main 3 key concepts behind Harvey’s design. 1.) The rover’s vision recognition component was successful in capturing the proper objects that were placed in the square testing area. OpenCV was properly implemented in order to create the classifiers used for the objects Harvey was to locate and pickup. While this process was successful, certain objects proved to be unideal for object retrieval.

2.) The object gripper component of the articulated arm was successful in reaching for, picking up, holding, and putting down the objects targeted by Harvey’s camera and the vision recognition components. The control algorithm for the articulated arm was successfully integrated with the hardware component of the arm itself in order to make this possible. While the process of picking up objects by the arm was successful, there were certain objects that proved difficult for the gripper to grasp properly and not ideal for object retrieval.

3.) The path finding component for the chassis was successful in moving Harvey both to and from the proper location of the objects that were targeted in its sight by the camera hardware and vision recognition software components. The character stack which was created to direct and keep track of Harvey’s movements integrated successfully with the path finding algorithm, which allowed the rover’s movements to be completely autonomous.

Additionally, the team realized during the later portions of the project development that the original Gantt chart was inaccurate regarding the time frames in which the project phases took place. Towards the end of the project, the team created a new, revised Gantt chart to better reflect the fully accurate project development cycle. Despite some minor set-backs during the development process, the team was able to create the proper solutions needed to solve these issues. Overall project Harvey has become an outstanding success. The rover was able to complete the objectives that it was originally given at its onset, and there is a very high chance that the prototype model can be used as a basis for a larger, up-scaled model in the future.

**Appendix I: Project Website**

****

**Project Harvey Website URL:** [**http://makeseniorprojectgreatagain.azurewebsites.net/**](http://makeseniorprojectgreatagain.azurewebsites.net/)

This is the Project Harvey team website. The screen capture shows the project’s homepage, which contains a brief description about the project, the team member’s names, and several other links. From there you can navigate to the previous documentation created for the project, such as the Requirements, Specifications, and Design documents, along with the Gantt chart.

These documents all precede the time of writing for this Implementation document, which unlike the previous documents, records the efforts of the project Harvey team after the source code and hardware for the project was fully implemented and integrated. Therefore, the previous documents may not accurately reflect the content included in this document, and some content for this document has changed from what was originally established for the project.

**Appendix III: Test Modules**

The following are the test modules for microcontroller and path classes. These modules were included because of the large part of these modules played in development, integration and test. The vision recognition module was not included because the module itself was tested as is and did not require a test module. The main module was tested in conjunction with all modules, and the final version of the main class is a test module of the entire system.

**Path: Main Test Module**

/\* Main program for path class, main program will be implemented into Harvey main software package

Program simulates one pass without finding an object. The program makes an additional pass before

switching to found algorithm. Tests functionality of path class in its entirety.

\*/

#include <iostream>

#include <unistd.h>

#include "path.h"

using namespace std;

int main(void) {

Path rbtpath;

char forward = 'f';

char backward = 'b';

char right = 'r';

char left = 'l';

char direction;

int pass = 0;

int step = 0;

int edge = 10;

int leg = 0;

bool found = false;

bool flag = false;

while(!flag)

{

//function for reading video frames will go here

if(!found)

{

if (step < edge)

{

rbtpath.push(forward);

cout << "Pushed directional command: " << forward << endl;

step++;

cout << "Step: " << step << endl;

sleep(1);

}

else if (step == edge && (leg == 0 || leg == 2))

{

leg++;

edge += 10;

rbtpath.push(right);

cout << "Pushed directional command: " << right << endl;

step++;

cout << "Step: " << step << endl;

sleep(1);

}

else if (step == edge && (leg == 1 || leg == 3))

{

leg++;

edge += 10;

rbtpath.push(left);

cout << "Pushed directional command: " << left << endl;

step++;

cout << "Step: " << step << endl;

sleep(1);

}

else if (step == edge && leg == 4)

{

cout << "Pass complete, returning to base" << endl;

pass++; //increment pass counter

if (pass == 2)

{

found = true; //test code to exit loop once one pass is completed

}

else

{

rbtpath.~Path(); //destroy path stack

//function for turning the robot around will go here

step = 0;

leg = 0;

edge = 10; //reset path setting values

}

}

else

{

cout << "Unable to push command " << endl;

}

}

if(found)

{

cout << "Object detected " << endl;

cout << "Retrieving Object " << endl;

cout << "Returning back to home " << endl;

while(rbtpath.getPathSize() > 0)

{

direction = rbtpath.pop(); //receive direction command from path stack

switch (direction)

{

case 'f':

cout << "received directional command: " << direction << endl;

//function to move robot forward will go here

cout << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'l':

cout << "Recieved directional command: " << direction << endl;

//function to move robot right will go here

cout << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'r':

cout << "Received directional command: " << direction << endl;

//function to move robot right will go here

cout << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'b':

cout << "Recieved directional command: " << direction << endl;

//function to move the robot backwards will go here

cout << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

default:

cout << "Error: could not determine retrieval command" << endl;

break;

}

}

found = false; //reset found flag

if (pass == 2) //test code block not to be integrated into main program

{

flag = true;

}

}

} //simulate video loop for main program

return 0;

}

**Microcontroller: Main Test Module**

/\*

Main program for control of Pololu Maestro servo controller. Class and Functions to be integrated into Robotic platform

\*/

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <iostream>

#include "microcontroller.h"

#ifdef \_WIN32

#define O\_NOCTTY 0

#else

#include <termios.h>

#endif

using namespace std;

int main()

{

microcontroller mcroctr; //initialize the microcontroller class

const char \* device = "\\\\.\\COM7";

int fd = open(device, O\_RDWR | O\_NOCTTY);

if(fd == -1)

{

perror(device);

return 1;

}

#ifdef \_WIN32

\_setmode(fd, \_O\_BINARY);

#else

struct termios options;

tcgetattr(fd, &options);

options.c\_iflag &= ~(INCR | IGNCR | ICRNL | IXON | IXOFF);

options.c\_oflag &= ~(ONLCR | OCRNL);

options.c\_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);

tcsetattr(fd, TCSANOW, &options);

#endif

int position = mcroctr.getPosition(fd, 0);

cout << "Base position is " << position << " " << position/4 << endl;

position = mcroctr.getPosition(fd, 2);

cout << "Wrist position is " << position << " " << position/4 << endl;

position = mcroctr.getPosition(fd, 3);

cout << "Forearm position is " << position << " " << position/4 << endl;

mcroctr.moveForward(fd);

sleep(2.5);

mcroctr.moveBase(fd, 6000);

mcroctr.setTarget(fd, 2, 6000);

mcroctr.openGripper(fd);

sleep(2);

mcroctr.moveBase(fd, 4000);

sleep(0.75);

mcroctr.setTarget(fd, 2, 5200);

sleep(2);

mcroctr.closeGripper(fd);

sleep(2);

mcroctr.moveBase(fd, 5000);

sleep(2);

mcroctr.moveBase(fd, 4000);

sleep(0.75);

mcroctr.closeGripper(fd);

mcroctr.moveRight(fd);

sleep(0.25);

mcroctr.moveRight(fd);

mcroctr.moveForward(fd);

sleep(2.5);

close(fd);

return 0;

}

**Appendix IV: Code Lisitngs**

**Microcontoller.h**

/\* CLASS SUMMARY: MICROCONTROLLER.H

\* Microcontroller.h defines all hardware control for the arm

\* and the drive system. The class conects with the Pololu

\* microcontroller via serial port and sends pulse commands

\* to appropriate channels. Class defines servo movements and

\* action commands for quick and simple movement and object

\* retrieval.

\*

\* FUNCTION DEFINITIONS:

\* microcontroller() : DEFAULT CONSTRUCTOR

\* PRE-CONDITIONS: NONE

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Default constructor for microcontroller class. Class defines intitial private

\* data for device, position, and target.

\*

\* getPosition() : RETURNS SERVO POSITION

\* PRE-CONDITIONS: INT, UNSIGNED CHAR

\* POST-CONDITIONS: INT

\* DESCRIPTION:

\* Function that accepts device, and servo channel. Function returns servo position

\* of given channel and device and returns and integer of the given position

\*

\* setTarget() : SETS SERVO TARGET

\* PRE-CONDITIONS: INT, UNSIGNED CHAR, UNSIGNED SHORT

\* POST-CONDITIONS: INT

\* DESCRIPTION:

\* Function that sets servo position. Accepts device, channel, and target

\* respectively and sends commands to the microcontroller. If there is an error,

\* returns -1 back to calling module, if successful returns 0.

\*

\* moveForward() : MOVES ROVER FORWARD

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function accepts the device and activates drive motors to move the rover forward for a half-second

\* motor values are returned to nothing after function call

\*

\* moveBackward() : MOVES ROVER BACKWARD

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function accepts the device and activates drive motors to move rover backward for half-second,

\* motor values are returned to null after delay finishes

\*

\* moveRight() : MOVES ROVER TO THE RIGHT

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Moves the rover 90 degrees to the right. Function ceases after delay finishes

\*

\* moveLeft() : MOVES ROVER TO THE LEFT

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Moves the rover 90 degrees to the left. Function ceases after delay finishes

\*

\* moveBase() : MOVES BASE OF ARM

\* PRE-CONDITIONS: INT, INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function accepts device and target, and moves the base of the arm to the

\* desired location. If base is already at target location it will print

\* message to console

\*

\* moveForearm() : MOVES FOREARM OF ARM

\* PRE-CONDITIONS: INT, INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function accepts device and target, and moves the forearm to the desired

\* location. If forearm is already at target, function will print to console

\*

\* moveWrist() : MOVES WRIST OF ARM

\* PRE-CONDITIONS: INT, INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function accepts device and target, and moves the wrist to the desired

\* location. If the wrist is already at target, function will print to console

\*

\* openGripper() : OPENS THE GRIPPER

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function opens the gripper to the open position. ceases once gripper opens

\*

\* closeGripper() : CLOSES THE GRIPPER

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function closes the gripper to the closed position. ceases once gripper

\* is closed.

\*

\* setInitialPosition() : SETS ARM FOR INITIAL TARGET POSITION

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function sets arm in initial position during search and retrieval

\*

\* grabObject() : CAPTURES OBJECT FROM GROUND

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Moves the arm to capture object. base moves down and gripper encases

\* target object

\*

\* retrieveObject() : HOLDS OBJECT FOR RETRIEVAL

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* moves the object off the ground in a steady position for retrieval

\*

\* dropObject() : PLACES OBJECT ON GROUND

\* PRE-CONDITIONS: INT

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Places object on the ground and returns the arm to the initial position

\*

\*

\*

\* microcontroller.h

\*

\* Created on: Mar 29, 2018

\*

\*/

#ifndef MICROCONTROLLER\_H\_

#define MICROCONTROLLER\_H\_

class microcontroller

{

public:

microcontroller();

int getPosition(int, unsigned char);

int setTarget(int, unsigned char, unsigned short);

void moveForward(int);

void moveBackward(int);

void moveLeft(int);

void moveRight(int);

void moveBase(int, int);

void moveForearm(int, int);

void moveWrist(int, int);

void openGripper(int);

void closeGripper(int);

void setInitialPosition(int);

void grabObject(int);

void retrieveObject(int);

void placeObject(int);

private:

int fd;

int position;

int target;

};

#endif /\* MICROCONTROLLER\_H\_ \*/

**Microcontroller.cpp**

/\*

\* microcontroller.cpp

\*

\* Created on: Mar 29, 2018

\* Author: Supreme Leader Jake

\*/

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <iostream>

#include "microcontroller.h"

using namespace std;

microcontroller::microcontroller() //default constructor

{

fd = 0;

position = 0;

target = 0;

}

int microcontroller::getPosition(int fd, unsigned char channel) //function to get servo target

{

unsigned char command[] = {0x90, channel}; //convert to channel

if(write(fd,command, sizeof(command)) == -1)

{

perror("Error");

return -1;

} //print error

unsigned char response[2];

if (read(fd,response,2) != 2)

{

perror("Error");

return -1;

} //print error

return response[0] + 256\*response[1]; //return posiiton

}

int microcontroller::setTarget(int fd, unsigned char channel, unsigned short target) //function to set servo target

{

unsigned char command[] = {0x84, channel, target & 0x7F, target >> 7 & 0x7F}; //pass device, channel, and target

if (write(fd, command, sizeof(command)) == -1)

{

perror("Error");

return -1;

} //print error

return 0; //return if successful

}

void microcontroller::moveForward(int fd) //function to move forward

{

setTarget(fd, 4, 7200);

setTarget(fd, 5, 4800);

sleep(0.5);

}

void microcontroller::moveBackward(int fd) //function to move backward

{

setTarget(fd, 4, 4800);

setTarget(fd, 5, 7200);

sleep(0.5);

}

void microcontroller::moveRight(int fd) //function to move right

{

setTarget(fd, 4, 4800);

sleep(0.25);

}

void microcontroller::moveLeft(int fd) //function to move left

{

setTarget(fd, 5, 7200);

sleep(0.25);

}

void microcontroller::moveBase(int fd, int target) //member function to move arm base up or down

{

int position = getPosition(fd, 0); //receives servo position from control board

if (target == position) //checks if servo is already at target position

{

cout << "Base already at desired position" << endl;

}

else if (target > position) //move the base up if target is greater than current position

{

for (int i = position; i < target; i++)

{

setTarget(fd, 0, i);

sleep(0.75);

}

}

else if (target < position) //move base down if target is less than current position

{

for (int j = position; j > target; j--)

{

setTarget(fd, 0, j);

sleep(0.75);

}

}

else //print error message to console if a failure occurs

{

cout << "Error: Could not move to position" << endl;

}

}

void microcontroller::moveForearm(int fd, int target) //member function for forearm servo movement

{

int position = getPosition(fd, 3); //get forearm servo position

if (target == position)

{

cout << "Forearm is already at target position" << endl;

}

else if (target > position) //if forearm target is greater than current position, move servo up

{

for (int i = position; i < target; i++)

{

setTarget(fd, 3, i);

sleep(0.75);

}

}

else if (target < position) //if target is less than current position, move servo down

{

for (int j = position; j > target; j-- )

{

setTarget(fd, 3, j);

sleep(0.75);

}

}

else //if unable to communicate with servo, print error message to console

{

cout << "Error: could not move to target position" << endl;

}

}

void microcontroller::moveWrist(int fd, int target) //member function for wrist servo movement

{

int position = getPosition(fd, 2);

if (target == position) //print to console if servo is presently at target location

{

cout << "Servo located in target position" << endl;

}

else if (target > position) //if target is greater than position, move wrist servo up

{

for (int i = position; i < target; i++)

{

setTarget(fd, 2, i);

sleep(0.75);

}

}

else if (target < position) //if target is less than position, move wrist servo down

{

for (int j = position; j > position; j--)

{

setTarget(fd, 2, j);

sleep(0.75);

}

}

else //print error message to console if failure occurs

{

cout << "Error: could not communicate with wrist servo" << endl;

}

}

void microcontroller::openGripper(int fd) //function to open gripper

{

setTarget(fd, 1, 8000);

}

void microcontroller::closeGripper(int fd) //function to close gripper

{

setTarget(fd, 1, 6000);

}

void microcontroller::setInitialPosition(int fd) //function to set initial position

{

moveWrist(fd, 6000);

sleep(0.75);

moveForearm(fd, 7000);

sleep(0.75);

moveBase(fd, 6000);

sleep(0.75);

openGripper(fd);

sleep(0.75);

}

void microcontroller::grabObject(int fd) //function to grab object

{

moveBase(fd, 4000);

sleep(0.75);

moveWrist(fd, 5000);

sleep(0.75);

moveForearm(fd, 7000);

sleep(0.75);

}

void microcontroller::retrieveObject(int fd) //function to retrieve object

{

closeGripper(fd);

sleep(0.75);

moveBase(fd, 5000);

sleep(0.75);

}

void microcontroller::placeObject(int fd) //function to place object

{

moveBase(fd, 4000);

sleep(0.75);

openGripper(fd);

sleep(0.75);

setInitialPosition(fd);

}

/\*

\* PATHFINDER ROBOTIC PLATFORM CONTROL SOFTWARE

\* VERSION 1.0.1

\*

\* DESCRIPTION OF PACKAGE

\* The following files consist of the proof of concept of PATHFINDER robotic

\* platform. This software controls the microcontroller and vision recognition

\* system of the robotic platform. Use of previous defined functions from OPENCV

\* and POLOLU. All credit for those packages is given to them respectively.

\*

\* THIS SOFTWARE IS FREE FOR GENERAL USE BY BOTH INDIVIDUAL AND COMMERCIAL USES

\* ALL USE AND CREDIT MUST BE CITED TO JACOB PITZER, JAQUA STARR, BARRY HUEY

\* MOUNIR BARJAOUI, OPENCV, AND POLOLU FOR ANY USE

\*

\* main.cpp

\*

\* Created on: Feb 28, 2018

\*

\*/

//OpenCV and Harvey Object libraries

#include "opencv2/objdetect.hpp"

**Main.cpp**

#include "opencv2/highgui.hpp"

#include "opencv2/imgproc.hpp"

#include "visionrecognition.h"

#include "path.h"

#include "microcontroller.h"

//C/C++ Native libraries

#include <stdio.h>

#include <iostream>

#include <unistd.h>

#include <termios.h>

#include <fcntl.h>

//std and OpenCV namespaces

using namespace std;

using namespace cv;

/\*\* @function main \*/

int main(int argc, const char\*\* argv)

{

//object declarations

visionrecognition vrs;

Path rbtpath;

VideoCapture capture;

microcontroller mcrctr;

Mat frame;

//main variable declarations

String window\_name = "Harvey Video Feed";

bool found = false;

int leg = 0;

int edge = 10;

int step = 0;

int pass = 1;

int fd;

char forward = 'f';

char right = 'r';

char left = 'l';

char backward = 'b';

char direction;

const char \* device = "/dev/ttyACM0";

fd = open(device, O\_RDWR | O\_NOCTTY); //open Pololu Maestro microcontroller

if(fd == -1)

{

perror(device);

cout << "Error opening microcontroller" << endl;

return 1;

} //return error if device fails to open

struct termios options; //struct and termios options for serial port communication

tcgetattr(fd, &options);

options.c\_iflag &= ~(INLCR | IGNCR | ICRNL | IXON | IXOFF);

options.c\_oflag &= ~(ONLCR | OCRNL);

options.c\_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);

tcsetattr(fd, TCSANOW, &options);

vrs.getFaceCascade(argc, argv); //load cascade files into main String variables

vrs.getPlateCascade(argc, argv);

vrs.getCatCascade(argc, argv);

//initialize camera feed

capture.open(0);

if (!capture.isOpened())

{

cout << "Could not initialize Camera, Sorry m8";

return -1;

}

while (capture.read(frame)) //infinite loop for video feed

{

if(frame.empty())

{

cout << "Error -- No frame detected you scallywag";

break;

}

found = vrs.detectObj(frame, found, window\_name); //load classifier into frame, function will mark found = true if object detected

if(!found)

{

cout << "No objects detected" << endl;

cout << "Advancing Rover " << endl;

if (step < edge) //if not at edge & object !found, move forward

{

rbtpath.push(forward);

cout << "pushed directional command: " << forward << endl;

mcrctr.moveForward(fd); //move rover forward

step++;

cout << "Step: " << step << endl;

}

else if (step == edge && (leg == 0 || leg == 2))

{

leg++;

edge += 10;

rbtpath.push(right);

cout << "Pushed directional command" << right << endl;

mcrctr.moveRight(fd); //function to move rover right

step++;

cout << "Step: " << step << endl;

}

else if (step == edge && (leg == 1 || leg == 3))

{

leg++;

edge += 10;

rbtpath.push(left);

cout << "Pushed directional command: " << left << endl;

mcrctr.moveLeft(fd); //function to move rover left

step++;

cout << "Step: " << step << endl;

}

else if (step == edge && leg == 4)

{

pass++;

cout << "Pass complete. Now moving to pass: " << pass << endl;

rbtpath.~Path(); //destroy path stack

step = 0;

leg = 0;

edge = 10; //reset path setting variables

mcrctr.moveRight(fd);

sleep(0.25);

mcrctr.moveRight(fd);

}

else

cout << "Unable to push command" << endl; //print to console if error occurs

}

else if (found)

{

cout << "object detected" << endl;

mcrctr.grabObject(fd);

mcrctr.retrieveObject(fd); //grab and retrieve sequence

cout << "Object retrieved" << endl;

cout << "Returning back to home" << endl;

mcrctr.moveRight(fd);

sleep(0.25);

mcrctr.moveRight(fd); //pivot rover to turn around

while(rbtpath.getPathSize() > 0)

{

direction = rbtpath.pop(); //receive directional node from path stack

switch(direction)

{

case 'f':

cout << "Received directional command: " << direction << endl;

mcrctr.moveForward(fd);

cout << "Current path size: " << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'l':

cout << "Received directional command: " << direction << " Moving right" << endl;

mcrctr.moveRight(fd);

cout << "Current path size: " << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'r':

cout << "Received directional command" << direction << " Moving left" << endl;

mcrctr.moveLeft(fd);

cout << "Current path size: " << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

case 'b':

cout << "Received directional command" << direction << endl;

mcrctr.moveBackward(fd);

cout << rbtpath.getPathSize() << endl;

step--;

cout << "Step: " << step << endl;

sleep(1);

break;

default:

cout << "Error: could not determine return node" << endl;

break;

}

}

mcrctr.moveRight(fd);

sleep(0.25);

mcrctr.moveRight(fd); //pivot rover to turn around

found = false; //reset found variable

}

if(waitKey(10) == 27) //break loop if there is a 10 second delay or hotkey is pressed

{

break;

}

else if(pass == 3) //if passes reach 3, end program

{

break;

}

}

return 0;

}

**Path.h**

/\*

CLASS SUMMARY: PATH.H

Path.h defines a stack of characters that can be of any size.

The sole purpose of the stack is to track robot movements in

the field of operation. while the robot is seaching for objects

the main file will push directions moved unto the stack. Once an

object is found or the robot reaches the end of a search path,

values will be popped off until the stack is empty

FUNCTION DEFINITIONS:

Path() DEFAULT CONSTRUCTOR:

PRE-CONDITIONS: NONE

POST-CONDITIONS: NONE

DESCRIPTION:

Default constructor for Path.h. Initializes path size to 0 once

program begins.

~Path() DEFAULT DESTRUCTOR:

PRE-CONDITIONS: NONE

POST-CONDITIONS: NONE

DESCRIPTION:

Default destructor for path.h. At the end of the program, main will

call destructor to destroy anything remaining of the stack to free

memory. Called when the robot makes a pass and does not find any

objects.

push() PUSH ON STACK FUNCTION:

PRE-CONDITIONS: CHAR

POST-CONDITIONS: NONE

DESCRIPTION:

Push function for path.h. Function will take a char input and push it

onto the stack. Function will also increment the stack size counter,

initialized by the constructor

pop() POP OFF STACK FUNCTION:

PRE-CONDITIONS: NONE

POST-CONDITIONS: CHAR

DESCRIPTION:

Pop function for path.h. Function will pop off the top character from

the stack and return it to the main program. It will decrement the

stack size counter.

getPathSize() STACK SIZE COUNTER:

PRE-CONDITIONS: NONE

POST-CONDITIONS: INT

DESCRIPTION:

Returns the stack size whenever called. Either from main or from the

path class

\* path.h

\*

\* Created on: Apr 11, 2018

\*

\*/

#ifndef PATH\_H\_

#define PATH\_H\_

class Path

{

public:

Path(); //default constructor for Path class

~Path(); //default destructor for Path class

void push(char); //pushes a direction value onto the stack

int getPathSize(); //returns path size

char pop(); //pops a directional value back to the main program

private:

struct Node { //struct data type for Node

char direc;

Node \*next;

};

struct Node \*top; //pointer for top of the stack

int pathsize; //counter for path stack size

};

#endif /\* PATH\_H\_ \*/

**Path.cpp**

/\*

\* path.cpp

\*

\* Created on: Apr 11, 2018

\*

\*/

#include <iostream>

#include "path.h"

using namespace std;

Path::Path()

{

pathsize = 0;

}

Path::~Path()

{

Node \*temp;

char tempchar;

while(pathsize != 0)

{

if (pathsize != 0)

{

temp = top;

top = top -> next;

tempchar = temp -> direc;

delete temp;

temp = 0;

pathsize --;

}

else if (pathsize == 1)

{

temp = top;

top = top -> next;

tempchar = temp -> direc;

delete temp;

temp = 0;

top = NULL;

delete top;

pathsize --;

}

}

}

void Path::push(char data) //member function for pushing directional node onto the path stack

{

Node \*entry;

entry = new Node; //create an instance of a new node

entry -> direc = data;

entry -> next = top;

top = entry;

pathsize++;

}

char Path::pop() //member function for popping directional nodes off the path stack

{

Node \*temp;

char currdirec; //value for direction data popped off stack

if (pathsize != 0)

{

temp = top;

top = top -> next;

currdirec = temp -> direc;

delete temp;

temp = 0;

pathsize --;

return currdirec;

}

else if (pathsize == 1)

{

temp = top;

top = top -> next;

currdirec = temp -> direc;

delete temp;

temp = 0;

top = NULL;

delete top;

pathsize --;

}

else

cout << "Path stack is empty bruh" << endl;

}

int Path::getPathSize()

{

return pathsize;

}

**Visionrecognition.h**

/\* VISION RECOGNITION

\* CLASS DEFINITION:

\* This class defines the attributes and functions for VISION RECOGNITION

\* Performs all cascade file loading, object detection, and drawing targets

\* on the video screen

\*

\* FUNCTION DEFINITIONS:

\*

\* visionrecognition() : DEFAULT CONSTRUCTOR

\* PRE-CONDITIONS: NONE

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* default constructor for vision recognition class. Initializes

\* Cascade classifiers and string variables

\*

\* detectObj() : FUNCTION FOR OBJECT RECOGNITION

\* PRE-CONDITIONS: MAT, BOOLEAN, STRING

\* POST-CONDITIONS: BOOLEAN

\* DESCRIPTION:

\* Function for object detection. This function accepts a Mat object,

\* a boolean, and a String. Uses CascadeClassifier definitions to

\* scan image frame. If an object is found, function will place a rectangle

\* around the center and return true. else, it will remain false and return

\* false

\*

\* getFaceCascade() : FUNCTION FOR LOADING FACE CLASSIFIER

\* PRE-CONDITIONS: INT, CONST CHAR

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function for loading face classifier file. If successful, it will

\* set the string to the face classifier file, if failure it will print

\* a error message to the screen

\*

\* getPlateCascade() : FUNCTION FOR LOADING PLATE CLASSIFIER

\* PRE-CONDITIONS: INT, CONST CHAR

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function for loading the plate classifier file. If successful, it will

\* set the string to the plate classifier file. If there is a failure,

\* a message will be printed to the console

\*

\* getCatCascade() : FUNCTION FOR LOADING PLATE CLASSIFIER

\* PRE-CONDITIONS: INT, CONST CHAR

\* POST-CONDITIONS: NONE

\* DESCRIPTION:

\* Function for loading the cat classifier file. If successful, it will

\* set the string to the plate classiifer file. If there is a failure, a m

\* message will be printed to the console.

\* visionrecognition.h

\*

\* Created on: Feb 28, 2018

\*

\*/

#ifndef VISIONRECOGNITION\_H

#define VISIONRECOGNITION\_H

#include "opencv2/objdetect.hpp"

#include "opencv2/highgui.hpp"

#include "opencv2/imgproc.hpp"

#include <stdio.h>

#include <iostream>

using namespace std;

using namespace cv;

class visionrecognition {

public:

visionrecognition(); //default constructor for vision recognition class

bool detectObj(Mat, bool, String);

void getFaceCascade(int, const char \*\*);

void getPlateCascade(int, const char \*\*);

void getCatCascade(int, const char \*\*);

private:

String face\_cascade\_name;

String cat\_cascade\_name;

String plate\_cascade\_name;

CascadeClassifier face\_cascade;

CascadeClassifier plate\_cascade;

CascadeClassifier cat\_cascade;

};

#endif /\* VISIONRECOGNITION\_H \*/

**Visionrecognition.cpp**

/\*

\* visionrecognition.cpp

\*

\* Created on: Feb 28, 2018

\*

\*

\*/

#include "opencv2/objdetect.hpp"

#include "opencv2/highgui.hpp"

#include "opencv2/imgproc.hpp"

#include "visionrecognition.h"

#include <stdio.h>

#include <iostream>

using namespace std;

using namespace cv;

visionrecognition::visionrecognition()

{

String face\_cascade\_name = "";

String cat\_cascade\_name = "";

String plate\_cascade\_name = "";

CascadeClassifier face\_cascade;

}

bool visionrecognition::detectObj(Mat frame, bool found, String window\_name)

{

vector<Rect> faces; //create rectangle vectors for target images

vector<Rect> cats;

vector<Rect> plates;

Mat frame\_gray; //create greyframe mat image

const static Scalar colors[] =

{

Scalar(255,0,0),

Scalar(255,128,0),

Scalar(255,255,0),

Scalar(0,255,0),

Scalar(0,128,255),

Scalar(0,255,255),

Scalar(0,0,255),

Scalar(255,0,255)

}; //scalar array of frame color values

cvtColor( frame, frame\_gray, COLOR\_BGR2GRAY );

equalizeHist( frame\_gray, frame\_gray );

//Detect object functions

face\_cascade.detectMultiScale( frame\_gray, faces, 1.1, 2, 0|CASCADE\_SCALE\_IMAGE, Size(60, 60) );

cat\_cascade.detectMultiScale( frame\_gray, cats, 1.1, 2, 0|CASCADE\_SCALE\_IMAGE, Size(60, 60) );

plate\_cascade.detectMultiScale( frame\_gray, plates, 1.1, 2, 0|CASCADE\_SCALE\_IMAGE, Size(60, 60) );

for ( size\_t i = 0; i < faces.size(); i++ ) //if face detected, place rectangle over the center

{

Rect r = faces[i];

Point center;

Scalar color = colors[i%8]; //create scalar of colors

double scale = 1;

rectangle( frame, Point(cvRound(r.x\*scale), cvRound(r.y\*scale)),

Point(cvRound((r.x + r.width-1)\*scale), cvRound((r.y + r.height-1)\*scale)),

color, 3, 8, 0); //create the rectangle

Mat faceROI = frame\_gray( faces[i] );

found = true; //return true, object found

}

for ( size\_t j = 0; j < cats.size(); j++ ) //if cat detected, place a rectangle around it

{

Rect r2 = cats[j];

Point center2;

Scalar color2 = colors[j%8];

double scale2 = 1;

rectangle( frame, Point(cvRound(r2.x\*scale2), cvRound(r2.y\*scale2)),

Point(cvRound((r2.x + r2.width-1)\*scale2), cvRound((r2.y + r2.height-1)\*scale2)),

color2, 3, 8, 0); //create the rectangle

Mat catROI = frame\_gray( cats[j] );

found = true; //return true, object found

}

for ( size\_t k = 0; k < plates.size(); k++ ) //if plate found, place a rectangle around it

{

Rect r3 = plates[k];

Point center3;

Scalar color3 = colors[k%8];

double scale3 = 1;

rectangle( frame, Point(cvRound(r3.x\*scale3), cvRound(r3.y\*scale3)),

Point(cvRound((r3.x + r3.width-1)\*scale3), cvRound((r3.y + r3.height-1)\*scale3)),

color3, 3, 8, 0); //draw the rectangle

Mat plateROI = frame\_gray( plates[k] );

found = true; //return true, object found

}

//show the video feed

imshow(window\_name, frame);

return found;

}

void visionrecognition::getFaceCascade(int argc, const char \*\* argv) //load the face cascade file

{

CommandLineParser parser(argc, argv,

"{face\_cascade|/media/jake/NDrive/opencv/data/haarcascades/haarcascade\_frontalface\_alt.xml|}"); //get face cascade from openCV data folder

face\_cascade\_name = parser.get<String>("face\_cascade"); //set cascade to face\_cascade string

if (!face\_cascade.load(face\_cascade\_name)) //if fail to load, print error message to console

{

cout << "Failure to load face cascade" << endl;

}

}

void visionrecognition::getPlateCascade(int argc, const char \*\* argv) //load the plate cascade file

{

CommandLineParser parser(argc, argv,

"{plate\_cascade|/media/jake/NDrive/opencv/data/haarcascades/haarcascade\_russian\_plate\_number.xml");

plate\_cascade\_name = parser.get<String>("plate\_cascade");

if (!plate\_cascade.load(plate\_cascade\_name))

{

cout << "failure to load plate cascade" << endl;

}

}

void visionrecognition::getCatCascade(int argc, const char \*\* argv) //load the cat cascade file

{

CommandLineParser parser(argc, argv,

"{cat\_cascade|/media/jake/NDrive/opencv/data/haarcascades/haarcascade\_frontalcatface.xml");

cat\_cascade\_name = parser.get<String>("cat\_cascade");

if (!cat\_cascade.load(cat\_cascade\_name))

{

cout << "failure to load cat cascade" << endl;

}

}