Dancing Swarm of Robots

Final Report

Team 40
Client: Dr. Akhilesh Tyagi
Advisors: Dr. Akhilesh Tyagi, Dr. Diane Rover, Dr. Phillip Jones
Members: Abdalla Abdelrahman, Daniel Nikolic, Benjamin Schneider, Noah Thompson, Mason Walls, Cole Weitzel
sdmay21-40@iastate.edu
https://sdmay21-40.sd.ece.iastate.edu/

Revised: 4/25/2021

Executive Summary

Development Standards & Practices Used

Software development is conducted under clean, consistent practices with thorough documentation. Version control is managed by use of Github. This project follows an adapted Agile development structure with the basic plan, develop, test, deliver, and assess steps. Progress organization will be handled via Git issues. The use of the IEEE 291992-2-2013 standard, which highlights Software Testing, was utilized within the development of our project. This was especially utilized since this standard supports functional and nonfunctional testing as well as manual and automated testing. Another standard that was utilized during the development of our project was the IEEE 14764-2006 standards, which defines the Software Life Cycle Processes, specifically Maintenance.

Summary of Requirements

- Functional Requirements
    - The robot swarm should model the triangular pattern of bird flocks in a 2-D space.
    - The swarm will be made up of two follower robots following a single leader.
    - Only the lead robot may receive movement directions from a base computer station. There may be no communication between swarm members.
    - Both followers must determine their movements from sensor data alone.
    - Followers must maintain 60 cm of separation and a 30° relative angle from the leader.
    - Followers must maintain their position with a 10% maximum error tolerance.
    - The swarm should be able to operate in a closed, controlled environment with enough empty, level floor space for all members to complete their movements unobstructed.
    - Due to COVID restrictions, no lab spaces are available so all project components must be fully simulated in a virtual environment.

    - The total cost of the project should stay within the $500-$750 allotted budget.
    - Approximately 250 to 450 person hours will be required for project completion.
    - (Optional) The swarm should be able to dance to the beat of a song as directed by the leader.
- Non-Functional Requirements
    - The designed swarm system should be robust enough to hold the triangular shape during complicated maneuvers (dances).
    - The leader robot should not have unreasonable restrictions on how it should move.
    - The swarm should be able to withstand various numbers of follower robots while maintaining the same follower functionality.
    - The swarm should be able to dance to different types of songs.

Applicable Courses from Iowa State University Curriculum

Many of the classes that we have taken at ISU will be beneficial in completing our project. The first major course is computer engineering 288. Since everyone in our group is a computer engineering major, we have all taken this class. This class gave us the technical experience with the iRobot Create robots that we will be using in our project. It also gave us insight into embedded systems and how to efficiently integrate our software into the existing hardware. The second applicable course that we all have taken is Computer Science 311. This class is important because it taught us about efficient algorithm design. It also taught us how to use different data structures and how to effectively implement them into an algorithm. The combination of these two classes will help us succeed in completing our project.

New Skills/Knowledge acquired that was not taught in courses

Use of the WeBots software suite was a completely new topic for this project. Since this tool has not been taught in any of our courses, every group member came into this project with no prior experience. Additionally, managing a project as large as this with no in-person interaction was an entirely new concept. With the arrival of the COVID-19 pandemic last March, the first semester of this project was the first semester with fully-online course delivery for Senior Design.

Table of Contents

List of Figures/Tables/Symbols/Definitions

1 Introduction

1.1 Acknowledgement

We would like to thank our advisors Dr. Akhilesh Tyagi, Dr. Diane Rover, and Dr. Phillip Jones, as well as Lee Harker for their assistance and guidance throughout this project. Additionally, we would like to thank the Department of Electrical and Computer Engineering for providing funding for this project.

1.2 Problem and Project Statement

General Problem Statement

Bird swarms are thought to move with a single leader that determines the direction of the flock. All other members simply maintain a certain local position relative to their neighbors. This presents several advantages including increased aerodynamic efficiency, increased scalability for large groups, and simplified navigation for the group. Other animal species such as fish and lobsters have been observed exhibiting similar behavior. This movement heuristic has several applications in the field of robotics including transportation, search-and-rescue operations, and space operations. The purpose of this project is to model the movement behavior of these swarms in two dimensions on the ground with a set of three robots.

General Solution Approach

This project models a triangular bird swarm with a single leader robot and two followers. Only the lead robot receives instructions for movement while both followers have to compute their own movements without any communication between members of the swarm. The followers determine their movements from sensor data alone, maintaining a certain distance between themselves and the other two swarm members.

To determine their local movements, followers will perform several distance readings over a scan frame of 120°. The followers will scan through this frame until they detect a reflector mounted to the center of the leader's chassis. Once detected, the followers will lock onto this reflector, scanning left and right across it in a limited window. As the followers maneuver, this scanning window is free to rotate within the 120° frame.

Once the leader is found, the follower will then make corrections to its left and right wheel speeds to steer in the appropriate direction needed to maintain local positioning with the leader. By doing so, this project will produce a swarm system consisting of three physical robots that can perform coordinated swarm maneuvers within a controlled operating arena without any direct communication between participants.

Though this project originally sought to produce this system in a hardware implementation with three physical robots moving together, this was not possible due to COVID-19 lab restrictions. Instead, this project is entirely virtual in a simulated environment.

1.3 Operational Environment

The robot swarm will operate in a simulation of an indoor space with enough empty floor space for the lead robot to execute its maneuvers and for the followers to properly follow the leader. There should be no physical obstructions within the operating area. The floor of the operating area should be flat and level with an even terrain. Excessive infrared interference such as motion detectors should be removed if possible.

1.4 Requirements

- Functional Requirements
    - The robot swarm should model the triangular pattern of bird flocks.
    - The swarm will be made up of two follower robots following a single leader.
    - Only the lead robot may receive movement directions from a base computer station. There may be no communication between swarm members.
    - Both followers must determine their movements from sensor data alone.
    - Both followers must maintain a certain separation distance between the other two members within an approximate 10% tolerance.
    - The swarm should be able to operate in a closed, controlled environment with enough empty, level floor space for all members to complete their movements unobstructed.
    - Due to COVID restrictions, no lab spaces are available so all project components must be fully simulated in a virtual environment.
    - The total cost of the project should stay within the $500 allotted budget.
    - Approximately 250 to 450 person-hours will be required for project completion.
    - (Optional) The swarm should be able to dance to the beat of a song as directed by the leader.
- Non-Functional Requirements
    - The designed swarm system should be robust enough to hold the triangular shape during complicated maneuvers (dances).
    - The leader robot should not have unreasonable restrictions on how it should move.
    - The swarm should be able to withstand various numbers of follower robots while maintaining the same follower functionality.
    - The swarm should be able to dance to different types of songs.

1.5 Intended Users and Uses

The intended users of our final product are Iowa State University's Cpr E 288 professors. The intended use for this product is to help model and understand swarm robotic behavior and may be used to consider a redesign of the Cpr E 288 course. The results of this project would be incorporated into the course's lab in some capacity, either as a demonstration or student lab project. With the generalized algorithm that this project has proven, it can be more easily reproduced in a similar controlled environment such as the CprE 288 lab.

Considerable research has been done in the field of swarm robotics and its possible applications [1][2]. By modelling the group movement behaviors observed in birds, insects, fish, and even humans, groups of robots can coordinate together to perform complex coordinated tasks. Tasks done as a swarm tend to be much more efficient, scalable, and fault-tolerant than when done with a coordinating body governing the actions of individuals. Rather than being concerned with the group's movement as a whole, individual swarm members only have to maintain their local actions relative to those of their neighbors. Similar to these applications, this project demonstrates group movement by individuals maintaining a local position.

1.6 Assumptions and Limitations

Assumptions

- The swarm will operate without any external EM interference.
- The operating arena will be free of physical obstructions that may impede the swarm.
- The swarm movement will be started with all robots already in formation and fully charged.
- The swarm will consist of three robots.
- All robots that need to be detected by other robots will have adequate reflective material to be properly sensed.

Limitations

- Like a bird swarm, the leader cannot stop instantaneously, turn tightly toward one follower, or turn in place.
- No external IR light sources may be present in the operating arena.
- The project will be conducted fully virtually in a simulated environment with no physical implementation due to COVID-19 lab restrictions.

1.7 Expected End Product and Deliverables

Simulated System
As this project is fully virtual, the deliverable system is a WeBots simulation project fileset with three fully-modelled robot models and a correctly-configured simulation engine. Each robot has its own independent controller executed synchronously with the simulation's physics engine. The two follower robots must correctly implement our follower algorithm and should meet all performance requirements.

Follower Robot
The design for the follower robot consists of an iRobot Create robot base augmented with the added hardware from CprE 288's CyBot. This includes the acrylic baseplate, servo assembly, and sensor head mount. All components must be modelled to the specification of the CyBot and must be verified to function correctly. In addition to these components, a central reflector cylinder was added to allow for a scalable swarm with followers following other followers. Lastly, a controller for each follower must be written implementing our movement algorithm.

Lead Robot
The design for the lead robot consists of a modified follower robot with the servo and distance sensor disabled. The lead robot's controller must take input either from the user's keyboard through the simulation software. This robot will direct the swarm in its movement.

These product models were delivered April 22, 2021.

2 Project Plan

2.1 Task Decomposition

1. Planning
    a. Platform Evaluation - A motion platform was chosen to serve as the motion base and main logical hub for each of the three robots. The iRobot Create platforms from CprE 288 were used for this purpose.
    b. Sensor Evaluation - A shortlist of sensor setups for the two follower robots was compiled for later testing. This sensor setup is the follower's only communication with the outside world.
2. Simulated Design Testing - The entire design was first prototyped in a simulated environment with the WeBots software suite.
    a. Algorithm Design - A generalized algorithm heuristic was devised for the two followers to execute to maintain their distance and the shape of the swarm.
    b. Simulated Platform Movement - The base platform of a swarm member was modelled in the simulated environment and basic movement controls will be implemented through a WeBots node controller. A platform shortlist must be started before this task can be completed.
    c. Simulated LiDAR Testing - A LiDAR sensor node was implemented and tested with a controller to enable distance data to be read from the simulation node. A sensor shortlist is required to complete this task.
    d. Integrated Object Tracking - The LiDAR sensor was integrated with the platform node and a basic tracking algorithm will be implemented to make the simulated bot follow a moving object. Both the simulated platform and LiDAR sensor must be complete to start this task.
3. Virtual Swarm Development - The simulated environment, swarm, and movement algorithm must be developed in order to complete the project.
    a. World Configuration - The physics engine, time step, and operating arena must be properly configured in order to run the simulation.
    b. Swarm Robot Modelling - The three-participant swarm was constructed using the prototype as a basis. The prototype from Taskset 2 must be complete in order to model the swarm.
    c. Lead Robot Control - Keyboard controls were set up to allow the user to give movement commands to the lead robot. Task 2.b. must be complete in order to complete the lead robot's controller.
4. Simulated Algorithm Development - The follower movement algorithm was fully developed to allow followers to correctly form the swarm's shape.
    a. Straight-Line Movement - The previously designed algorithm was implemented with a single robot following a moving target at a fixed distance without changing direction. The followers were given the ability to modulate their speed with the measured distance between the leader and themselves. Tasksets 2 and 3 are required to be finished for this task.
    b. Turning Movement - Followers were given the ability to turn in place in order to follow a turning leader. Tasksets 2 and 3 are required to be finished for this task.
    c. Target Locking - Followers were given the ability to lock onto a lead robot's reflector and track it within a smaller scan window which could slide across the follower's full angular scan frame. Tasksets 2 and 3 are required for this task.

5. Virtual Testing - The constructed swarm members will be integrated into a cohesive group, the swarm behavior will be replicated on hardware, and requirements qualification will be completed.
   a. Basic Movement Testing - Basic movement and sensor integration will be implemented on single robots. The lead robot will be made to follow movement instructions sent from a main controller. Task 4.b. must be completed before starting this task.
   b. Object Tracking Testing - A follower will be made to track a moving target via the mounted LiDAR sensor. Tasksets 3 and 4 must be completed before this point.
   c. Movement Algorithm Testing - The moving target from task 5.b. will be followed at a fixed distance as the target changes direction, following the designed movement algorithm. All tasks must be complete up to this point.
   d. 3-Participant Formation Movement Testing - All three participants will be tested together to perform the swarm movement behavior with both follower robots following the leader. Tasks 5.a., 5.b., and 5.c. must be completed before this point.
   e. Final Qualification - The full system will be formally qualified to ensure that all requirements are met. All previous tasks for the project must be finished to complete this task.

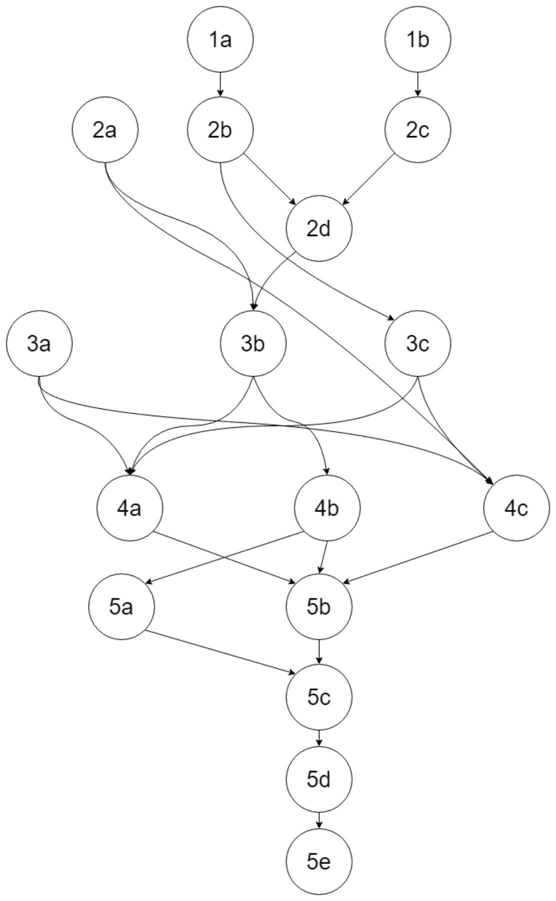


Figure 1 - Subtask Dependency Graph

2.2 Project Timeline/Schedule

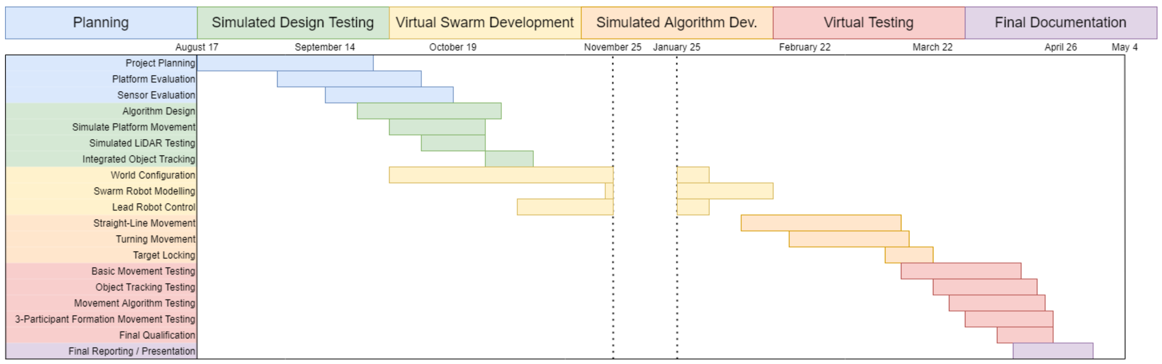


Figure 2 - Project Timeline

Major Project Planning was conducted from the beginning of the fall semester, August 17, through October 19th. This planning included platform and sensor evaluation. During this time, we evaluated which physical and simulated environments would be feasible for this project and researched multiple sensor schemes. The next stage is Simulated Design Testing which lasted from September 21st through November 2nd. This stage was made up of algorithm design, 3-D modeling of the iRobot, simulated LiDAR design, and basic object tracking. The third stage started toward the end of the fall semester and continued into the spring semester. The Virtual Swarm Development phase included the setup of the simulated world, swarm robot modeling, and controlling the lead robot with the keyboard. The next stage, Simulated Algorithm Development, is where the primary development of our project took place. This included moving the swarm in a straight line, then turning left and right, and object target locking. The fifth stage was Virtual Testing. This is where we tested basic movement of the robots, object tracking, movement controls, formation of the follower robots in relation to the leader, and implementation of music controls. The final phase of our project was Documentation and Presentation.

With this 6-phased incremental development roadmap, the project was able to be completed in two semesters. Additionally, many of these tasks could be conducted simultaneously, allowing us to leverage our 6-person workforce.

2.3 Other Resource Requirements

Three iRobot Create motion platforms were originally required for each of the three swarm participants. Additionally, a suitable testing arena in the CprE 288 lab was selected for physical designing and testing. However, since all labs remained closed during the project's second semester due to COVID-19 restrictions, all of our prototyping, system design, testing, and final production was conducted within the WeBots software suite.

2.4 Financial Requirements

A $500-$750 budget was allocated to purchase the additional hardware required for this project such as sensors, additional controllers, or third-party motion platforms. However, as the hardware component of this project was removed and WeBots is a free software, this budget was unnecessary.

3 Design Implementation

3.1 Previous Work And Literature

The most applicable course to this project is CprE 288 as the entire project is centered around the CyBot platform from that course. The embedded systems principles of this course are the backbone of this project's implementation, albeit in a simulated manner. For the follower bots, we will need to implement new designs to allow the bots to sense their surroundings using only the sensors on the bots. The combination of three CyBots into a working design is something that was not done in CprE 288, and required much more work and testing to achieve the desired swarm behavior.

There has been considerable research conducted in the area of swarm robotics, but not in this project's application[1][2]. Most swarm experiments are centered around groups of flying drones such as quadcopters. These swarms usually navigate an area or perform flying maneuvers as a group, sometimes with a designated leader guiding the swarm's moves. Similarly to our project, though, followers simply move to maintain their own local separation between swarm members. The limitation to ground movement alone and to simple sensors is somewhat unique to this project.

3.2 Design Thinking

The design for our robot swarm stems from the already implemented Cybot created by the CprE department. This design has been proven to work in single-bot designs, utilizing all of the equipped sensors. We chose this platform because we are familiar with how it works, and because it can be very accurately modelled in WeBots. Before making this decision, we researched other potential platforms such as the Sphero Rovr. We stayed with the CyBot platform because it already has the fundamental platform components needed and proven performance. All of this would have been missing on the Sphero Rovr, requiring much more time spent finding compatible base robot models and modelling new sensors from scratch. Apart from the platform, we have also decided to swap the attached infrared sensor from the cybot, and replace it with the model of a LiDAR Mini sensor.

This new sensor allows for faster scanning speeds than the CyBot's stock sensor and is more accurate to boot. We considered a few different versions of a LiDAR sensor, including an iteration that would allow for 360° scanning. We decided to go with the LiDAR Mini as it would easily be modeled in WeBots and could realistically be added to the CyBot if this project is implemented in hardware in the future.

3.3 Proposed Design

Our design centers around the iRobot Create platform for two main reasons. Firstly, ISU has several of the platforms available for the project which we planned to use at this project's outset. This would have allowed us to more easily fit within our required budget without limiting the movement or sensor capabilities of the participants. Secondly, our simulation software, WeBots, has a prebuilt node for the iRobot Create. This saved a considerable amount of time in modelling and improved the realism of the simulation. This allowed us to pivot to implementing the project entirely within WeBots when ISU's labs remained closed at the beginning of this project's second semester.

To allow follower robots to detect the leader's position, a Parallax Standard Servo is mounted on the CyBot with a sensor head attached to the servo arm assembly. This sensor head mounts a LiDAR distance sensor capable of detecting the distance between the sensor and the nearest reflecting object within the sensor's line of sight. Dimensions of the servo arm and mounting assemblies were found in design schematics available from the ETG.

By sweeping the sensor through a portion of its 180-degree Field of View (FoV), a picture of the robot's surroundings can be constructed. Using this setup, the follower robots are able to lock onto the leader, figure out the leader's position relative to themselves, and maneuver to maintain a certain separation between the moving leader and itself. This results in a swarm-like movement with followers simply focused on maintaining their local position as described in the figure below.

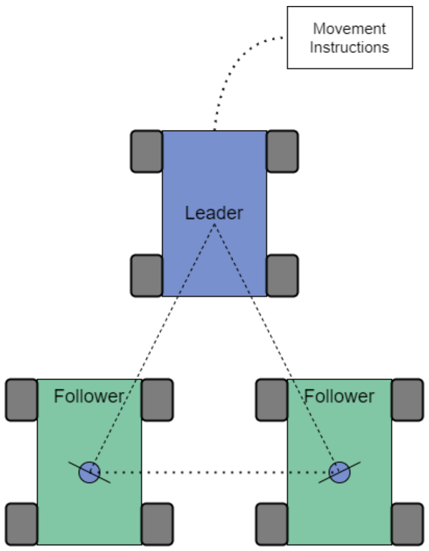


Figure 3 - Proposed Design Diagram

The determination of a follower's next move is a product of the robot's distance to the leader read by the LiDAR sensor and the relative angle between the follower's nose and the leader's detected position. If the follower reads the leader as moving too far left or right from the center of the follower's FoV, the scanning window will be slewed left or right to maintain a lock on the leader. The follower will then steer relative to this angular rotation of the FoV window. Doing so will realign the follower's movement vector with that of the leader.

We elected to use WeBots for our modelling due to its high fidelity of sensor and physics modelling, expansion, and wide availability of documentation. The prebuilt iRobot Create node was used as a base for our simulated robot. From there, additional child nodes were added to model the CyBot's added acrylic plates, circuit boards, and extra hardware mounted on top of the base platform. Dimensional schematics of these added components were, again, provided by the ETG.

Though we have had some difficulties with modelling all geometries correctly and ensuring that child nodes interact with their surroundings properly, WeBots was still the best tool for our uses. It had the best balance of expandability, simplicity of use, realism, and potential portability to hardware. Now that we have a refined WeBots swarm model, it is possible for the design to be ported to hardware in the future.

3.4 Implementation

WeBots Simulation

Each robot is composed of a node tree stemming from a Robot node. The base robot node has an iRobot Create child node serving as the platform for each robot. This child node contains an attribute which associates a controller with a given robot. It also contains a bodySlot attachment which serves as a connecting point to attach additional nodes to the robot. By putting these added nodes in the bodySlot, they will become accessible to the robot's controller as well as being physically attached to the robot's physics model. This bodySlot contains the geometric models of all components attached to the CyBot's back, as well as the servo's HingeJoint node.

A HingeJoint allows you to define an axis which other nodes can rotate around. In the case of our servo model, the HingeJoint is configured as a motor with parameters identical to those of the CyBot's servo. The HingeJoint node has an endPoint attachment which acts like the robot's bodySlot. All nodes attached to the endPoint will rotate with the motor axle when the controller commands it to turn.

The endPoint of the servo model contains the geometric components of the sensor head assembly, the sensor block, and the sensor device itself. Modelled as a DistanceSensor node, the LiDAR sensor contains a lookup table accessible by the controller. This table contains the data corresponding to the distance between the sensor and the nearest object along its line of sight. We configured our DistanceSensor to the data from the LiDAR Mini's datasheet. The following class diagram describes the node configuration of each robot in the simulation.

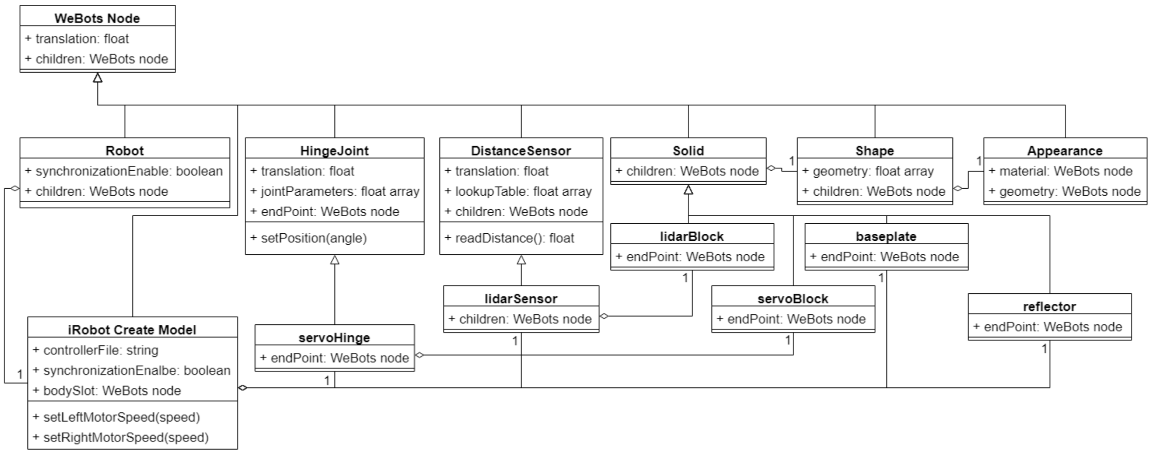


Figure 4 - WeBots Robot Node Class Diagram

The WeBots world simulator synchronizes all robot controllers and operations by the use of time steps. A special internal WeBots function will pause code execution until a specified number of time steps in the simulated world have passed. In the case of our simulation, we have one time step pause per scan of the LiDAR sensor with a time step period of 10 ms. This allows us to simulate data processing time, as well as limiting the sample rate of our distance sensor to 100 Hz as is specified in the LiDAR Mini's datasheet. The below figure shows a screenshot from our finished simulation.

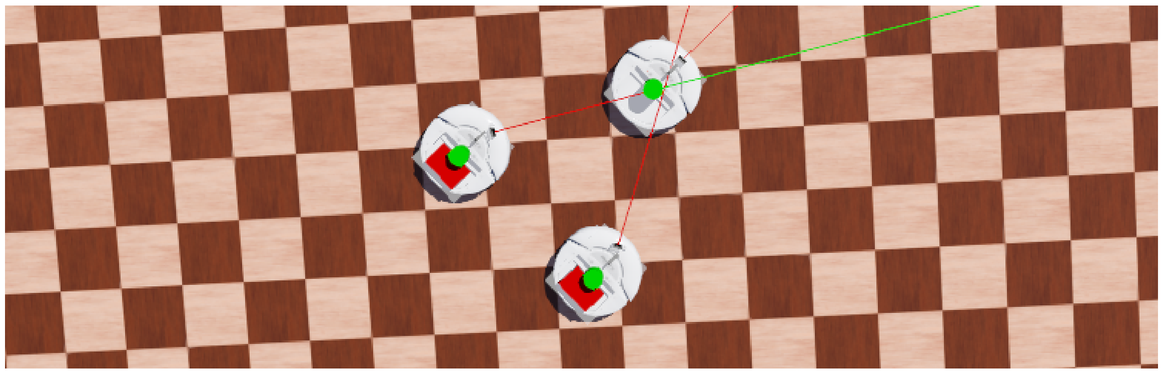


Figure 5 - WeBots Simulation Screenshot

Follower Algorithm

The follower algorithm relies primarily on its ability to lock onto the leader's reflector and track it in a minimized window within its 120° scan frame. This is done in two phases. The first phase has the follower continue its current movement operation while seeking a lock. This allows a follower to potentially catch up to a lost formation or to complete a turn and then lock back onto the leader. As a follower searches for a lock, it will sweep through its full scan frame until it reads a distance under 1 m. If the sensor reaches one end of the scan frame, then the scan direction is reversed and any cached data from the current sweep is cleared.

Once a follower finds a reflector within 1 m, it will begin the second phase: active tracking. When a reading under 1 m is first read, a flag is set indicating that a reflector has been found. It will continue sampling across the face of the reflector until it reads a distance greater than 1 m, indicating that it has found the opposite edge of the reflector. The flag is then reset, the direction of the scan reversed, and the whole process starts again. If the follower manages to lose its lock on the leader and no sample under 1 m is read, then the first phase begins again. This process allows for very rapid tracking of a leader's reflector and a higher response rate by the follower robots.

At every sample within the 1 m sensitivity range, a movement decision is made. If the angle at which the sample was made is outside of a +/-4° deadzone around the target 30° angular offset, then the robot will turn in place toward the measurement. If the angle is within the deadzone but the measured distance is outside of a +/-2 cm deadzone around the target 60 cm separation distance, then the forward speed of the robot is changed. For the iRobot Create model in WeBots, wheel speeds are set as floating point values from -16 to 16 where 0 is stationary, negative values are reverse, and positive values are forward. If the measured separation distance is greater than the target, then the speed is set according to the following formula:

$$speed = 20 * (10 * (target - distance)/target + 20 * ((target - distance)/target)^2)$$

If the measured separation distance is less than the target, then the speed is set according to the following formula:

$$speed = 20 * (10 * (target - distance)/target - 20 * ((target - distance)/target)^2)$$

Both equations are bounded by the algorithm within the +/-16 bounds of the platform. The use of a quadratic equation here allows the robot to speed up quickly if it falls significantly behind the target while also coming to a more gradual stop at low error distances. The coefficients for these equations were determined experimentally. The following flowchart describes both the two-phase locking functionality and movement decision making of the algorithm.

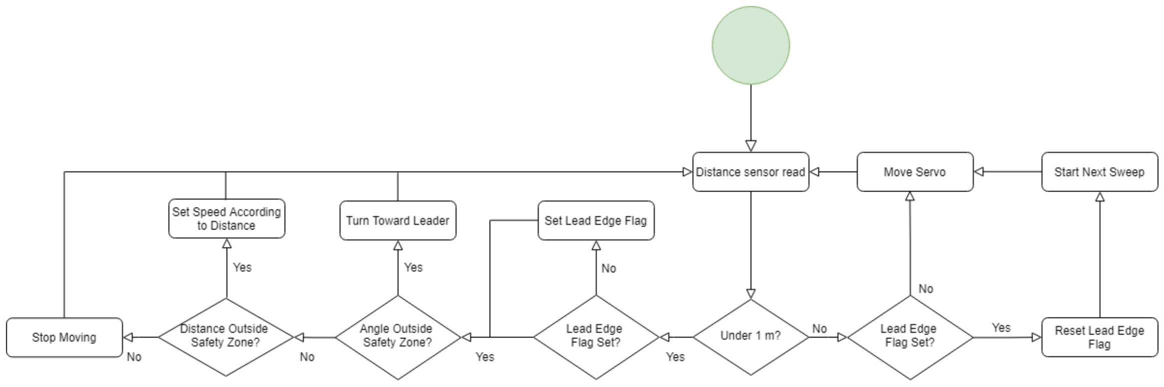


Figure 6 - Follower Algorithm Flowchart

Music Parser

Initially, we were going to write a program that would parse midi files. Midi files contain useful information regarding the song information, such as lyrics, notes, pitch, frequency, tempo and time signature. Unfortunately, due to time restrictions placed upon us, as well as the difficulty of parsing midi files, we opted to parse the music information in a different way. We manually capture the notes played by a song, as well as the duration of each note, and record the information into a text file named with the song title. The program asks the user to specify the song that they would like to parse (given that the music file that contains the notes and note durations is provided). If the music file is found, the program would parse the information written. This makes it so we can get unique dances for all songs. Within the program that parses the notes, we read each note played and write the corresponding movement to an external .txt file. Each note corresponds to a different movement that the leader will receive. The notes and the movement corresponding to the note are as follows:

Note	Movement
A	Backwards (denoted as b in the movement file)
B	Turn right in place (denoted as r in the movement file)
C	Turn left in place (denoted as l in the movement file)
D	Turn Right while moving (denoted as R in the movement file)
E	Turn Left while moving (denoted as L in the movement file)
F	Move Forward (denoted as f in the movement file)
G	Stop (denoted as s in the movement file)

Figure 7 - Note-Movement Conversion

Once we capture the note and record the movement into the text file, we read in the duration of which the note is played. If a note is played for 250 ms (1/4th of a second) we would insert 2 periods(.), since each period represents a step which is 1/8th of a second.. The periods inform the robot to hold the previous movement that was recorded. For example if the the note F#4 was played for a duration of 250 ms, we would insert an f into the movement file followed by 2 periods so the robot would continue moving forward for a total of 250 ms.

3.5 Technology Considerations

Our simulation software, WeBots, has some considerable simplifications in terms of environmental interactions with the modelled robots. Though it has the capability to model noise in signals, we were not able to implement this component this semester. Because we had to drop the hardware component halfway through the project, redesigning the world model to add EMI interference and noise was not practical.

These potential issues must be dealt with if the system is to be reproduced on actual hardware, as there isn't a way to effectively reproduce them in the simulator. Additionally, the controller code in the simulation is not identical to the code needed to implement the actual sensor hardware we originally planned to use, so this will need to be considered if the project is to be used as a physical CprE 288 demo. Despite these shortcomings, though, WeBots is still the most effective simulation tool that we can make use of, given our budget and time constraints.

3.6 Design Analysis

In the end, our proposed design seems to be effective for our purposes. One major limitation with the design having only a single distance sensor is the fact that follower robots cannot tell if or when the lead robot rotates in place. Additionally, if the leader abruptly changes its course at high speed, it is possible for a follower to lose its lock and become lost. However, both of these movement cases do not reflect the swarm movement of birds that we are trying to model with this project. After all, most bird swarms aren't capable of hovering and turning in place or quickly changing course without collapsing on themselves. Therefore, we chose to limit our scope to following leader movements that are more gradual in nature. This both alleviates some potential issues with the design and reflects a more realistic model.

The overall design of our project changed going into the second semester. We originally planned on developing the project using physical hardware, such as the iCreate Roombas available in Coover. Unfortunately, due to restrictions placed upon us because of the pandemic, our project was completely virtual. However, the design of how we wanted to implement the swarm movement stayed relatively the same. Throughout this semester, we had to analyze the meaning of "swarm" and what it means to be a leader robot. Throughout our PIRM presentations, we encountered several questions revolving around if the swarm were to turn, would a new robot be assigned as the leader or would the swarm still follow the predefined leader. We were also asked, "What swarm movement are we basing our design on?" since bird swarm movements are different from other swarms, such as fish swarms. With these questions, we were able to design our project more thoroughly. Taking these design considerations into account, we were able to produce a design that effectively met all of our project requirements in a fully virtual environment.

3.7 Development Process

Our project followed an adapted Agile development process. As the method has been proven in other projects and has been emphasized through the course lectures, we decided it would be the best fit. Incremental features were implemented in a sprint-like structure with time frames shifted for the pace of the project. The flexibility of Agile proved to be a significant advantage at the beginning of our second semester. Since we were forced to drop the hardware component of the project halfway through, we were able to take the prototyped components from the project's first semester and redirect our future sprints with them as a basis.

3.8 Project Evolution

Our original plan for this project was a two-phase development roadmap. The first semester would be spent prototyping our robot model based off of the CyBot's specifications with some limited controller development. To demonstrate the functionality of all robot subcomponents, a demo object-avoidance controller was made which allowed a robot to maneuver through a simple obstacle course.

We had initially started using WeBots solely as a method to inexpensively prototype our design with the intention of recreating it in the Embedded Systems lab. However, when labs remained closed at the beginning of the second semester, the entire focus of the project pivoted to WeBots once we ensured it was a viable platform for full swarm implementation.

Three robot models were made from the prototype and adapted to each swarm participant's needs. A reflector cylinder was added to each robot to allow them to see each other. The lead robot's controller was augmented with keyboard controls so we could direct the swarm leader ourselves. With all hardware modelling complete, our simple algorithm went through four revisions with essential features added at each step. Starting as a basic straight-line follow-the-leader program without any turning allowed, followers gained the capability to steer, control their speed dynamically, turn in place, and finally lock onto only the lead robot's reflector.

Some experimentation was done to see how scalable the swarm design was by adding additional followers and having them latch onto another follower. This creates a large V pattern, similar to how migratory geese fly in formation. Though tight turns tend to flatten and eventually collapse the swarm's shape, we found it had a remarkable ability to self-repair and regain its V shape. The below figures show an example of this recovery ability on a swarm with two follower layers. The swarm shape was recovered by simply driving the leader straight ahead until the swarm repaired itself.

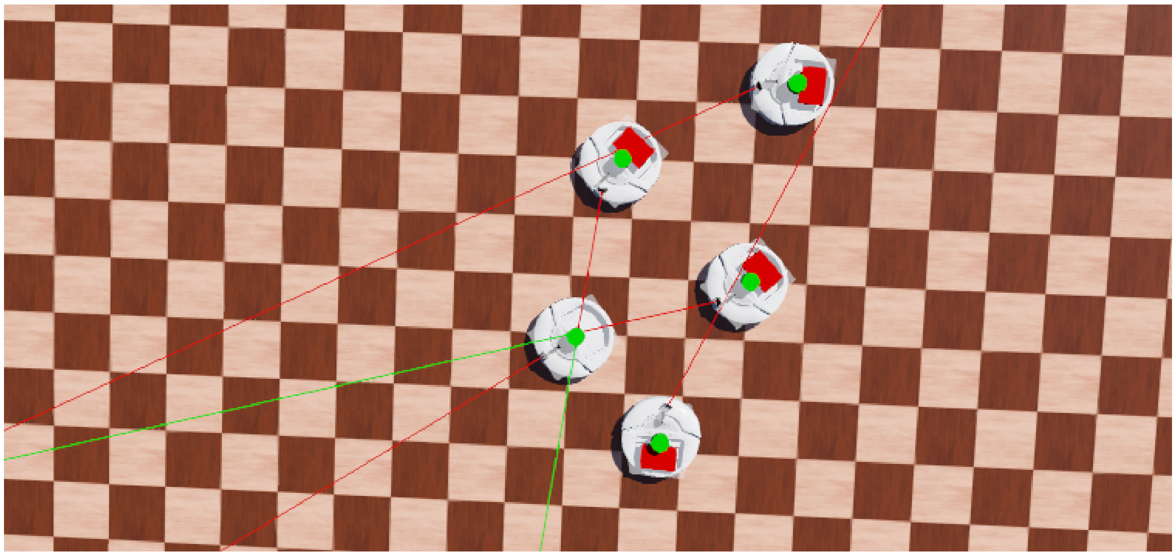


Figure 8 - Collapsed Two-Tier Swarm

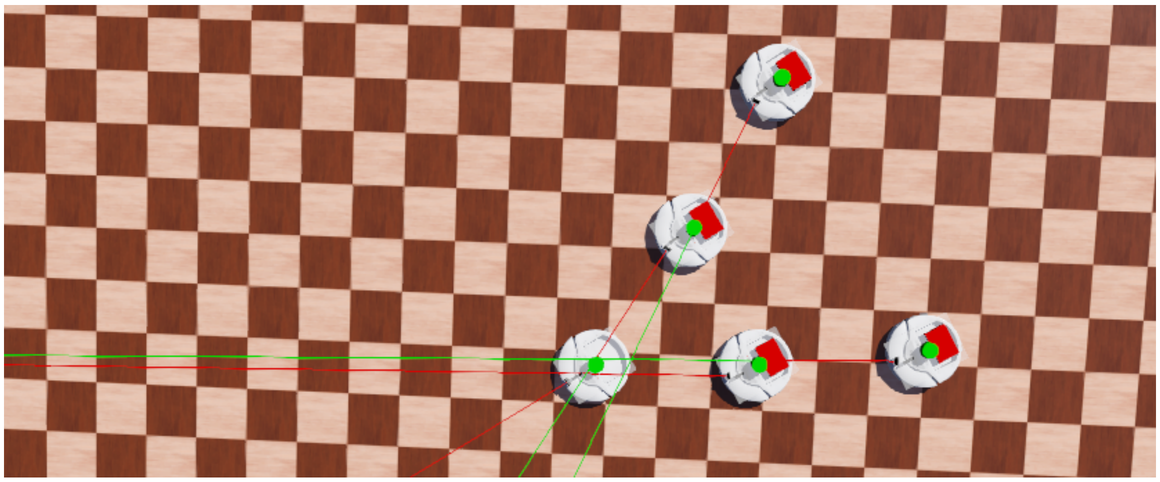


Figure 9 - Recovered Two-Tier Swarm

During the last month of the semester, we re-introduced the idea of making the swarm dance to a song with a music parser. This parser would take data from a song and construct a dance pattern for the lead robot. The lead robot reads this file and directs the swarm in a combined dance. Though there were several unexpected twists and turns, we were able to meet all of our goals in this new simulated environment.

4 Testing

4.1 Unit Testing

Unit testing of the followers' LiDAR distance sensors were conducted using a series of modelled reflective objects modelled in our simulation placed between 40 cm and 100 cm away in increments of 5 cm from the robot's sensor head. Sensor readings were taken to verify correct distance measurements from the follower robot. Once verified, this test set was performed again after integrating the distance-based variable speed control function. At each increment, the output speed of the speed controller was verified against a table that we created of expected speeds for a given distance.

To test the servo motor on each follower, a special controller was made which would step the motor through the full 120° sweep frame in 1° increments. This controller was run with the simulation paused every 5 intervals. With each pause, the position of the servo and sensor head was verified within the simulation.

Lastly, the leader's keyboard-controlled movement function was verified with a set of sample maneuvers. Once the key patterns were entered by the controller, the leader's behavior was observed to verify the correct movement of the leader.

4.2 System Testing

Once all components of each robot were tested and verified to be functional, the three robots were integrated into a single swarm. First, it was verified that each follower could lock onto the leader's reflector by use of the servo and LiDAR sensor in combination with each other. Both followers were positioned 60 cm from the leader's reflector and the simulation was run. When working correctly, both followers would jog their servo-mounted sensors back and forth across the reflector with only one sample made on either side.

With the Lock-On functionality verified, the lead robot was moved forward for varying distances. While maintaining lock, the followers would record the maximum and minimum distances read by their distance sensors. This test passes if both followers speed up and slow down properly with the leader without losing a sensor lock or falling more than 6 cm outside of the 60 cm target separation distance.

Once the Straight-Line Test passed, a turn test was performed to check the correct operation of the differential steering capability of the movement algorithm. Like the previous test, the leader was moved forward for varying distances with occasional stops. In addition the leader would turn left or right while moving forward. This test passes if the followers meet all of the Straight-Line Test requirements, as well as maintaining the triangular shape of the formation. Like with bird flocks, a slight flattening of the formation in the direction of the turn is tolerated, but the swarm's shape should not collapse.

After turning maneuvers were verified, an extended run of the turn test was performed for 30 simulation minutes. This was done in WeBot's Accelerated Mode, rendering the simulation at 10x speed. While maneuvering, the swarm's shape was observed to make sure it did not collapse. After maneuvering for this extended time, the maximum and minimum readings of each follower were checked to ensure the 10% error tolerance was not violated.

4.3 Acceptance Testing

During the testing phase, we were able to video record our simulation while it was running. In order to ensure that the functional requirements have been met, we designed a sample maneuver set, recorded the simulation, and delivered it to the client for approval. A live demo was also shown to the client and approved. We also created our own secondary test cases for additional proofs of concept.

4.4 Results

All LiDAR, servo, and speed control unit tests passed as expected. As there is no noise in the simulated system, all three test sets came out exactly according to their expected values. With a stationary leader, error margins were under 1%. Straight-line movement error margins tended to not exceed 4%, except in the case of illegal reversing movements. With turning maneuvers added, the error margin did not exceed 8%. This sits well within our 10% allowable margin. Even over the extended-time maneuvering test, the error margin did not break our limit.

When experimenting with illegal movements, it was found that extended periods of reversing or turning tightly toward one follower. As these maneuvers do not reflect the real-world behavior of bird flocks in flight, though, they are not allowed in this project's model. The below figure shows one of these collapses.

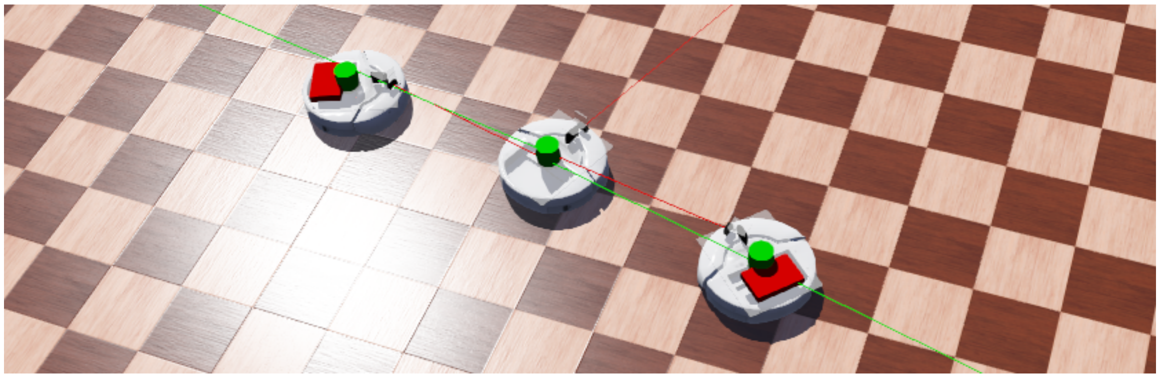


Figure 10 - Collapsed Formation After Repeated Illegal Movements

5 Closing Material

5.1 Conclusion

During the fall semester, we completed our preliminary modelling of the CyBot platform in WeBots. We selected the WeBots software suite to conduct our simulated prototyping because it offers the greatest level of realism without an excessive setup overhead cost. Additionally, the software is free with plenty of documentation available. Other similar simulation software platforms are either too simplified to be useful for our testing, have an excessive amount of setup required to model different designs, or don't allow for realistic controller development. Though the controllers used in WeBots are relatively simplified with sensors and actuators being modelled as lookup tables rather than actual memory-mapped components, it still allows us to test the system at a higher level. By use of this controller scheme, we are still able to design and test the underlying movement algorithm, albeit with a slightly more abstracted implementation.

Our simulated model includes the iRobot Create base, added acrylic plates, circuit boards, servo, and sensor head assembly. We chose to use ISU's Cybot design as a platform due to its simplicity and cost-effectiveness. Since the university already has hardware platforms available, using them would eliminate the cost of purchasing a new robot platform for each of the three individual swarm members. Additionally, the CyBot already has all of the three required components for a follower: a controllable movement platform, a distance sensor to measure separation from the leader, and a rotating sensor head assembly mounted on a programmable servo. The only required addition would be an optional improved LiDAR sensor for the two followers to more accurately and quickly perform distance measurements.

The simulated robot's LiDAR sensor and servo is controllable by the robot's controller code. This controller is written to resemble the code of a potential future hardware implementation. The controller implements our follower algorithm design with straight-line speed control, turning capabilities, and the ability to lock onto a target reflector within a follower's scan frame. This controller is able to direct follower movements in the swarm without breaking our required position error tolerance.

During the spring semester, we completed our follower algorithm and successfully simulated swarm behavior using the WeBots software suite. Additionally, we implemented a parsing tool to convert music files into sets of instructions in the form of a "dance routine" for the leader to implement as follower bots used the follower algorithm to replicate these movements, displaying swarm behavior. While .midi files were not directly parsed due to time constraints, music files were manually abstracted into notes, with each note corresponding with an action (i.e. "dance move") as described in the section 3.4 table. Overall, the project was successful, meeting all previously-defined requirements and functioned as intended.

5.2 References

[1] Tan, Ying, and Zhong-Yang Zheng. “Research Advance in Swarm Robotics.” Defence Technology, vol. 9, no. 1, 2013, pp. 18–39., doi:10.1016/j.dt.2013.03.001.

[2] Schranz, Melanie, et al. “Swarm Robotic Behaviors and Current Applications.” Frontiers in Robotics and AI, vol. 7, 2020, doi:10.3389/frobt.2020.00036.

Appendices

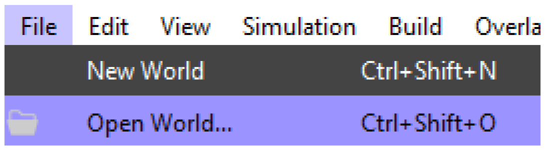
Appendix I Operation Manual

Step 1. Setup

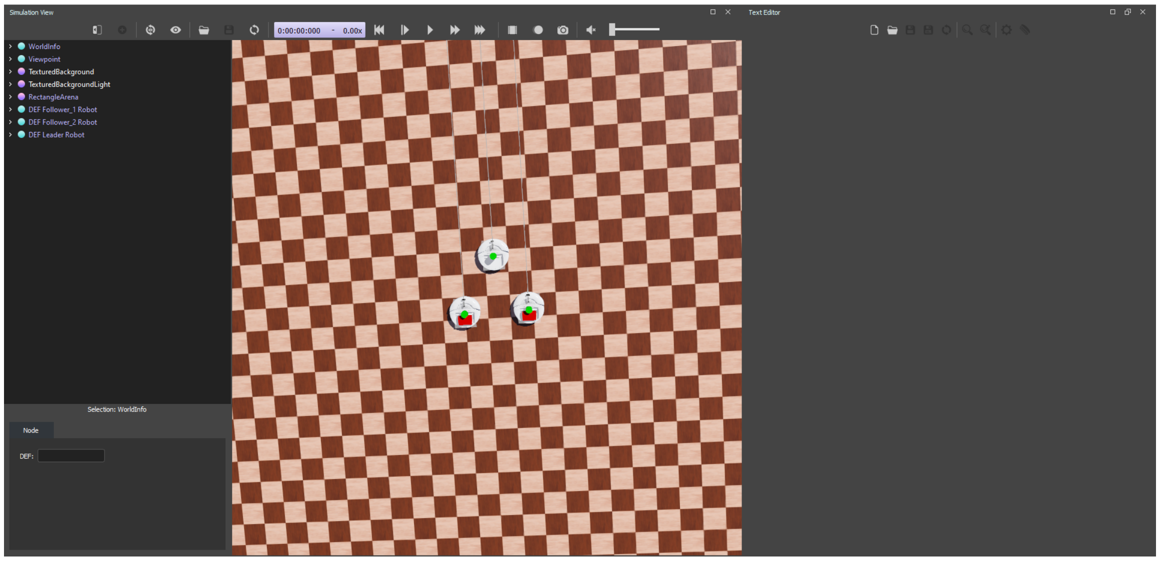
This project revolves around the WeBots simulation software, available for free from cyberbotics.com. All testing and demonstrations are done through this software suite. Begin by downloading and installing the software from Cyberbotics's website. Additionally, a local clone of our repository will be needed. This repo can be found at git.ece.iastate.edu/sd/sdmay21-40. With the software installed and repository pulled, the world file containing the project can now be opened.

Step 2. Opening Project

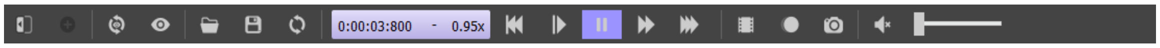
With WeBots installed and opened, click File > Open World... as seen below.



Navigate to the directory where your local copy of the repository is located. From there, select sdmay21-40/WeBots/test/worlds/testWorld.wbt. Once open, your window should appear like the image below with three robots positioned in a triangle.



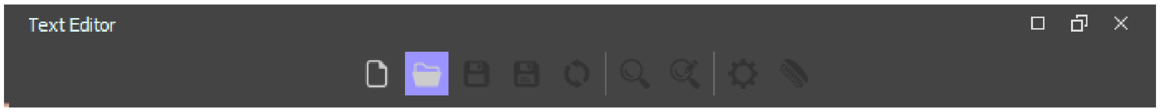
If the simulation starts automatically running, click the pause button in the toolbar across the top of the window as seen in the following image.



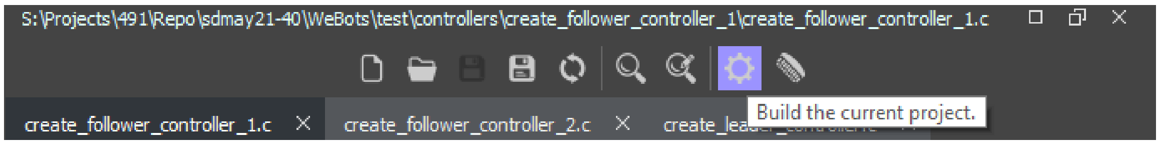
To display a ray showing the orientation of the robots' distance sensors in the simulation world, check the option under View > Optional Rendering > Show DistanceSensor Rays. Enabling this option renders a projection line from each distance sensor in the world that changes from red to green once it reads an object. This will allow you to see exactly what each of the followers sees.

Step 3. Compiling Controllers

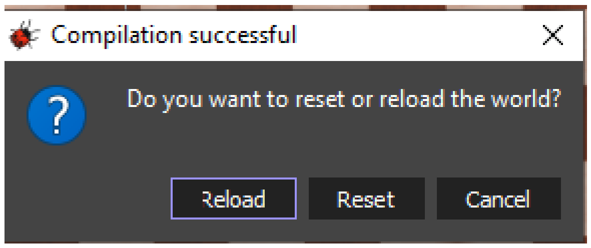
Once you have the world configured, you will need to compile each robot's controller to ensure correct operation. To do this, click the Open File button in the Text Editor pane at the right of the window. You may need to resize the pane if the button is not visible. If the Text Editor pane is not open, enable it under Tools > Text Editor in the File Menu Bar.



The three controllers will be located at sdmay21-40/WeBots/test/controllers/create_follower_controller_1/create_follower_controller_1.c, sdmay21-40/WeBots/test/controllers/create_follower_controller_2/create_follower_controller_2.c, and sdmay21-40/WeBots/test/controllers/create_leader_controller/create_leader_controller.c respectively. Once all three controllers are open in the Text Editor pane, select create_follower_controller_1.c and click the Compile button at the top of the Text Editor pane.

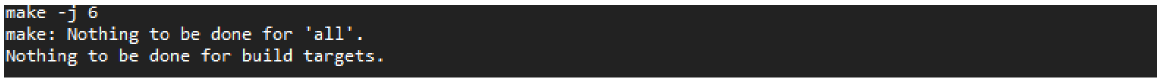

S:\Projects\491\Repo\sdmay21-40\WeBots\test\controllers\create_follower_controller_1\create_follower_controller_1.c
create_follower_controller_1.c
create_follower_controller_2.c
create_lea
Build the current project.

This will begin building the controller currently selected and a printout of its progress will appear in the console at the bottom of the window. Once the compilation is complete, a success window will appear. Click Reload.


Compilation successful

Do you want to reset or reload the world?

If the console prints the following message:

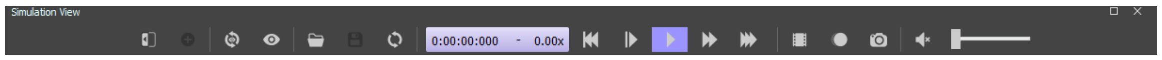


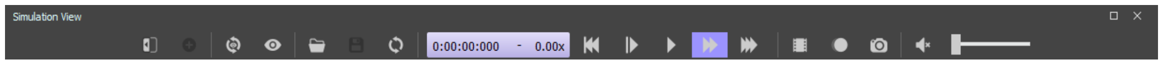
```
make -j 6
make: Nothing to be done for 'all'.
Nothing to be done for build targets.
```

Then no compilation is necessary for your WeBots install. Repeat the above compilation process for create_follower_controller_2.c and create_leader_controller.c.

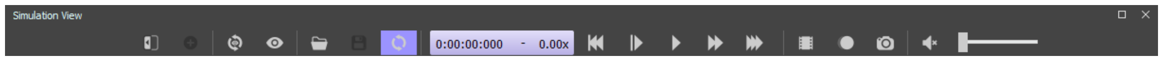
Step 4. Running Simulation

With all three controllers compiled, the simulation can now be run. Begin running the simulation by clicking either the Real-Time Simulation or Accelerated Simulation button at the top of the Simulation View pane as seen below.





Running in Real-Time Mode has a 1:1 correspondence with simulation time vs real time. Accelerated Mode will run the simulation as fast as the simulator is able. Because the speed of the iRobot Create is so slow, it is recommended to use Accelerated Mode. To pause the simulation, click the run button corresponding to your current mode again. To reset the simulation back to the original position, click the Reset button as shown below.

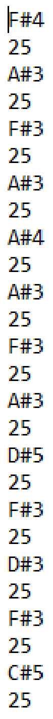


If a pop-up window appears asking to save changes, click Discard. Once the simulation begins running, both followers will immediately lock onto the leader and position themselves around it.

Step 5. Controlling Swarm

With the simulation running, click anywhere on the checkered floor board to select the Simulation View panel. With this panel selected, you can now control the leader robot with your keyboard. Standard WASD keys are used for control forward, left, reverse, and right respectively. As a general rule, the swarm model only allows the lead robot to move forward and make arcing turns left and right as bird flocks in real-life do not instantaneously stop, turn in place, or move in reverse. Though the swarm is generally able to handle illegal moves outside of this envelope, it is possible to collapse the formation by reversing into the followers for long periods of time, turning sharply toward one follower for several rotations, or running the swarm into the arena's walls.

An alternate way to control the swarm is using the movements recorded from the music parser. In order to parse a music file, a text file with the notes played as well as the duration of each note must be provided.

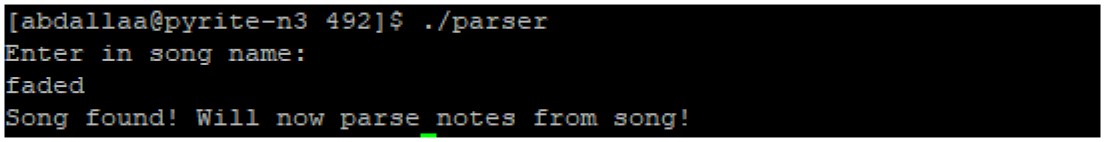


```
F#4
25
A#3
25
F#3
25
A#3
25
A#4
25
A#3
25
F#3
25
A#3
25
D#5
25
F#3
25
D#3
25
F#3
25
C#5
25
```

Compile the music parser, which is found under our project in the folder labeled *Music Parser,* using the following steps.

1. gcc -c musicParser.c
2. gcc -o parser musicParser.o

You will now have an executable file which can be run by writing *./parser*. Once the program is run, you will be asked to input in the name of the song you would like to parse. The name will be the name of the note information file, excluding the .txt extension.



```
[abdallaa@pyrite-n3 492]$ ./parser
Enter in song name:
faded
Song found! Will now parse notes from song!
```

If the file is found, the program will parse the notes and create a file labeled song.txt which highlights the movements that the robot will dance to. After the file is created, place the song.txt file into the leader controller folder which is located at *WeBots/test/controllers/create_leader_controller*. Once you run the world you can start the dance sequence by pressing the ***r*** key on your keyboard.

Appendix II Alternative Versions

Hardware Implementation

Our original project vision was to produce the three-robot system with a hardware implementation. This system would be composed of three modified CyBot robots from the CprE 288 Embedded Systems lab. However, due to lab closures from the COVID-19 pandemic, the hardware component of our project was dropped and the whole project was completed in the WeBots software suite.

360° LiDAR

An early idea for the design of our participant robot was to use a 360° rotating LiDAR sensor for the follower robots. Using this sensor would allow us to perform scans much faster and with more accuracy than our servo-mounted design, however designing a sensor mount presented several complications. Since the system would require two sensors at play, there was a risk of cross-talk between sensors if their rotations accidentally synchronized with the LiDAR heads facing each other. This could be overcome by mounting one sensor slightly higher than the other, but this approach would not be very scalable if more robots were added to the system. The mount for this sensor would also require fabricating a special bracket for the CyBot while our final design required no modifications to the CyBot body.

Single-Decision Sweep

Our initial algorithm design only made movement decisions after a full end-to-end sweep of the sensor was performed. In this case, a follower robot would begin its sweep at the left hand side of its sliding scanning window. It would then make samples at 1° increments across the leader's reflector until either the right hand bound of the window frame was reached or the right edge of the reflector was detected. The shortest measured distance would be recorded along with the angle position at which it was made. After the sweep was complete, this sample would then be processed to determine the robot's movement. The same process would then be executed in the opposite direction, scanning right to left.

This approach would give a more precise position of where the leader is in relation to a given follower, but ended up producing a larger position error margin. Because movements could only be determined after a full sweep was complete rather than at every sample and a full sweep consisted of 4-7 samples, the response time of followers was increased by 4-7 times. Additionally, since there is an 8° deadzone around the target offset angle of 30°, having a possible error of 4-7° would not be sufficient to produce "wagging" effect as a follower reads each edge of the leader's reflector.

Multi-Sensor

One large drawback to our design is the fact that followers are not sensitive to the leader turning in place. This is caused by our choice to make the leader's reflector a cylinder to ensure that LiDAR measurements are not reflected away from the follower's sensor. However, this makes it impossible to measure the leader's orientation. One of our alternative designs involved multiple LiDAR sensors tracking multiple reflectors mounted in different positions on the leader's body. This would allow the followers to tell exactly which direction the leader's nose is facing and grant them the capability to orient the entire swarm around the leader even when turning in place. To produce such a design, though, would require significant redesign to the algorithm and the CyBot's body. Since this would be impractical to reproduce in a hardware implementation, we chose to stick to the CyBot base platform.

Appendix III Other Considerations

Music Parsing

Initially, we considered parsing the music in a completely different way. Through research on available sensors that are able to be implemented on a physical microcontroller, we identified that the LM393 Sound Detection sensor was an effective sensor to pick up on frequencies played by a song. However, this sensor was not available on WeBots so we initially entertained the idea of developing the sensor from scratch onto WeBots. However due to our inexperience, this idea was quickly scrapped. We later decided to parse a musical file, and have the leader controller read from the parsed information. We originally were going to parse either .aiff or .midi files and after thorough research we identified that .midi files were better for our case. However due to time restrictions, and the complexity of parsing midi files, we decided to parse music files as how it is defined above.

Dance Movements

Once the robots took in the dance movements (from either a musical file or from keyboard command), we realized that if the leader robot were to spin in place, our current design does not allow for the followers to pick up on that motion. Instead, the followers would remain in place waiting for the next movement instruction. We considered sending in that specific instruction to the followers as well. However, we decided not to since it goes against a core element of our project, which is that only the leader is aware of the movement instructions.

Appendix IV Code

Leader Controller Code

```
/*
 * Description:  Controller for leader robot
 */

/* include headers */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <webots/distance_sensor.h>
#include <webots/led.h>
#include <webots/motor.h>
#include <webots/position_sensor.h>
#include <webots/receiver.h>
#include <webots/robot.h>
#include <webots/touch_sensor.h>
#include <webots/keyboard.h>

/* device stuff */
#define BUMPERS_NUMBER 2
#define BUMPER_LEFT 0
#define BUMPER_RIGHT 1
static WbDeviceTag bumpers[BUMPERS_NUMBER];
static const char *bumpers_name[BUMPERS_NUMBER] = {"bumper_left", "bumper_right"};

#define CLIFF_SENSORS_NUMBER 4
#define CLIFF_SENSOR_LEFT 0
#define CLIFF_SENSOR_FRONT_LEFT 1
#define CLIFF_SENSOR_FRONT_RIGHT 2
#define CLIFF_SENSOR_RIGHT 3
static WbDeviceTag cliff_sensors[CLIFF_SENSORS_NUMBER];
static const char *cliff_sensors_name[CLIFF_SENSORS_NUMBER] = {"cliff_left", "cliff_front_left",
"cliff_front_right",
                                                               "cliff_right"};

#define LEDS_NUMBER 3
#define LED_ON 0
#define LED_PLAY 1
#define LED_STEP 2
static WbDeviceTag leds[LEDS_NUMBER];
static const char *leds_name[LEDS_NUMBER] = {"led_on", "led_play", "led_step"};

static WbDeviceTag receiver;
static const char *receiver_name = "receiver";

WbDeviceTag left_motor, right_motor, left_position_sensor, right_position_sensor, servo, lidar;

/* Misc Stuff */
#define MAX_SPEED 16
#define HALF_SPEED 8
#define FIVE_SPEED 5
#define THREE_SPEED 3
#define ZERO_SPEED 0

#define WHEEL_RADIUS 0.031
#define AXLE_LENGTH 0.271756
#define ENCODER_RESOLUTION 507.9188

/* helper functions */
static int get_time_step() {
  static int time_step = -1;
  if (time_step == -1)
    time_step = (int)wb_robot_get_basic_time_step();
  return time_step;
}

static void step() {
  if (wb_robot_step(get_time_step()) == -1) {
    wb_robot_cleanup();
    exit(EXIT_SUCCESS);
  }
}
```

```
static void init_devices() {
  int i;

  receiver = wb_robot_get_device(receiver_name);
  wb_receiver_enable(receiver, get_time_step());

  for (i = 0; i < LEDS_NUMBER; i++)
    leds[i] = wb_robot_get_device(leds_name[i]);

  for (i = 0; i < BUMPERS_NUMBER; i++) {
    bumpers[i] = wb_robot_get_device(bumpers_name[i]);
    wb_touch_sensor_enable(bumpers[i], get_time_step());
  }

  for (i = 0; i < CLIFF_SENSORS_NUMBER; i++) {
    cliff_sensors[i] = wb_robot_get_device(cliff_sensors_name[i]);
    wb_distance_sensor_enable(cliff_sensors[i], get_time_step());
  }

  left_motor = wb_robot_get_device("left wheel motor");
  right_motor = wb_robot_get_device("right wheel motor");
  servo = wb_robot_get_device("servo motor 1");
  lidar = wb_robot_get_device("lidar sensor 1");
  wb_motor_set_position(left_motor, INFINITY);
  wb_motor_set_position(right_motor, INFINITY);
  wb_motor_set_velocity(left_motor, 0.0);
  wb_motor_set_velocity(right_motor, 0.0);
  wb_motor_set_position(servo, 0.0);

  left_position_sensor = wb_robot_get_device("left wheel sensor");
  right_position_sensor = wb_robot_get_device("right wheel sensor");
  wb_position_sensor_enable(left_position_sensor, get_time_step());
  wb_position_sensor_enable(right_position_sensor, get_time_step());
  wb_distance_sensor_enable(lidar, get_time_step());
}

static bool is_there_a_collision_at_left() {
  return (wb_touch_sensor_get_value(bumpers[BUMPER_LEFT]) != 0.0);
}

static bool is_there_a_collision_at_right() {
  return (wb_touch_sensor_get_value(bumpers[BUMPER_RIGHT]) != 0.0);
}

static void fflush_ir_receiver() {
  while (wb_receiver_get_queue_length(receiver) > 0)
    wb_receiver_next_packet(receiver);
}

static bool is_there_a_virtual_wall() {
  return (wb_receiver_get_queue_length(receiver) > 0);
}

static bool is_there_a_cliff_at_left() {
  return (wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_LEFT]) < 100.0 ||
          wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_FRONT_LEFT]) < 100.0);
}

static bool is_there_a_cliff_at_right() {
  return (wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_RIGHT]) < 100.0 ||
          wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_FRONT_RIGHT]) < 100.0);
}

static bool is_there_a_cliff_at_front() {
  return (wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_FRONT_LEFT]) < 100.0 ||
          wb_distance_sensor_get_value(cliff_sensors[CLIFF_SENSOR_FRONT_RIGHT]) < 100.0);
}

static void passive_wait(double sec) {
  double start_time = wb_robot_get_time();
  do {
    step();
  } while (start_time + sec > wb_robot_get_time());
}

static double randdouble() {
  return rand() / ((double)RAND_MAX + 1);
}
```

```c
static void moveForward(int spd) {
  wb_motor_set_velocity(left_motor, spd);
  wb_motor_set_velocity(right_motor, spd);
}

static void moveBackward(int spd) {
  wb_motor_set_velocity(left_motor, -spd);
  wb_motor_set_velocity(right_motor, -spd);
}

static void stop() {
  wb_motor_set_velocity(left_motor, ZERO_SPEED);
  wb_motor_set_velocity(right_motor, ZERO_SPEED);
}

static void turn(int left, int right) {
  // stop();
  // step();
  wb_motor_set_velocity(left_motor, left);
  wb_motor_set_velocity(right_motor, right);
  // step();
}

double deg_2_rad(double deg){
  return deg * 3.14159 / 180;
}

static void turn_servo(double angle) {
  angle = deg_2_rad(angle);
  if(angle > 1.5708){
    angle = 1.5708;
  }
  else if(angle < -1.5708){
    angle = -1.5708;
  }
  wb_motor_set_position(servo, angle);
}

static double read_lidar() {
  passive_wait(.00001);
  return wb_distance_sensor_get_value(lidar) * 10 / 1000;
}

static bool contains(int arr[], int value) {
  int i;
  for(i = 0; i < 4; i++) {
    if(arr[i] == value) return true;
  }
  return false;
}

/* main */
int main(int argc, char **argv) {
  printf("Leader Controller started...\n");
  wb_robot_init();
  wb_keyboard_enable(get_time_step());
  step();
  init_devices();
  srand(time(NULL));
  int rando = 0;
  wb_led_set(leds[LED_ON], true);
  passive_wait(0.5);

  int speed;

  int keys[4] = { -1, -1, -1 -1 };
  bool run = true;
  while(run){
    keys[0] = wb_keyboard_get_key();
    keys[1] = wb_keyboard_get_key();
    keys[2] = wb_keyboard_get_key();
    keys[3] = wb_keyboard_get_key();
    // printf("%d %d %d %d\n", keys[0], keys[1], keys[2], keys[3]);
    // press q to quit
    if(contains(keys, 80)) run = false;
    // move forward (w)
    else if(contains(keys, 87)) {
      if(contains(keys, 68))
        turn(THREE_SPEED, FIVE_SPEED);  // turn forward right (w+d)
      else if(contains(keys, 65))
        turn(FIVE_SPEED, THREE_SPEED);  // turn forward left (w+a)
      else moveForward(FIVE_SPEED);  // else move forward
    }
    // move backward (s)
    else if(contains(keys, 83)) {
      if(contains(keys, 68)) turn(-8, -16);       // if turning right
      else if(contains(keys, 65)) turn(-16, -8);  // if turning left
      else moveBackward(FIVE_SPEED);                      // else move forward
    }
    // right  in place
    else if(contains(keys, 68)) {
      turn(-6, 6);
    }
    // left in place
    else if(contains(keys, 65)) {
      turn(6, -6);
    }
    // no keys are pressed
    else if(keys[0] == -1 && keys[1] == -1 && keys[2] == -1 && keys[3] == -1) {
      stop();
    }
    step();
  }

  wb_keyboard_disable();
  wb_robot_cleanup();

  return 0;
}
```

Left Follower Controller Code

```
/*
 * Description:  Controller for left follower robot
 */

/* include headers */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <webots/distance_sensor.h>
#include <webots/led.h>
#include <webots/motor.h>
#include <webots/position_sensor.h>
#include <webots/receiver.h>
#include <webots/robot.h>
#include <webots/touch_sensor.h>

/* device stuff */
#define BUMPERS_NUMBER 2
#define BUMPER_LEFT 0
#define BUMPER_RIGHT 1
static WbDeviceTag bumpers[BUMPERS_NUMBER];
static const char *bumpers_name[BUMPERS_NUMBER] = {"bumper_left", "bumper_right"};

#define CLIFF_SENSORS_NUMBER 4
#define CLIFF_SENSOR_LEFT 0
#define CLIFF_SENSOR_FRONT_LEFT 1
#define CLIFF_SENSOR_FRONT_RIGHT 2
#define CLIFF_SENSOR_RIGHT 3
static WbDeviceTag cliff_sensors[CLIFF_SENSORS_NUMBER];
static const char *cliff_sensors_name[CLIFF_SENSORS_NUMBER] = {"cliff_left", "cliff_front_left",
"cliff_front_right",
                                                                "cliff_right"};

#define LEDS_NUMBER 3
#define LED_ON 0
#define LED_PLAY 1
#define LED_STEP 2
static WbDeviceTag leds[LEDS_NUMBER];
static const char *leds_name[LEDS_NUMBER] = {"led_on", "led_play", "led_step"};

static WbDeviceTag receiver;
static const char *receiver_name = "receiver";

WbDeviceTag left_motor, right_motor, left_position_sensor, right_position_sensor, servo, lidar;

/* Misc Stuff */
#define MAX_SPEED 8
#define NULL_SPEED 0
#define HALF_SPEED 8
#define MIN_SPEED -16
#define CENTER_ANGLE -30
#define ANGLE_RANGE 60
#define TARGET_DISTANCE 0.6
#define ST_CENTER_SPEED 20
#define ST_FACTOR_1 10
#define ST_FACTOR_2 2
#define TN_CENTER_SPEED 8
#define TN_FACTOR_1 1
#define TN_FACTOR_2 1
#define MIN_SAFE_ANGLE -24
#define MAX_SAFE_ANGLE -36

#define WHEEL_RADIUS 0.031
#define AXLE_LENGTH 0.271756
#define ENCODER_RESOLUTION 507.9188

/* helper functions */
static int get_time_step() {
  static int time_step = -1;
  if (time_step == -1)
    time_step = (int)wb_robot_get_basic_time_step();
  return time_step;
}

static void step() {
  if (wb_robot_step(get_time_step()) == -1) {
    wb_robot_cleanup();
    exit(EXIT_SUCCESS);
  }
}

static void init_devices() {
  int i;

  receiver = wb_robot_get_device(receiver_name);
  wb_receiver_enable(receiver, get_time_step());

  for (i = 0; i < LEDS_NUMBER; i++)
    leds[i] = wb_robot_get_device(leds_name[i]);

  for (i = 0; i < BUMPERS_NUMBER; i++) {
    bumpers[i] = wb_robot_get_device(bumpers_name[i]);
    wb_touch_sensor_enable(bumpers[i], get_time_step());
  }

  for (i = 0; i < CLIFF_SENSORS_NUMBER; i++) {
    cliff_sensors[i] = wb_robot_get_device(cliff_sensors_name[i]);
    wb_distance_sensor_enable(cliff_sensors[i], get_time_step());
  }

  left_motor = wb_robot_get_device("left wheel motor");
  right_motor = wb_robot_get_device("right wheel motor");
  servo = wb_robot_get_device("servo motor 1");
  lidar = wb_robot_get_device("lidar sensor 1");
  wb_motor_set_position(left_motor, INFINITY);
  wb_motor_set_position(right_motor, INFINITY);
  wb_motor_set_velocity(left_motor, 0.0);
  wb_motor_set_velocity(right_motor, 0.0);
  wb_motor_set_position(servo, 0.0);

  left_position_sensor = wb_robot_get_device("left wheel sensor");
  right_position_sensor = wb_robot_get_device("right wheel sensor");
  wb_position_sensor_enable(left_position_sensor, get_time_step());
  wb_position_sensor_enable(right_position_sensor, get_time_step());
  wb_distance_sensor_enable(lidar, get_time_step());
}

static void fflush_ir_receiver() {
  while (wb_receiver_get_queue_length(receiver) > 0)
    wb_receiver_next_packet(receiver);
}

static void stop() {
  wb_motor_set_velocity(left_motor, -NULL_SPEED);
  wb_motor_set_velocity(right_motor, -NULL_SPEED);
}

static void passive_wait(double sec) {
  double start_time = wb_robot_get_time();
  do {
    step();
  } while (start_time + sec > wb_robot_get_time());
}

static void turnLeftM() {
  wb_motor_set_velocity(left_motor, -6);
  wb_motor_set_velocity(right_motor, 6);
}
static void turnRightM() {
  wb_motor_set_velocity(left_motor, 8);
  wb_motor_set_velocity(right_motor, -8);
}

static void go_forward_proportional(double distance){
  double speedScale, speed;
  if(distance > 0.6){
    speedScale = (distance - TARGET_DISTANCE) / TARGET_DISTANCE;
    speed = ST_CENTER_SPEED * speedScale * (ST_FACTOR_1 + ST_FACTOR_2 * speedScale);
  }
  else{
    speedScale = (distance - TARGET_DISTANCE) / TARGET_DISTANCE;
    speed = ST_CENTER_SPEED * speedScale * (ST_FACTOR_1 - ST_FACTOR_2 * speedScale);
  }

  if(speed > MAX_SPEED) speed = MAX_SPEED;
  if(speed < MIN_SPEED) speed = MIN_SPEED;
  wb_motor_set_velocity(left_motor, speed);
  wb_motor_set_velocity(right_motor, speed);
}

static void process_movement(double distance, double angle){
  if(distance < 1){
    if(angle > MIN_SAFE_ANGLE){
      turnRightM(); //turn left if too far right
    }
    else if(angle < MAX_SAFE_ANGLE){
      turnLeftM(); //turn right if too far left
    }
    else if(distance > TARGET_DISTANCE + 0.02 ||  distance < TARGET_DISTANCE - 0.02){
      go_forward_proportional(distance); //go forward proportional to distance from leader
    }
    else if(distance <= TARGET_DISTANCE + 0.02 ||  distance >= TARGET_DISTANCE - 0.02){
      stop(); //stop if within margin of safety
    }
  }
}

double deg_2_rad(double deg){
  return deg * 3.14159 / 180;
}

static void turn_servo(double angle) {
  angle = deg_2_rad(angle);
  if(angle > 1.5708){
    angle = 1.5708;
  }
  else if(angle < -1.5708){
    angle = -1.5708;
  }
  wb_motor_set_position(servo, angle);
}

static double read_lidar() {
  return (wb_distance_sensor_get_value(lidar) * 10 / 1000);
}

/* main */
int main(int argc, char **argv) {
  wb_robot_init();

  printf("Follower Controller 1 started...\n");

  init_devices();
  srand(time(NULL));

  wb_led_set(leds[LED_ON], true);
  passive_wait(0.5);

  double angle = -50;
  int turnDir = 1;
  double deltaAngle = 1;
  double minDist = 1000;
  double temp = 0;
  int isTurn = 1;
  double maxDist = 0;
  int counter = 0;
  int foundLeaderFlag = 0;
  turn_servo(angle);
  while (true) {
   isTurn = 1;
   temp = 0;
   while(isTurn){
      if(turnDir == 0){//determines server movement direction
        temp = read_lidar();
        if(temp < minDist) minDist = temp;
        if(temp < 1 && temp > maxDist) maxDist = temp;
        process_movement(temp, angle); //decide movement from sensor data
        angle -= deltaAngle;//update current angle
        turn_servo(angle);//turn the servo

        if(temp < 1 && !foundLeaderFlag) foundLeaderFlag = 1; //indicates leading edge of leader has been
found
        else if(temp > 1 && foundLeaderFlag){ //skips to next sweep if trailing edge of leader has been
found
```

```
          turnDir = 1;
          isTurn = 0;
          foundLeaderFlag = 0;
        }
      }
      else
      {
        temp = read_lidar();
        if(temp < minDist) minDist = temp;
        if(temp < 1 && temp > maxDist) maxDist = temp;
        process_movement(temp, angle); //decide movement from sensor data
        angle += deltaAngle; //update current angle
        turn_servo(angle); //turn servo

        if(temp < 1 && !foundLeaderFlag) foundLeaderFlag = 1; //indicates leading edge of leader has been
found
        else if(temp > 1 && foundLeaderFlag){ //skips to next sweep if trailing edge of leader has been
found
          turnDir = 0;
          isTurn = 0;
          foundLeaderFlag = 0;
        }
      }
      if(angle >= CENTER_ANGLE + ANGLE_RANGE) { //maximum angle for unlocked scan sweep
        turnDir = 0;
        isTurn = 0;
        foundLeaderFlag = 0;
      }
      if(angle <= CENTER_ANGLE - ANGLE_RANGE) { //maximum angle for unlocked scan sweep
        turnDir = 1;
        isTurn = 0;
        foundLeaderFlag = 0;
      }
      counter++;
      if(counter > 999) { //printout for error margin measurement
        counter = 0;
        printf("Max dist Left Bot: %f\nMin dist Left Bot: %f\n",maxDist,minDist);
        minDist = 1000;
        maxDist = 0;
      }
      fflush_ir_receiver();
      step();
    }
  }

  return EXIT_SUCCESS;
}
```

Right Follower Controller Code

```c
/*
 * Description:  Controller for right follower robot
 */

/* include headers */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <webots/distance_sensor.h>
#include <webots/led.h>
#include <webots/motor.h>
#include <webots/position_sensor.h>
#include <webots/receiver.h>
#include <webots/robot.h>
#include <webots/touch_sensor.h>

/* device stuff */
#define BUMPERS_NUMBER 2
#define BUMPER_LEFT 0
#define BUMPER_RIGHT 1
static WbDeviceTag bumpers[BUMPERS_NUMBER];
static const char *bumpers_name[BUMPERS_NUMBER] = {"bumper_left", "bumper_right"};

#define CLIFF_SENSORS_NUMBER 4
#define CLIFF_SENSOR_LEFT 0
#define CLIFF_SENSOR_FRONT_LEFT 1
#define CLIFF_SENSOR_FRONT_RIGHT 2
#define CLIFF_SENSOR_RIGHT 3
static WbDeviceTag cliff_sensors[CLIFF_SENSORS_NUMBER];
static const char *cliff_sensors_name[CLIFF_SENSORS_NUMBER] = {"cliff_left", "cliff_front_left",
"cliff_front_right",
                                                                "cliff_right"};

#define LEDS_NUMBER 3
#define LED_ON 0
#define LED_PLAY 1
#define LED_STEP 2
static WbDeviceTag leds[LEDS_NUMBER];
static const char *leds_name[LEDS_NUMBER] = {"led_on", "led_play", "led_step"};

static WbDeviceTag receiver;
static const char *receiver_name = "receiver";

WbDeviceTag left_motor, right_motor, left_position_sensor, right_position_sensor, servo, lidar;

/* Misc Stuff */
#define MAX_SPEED 8
#define NULL_SPEED 0
#define HALF_SPEED 8
#define MIN_SPEED -16
#define CENTER_ANGLE 30
#define ANGLE_RANGE 60
#define TARGET_DISTANCE 0.6
#define ST_CENTER_SPEED 20
#define ST_FACTOR_1 10
#define ST_FACTOR_2 2
#define TN_CENTER_SPEED 8
#define TN_FACTOR_1 1
#define TN_FACTOR_2 1
#define MIN_SAFE_ANGLE 36
#define MAX_SAFE_ANGLE 24

#define WHEEL_RADIUS 0.031
#define AXLE_LENGTH 0.271756
#define ENCODER_RESOLUTION 507.9188

/* helper functions */
static int get_time_step() {
  static int time_step = -1;
  if (time_step == -1)
    time_step = (int)wb_robot_get_basic_time_step();
  return time_step;
}

static void step() {
  if (wb_robot_step(get_time_step()) == -1) {
    wb_robot_cleanup();
    exit(EXIT_SUCCESS);
  }
}

static void init_devices() {
  int i;

  receiver = wb_robot_get_device(receiver_name);
  wb_receiver_enable(receiver, get_time_step());

  for (i = 0; i < LEDS_NUMBER; i++)
    leds[i] = wb_robot_get_device(leds_name[i]);

  for (i = 0; i < BUMPERS_NUMBER; i++) {
    bumpers[i] = wb_robot_get_device(bumpers_name[i]);
    wb_touch_sensor_enable(bumpers[i], get_time_step());
  }

  for (i = 0; i < CLIFF_SENSORS_NUMBER; i++) {
    cliff_sensors[i] = wb_robot_get_device(cliff_sensors_name[i]);
    wb_distance_sensor_enable(cliff_sensors[i], get_time_step());
  }

  left_motor = wb_robot_get_device("left wheel motor");
  right_motor = wb_robot_get_device("right wheel motor");
  servo = wb_robot_get_device("servo motor 1");
  lidar = wb_robot_get_device("lidar sensor 1");
  wb_motor_set_position(left_motor, INFINITY);
  wb_motor_set_position(right_motor, INFINITY);
  wb_motor_set_velocity(left_motor, 0.0);
  wb_motor_set_velocity(right_motor, 0.0);
  wb_motor_set_position(servo, 0.0);

  left_position_sensor = wb_robot_get_device("left wheel sensor");
  right_position_sensor = wb_robot_get_device("right wheel sensor");
  wb_position_sensor_enable(left_position_sensor, get_time_step());
  wb_position_sensor_enable(right_position_sensor, get_time_step());
  wb_distance_sensor_enable(lidar, get_time_step());
}

static void fflush_ir_receiver() {
  while (wb_receiver_get_queue_length(receiver) > 0)
    wb_receiver_next_packet(receiver);
}

static void stop() {
  wb_motor_set_velocity(left_motor, -NULL_SPEED);
  wb_motor_set_velocity(right_motor, -NULL_SPEED);
}

static void passive_wait(double sec) {
  double start_time = wb_robot_get_time();
  do {
    step();
  } while (start_time + sec > wb_robot_get_time());
}

static void turnLeftM() {
  wb_motor_set_velocity(left_motor, -6);
  wb_motor_set_velocity(right_motor, 6);
}
static void turnRightM() {
  wb_motor_set_velocity(left_motor, 8);
  wb_motor_set_velocity(right_motor, -8);
}

static void go_forward_proportional(double distance){
  double speedScale, speed;
  if(distance > 0.6){
    speedScale = (distance - TARGET_DISTANCE) / TARGET_DISTANCE;
    speed = ST_CENTER_SPEED * speedScale * (ST_FACTOR_1 + ST_FACTOR_2 * speedScale);
  }
  else{
    speedScale = (distance - TARGET_DISTANCE) / TARGET_DISTANCE;
    speed = ST_CENTER_SPEED * speedScale * (ST_FACTOR_1 - ST_FACTOR_2 * speedScale);
  }

  if(speed > MAX_SPEED) speed = MAX_SPEED;
  if(speed < MIN_SPEED) speed = MIN_SPEED;
  wb_motor_set_velocity(left_motor, speed);
  wb_motor_set_velocity(right_motor, speed);
}

static void process_movement(double distance, double angle){
  if(distance < 1){
    if(angle < MAX_SAFE_ANGLE){
      turnLeftM(); //turn left if too far right
    }
    else if(angle > MIN_SAFE_ANGLE){
      turnRightM(); //turn right if too far left
    }
    else if(distance > TARGET_DISTANCE + 0.02 ||  distance < TARGET_DISTANCE - 0.02){
      go_forward_proportional(distance); //go forward/backward proportional to distance from leader
    }
    else if(distance <= TARGET_DISTANCE + 0.02 ||  distance >= TARGET_DISTANCE - 0.02){
      stop(); //stop if within margin of safety
    }
  }
}

double deg_2_rad(double deg){
  return deg * 3.14159 / 180;
}

static void turn_servo(double angle) {
  angle = deg_2_rad(angle);
  if(angle > 1.5708){
    angle = 1.5708;
  }
  else if(angle < -1.5708){
    angle = -1.5708;
  }
  wb_motor_set_position(servo, angle);
}

static double read_lidar() {
  return (wb_distance_sensor_get_value(lidar) * 10 / 1000);
}

/* main */
int main(int argc, char **argv) {
  wb_robot_init();

  printf("Follower Controller 2 started...\n");

  init_devices();
  srand(time(NULL));

  wb_led_set(leds[LED_ON], true);
  passive_wait(0.5);

  double angle = -50;
  int turnDir = 1;
  double deltaAngle = 1;
  double minDist = 1000;
  double temp = 0;
  int isTurn = 1;
  double maxDist = 0;
  int counter = 0;
  int foundLeaderFlag = 0;
  turn_servo(angle);
  while(true){
   isTurn = 1;
   temp = 0;
   while(isTurn){
      if(turnDir == 0){//determines server movement direction
        temp = read_lidar();
        if(temp < minDist) minDist = temp;
        if(temp < 1 && temp > maxDist) maxDist = temp;
        process_movement(temp, angle); //decide movement from sensor data
        angle -= deltaAngle;//update current angle
        turn_servo(angle);//turn the servo

        if(temp < 1 && !foundLeaderFlag) foundLeaderFlag = 1; //indicates leading edge of leader has been
found
        else if(temp > 1 && foundLeaderFlag){ //skips to next sweep if trailing edge of leader has been
found
```

```
          turnDir = 1;
          isTurn = 0;
          foundLeaderFlag = 0;
        }
      }
      else{
        temp = read_lidar();
        if(temp < minDist) minDist = temp;
        if(temp < 1 && temp > maxDist) maxDist = temp;
        process_movement(temp, angle); //decide movement from sensor data
        angle += deltaAngle;//update current angle
        turn_servo(angle);//turn servo

        if(temp < 1 && !foundLeaderFlag) foundLeaderFlag = 1; //indicates leading edge of leader has been
found
        else if(temp > 1 && foundLeaderFlag){ //skips to next sweep if trailing edge of leader has been
found
          turnDir = 0;
          isTurn = 0;
          foundLeaderFlag = 0;
        }
      }
      if(angle >= CENTER_ANGLE + ANGLE_RANGE) { //maximum angle for scan sweep
        turnDir = 0;
        isTurn = 0;
        foundLeaderFlag = 0;
      }
      if(angle <= CENTER_ANGLE - ANGLE_RANGE) { //minimum angle for scan sweep
        turnDir = 1;
        isTurn = 0;
        foundLeaderFlag = 0;
      }
      counter++;
      if(counter > 999){ //printout for error margin measurement
        counter = 0;
        printf("Max dist Right Bot: %f\nMin dist Right Bot: %f\n",maxDist,minDist);
        maxDist = 0;
        minDist = 1000;
      }
      fflush_ir_receiver();
      step();
    }
  }

  return EXIT_SUCCESS;
}
```

Music Parser Code

```
# include <stdio.h>
# include <string.h>

char decipherNote(char note[256]);
int decipherDuration(char duration[256]);

int main( )
{
 FILE *file;
 FILE *movementFile;
 char * line = NULL;
 ssize_t read;
 char songTitle[30];

 printf("Enter in song name: \n");
scanf("%s", songTitle);
int i;

for(i = 0; i < 30; i++){
    if(songTitle[i] == NULL){
        songTitle[i] = '.';
        songTitle[i + 1] = 't';
        songTitle[i + 2] = 'x';
        songTitle[i + 3] = 't';
        songTitle[i + 4] = '\0';
        break;
    }
}

file = fopen(songTitle, "r");

if(file == NULL){
         printf("%s does not exist!\n", songTitle);
}
else{
         printf("Song found! Will now parse notes from song!\n");
         movementFile = fopen("movements.txt", "w+");
         char note[256] = {0};
         char duration[256] = {0};
  while (fgets(note, 256, file))
    {
                  // if(note[0] == 'A' || note[0] == 'B' || note[0] == 'C'){
                           // fprintf(movementFile, "\nf");
                  // }
                  // else if(note[0] == 'D' || note[0] == 'E'){
                           // fprintf(movementFile, "\nr");
                           // }
                  // else if(note[0] == 'F' || note[0] == 'G'){
                           // fprintf(movementFile, "\nl");

                  // }
                  char movement = decipherNote(note);
                  fgets(duration, 256, file);
                  int length = decipherDuration(duration);
                  fprintf(movementFile, "\n%c", movement);
                  for(i = 1; i < length; i++){
                           fprintf(movementFile, "\n.");
                  }
    }
         fclose(file);
         fclose(movementFile);
}
return 0;
}
char decipherNote(char note[256])
{
         if(note[0] == 'A' || note[0] == 'a'){
                           return 'b';
                  }
                  else if(note[0] == 'B' || note[0] == 'b')
                  {
                           return 'r';
                  }
                  else if(note[0] == 'C' || note[0] == 'c')
                  {
                           return 'l';
                  }
                  else if(note[0] == 'D' || note[0] == 'd'){
                        return 'R';
                        }
                else if(note[0] == 'E' || note[0] == 'e'){
                        return 'L';
                }
                else if(note[0] == 'F' || note[0] == 'f')
                {
                        return 'f';
                }
                else
                {
                        return 's';
                }
}
int decipherDuration(char duration[256]){
        int length;
        sscanf(duration, "%d", &length);

        return length;
}
```