PLTW COMPUTER SCIENCE

Activity 2.1.2

Coding Fundamentals: Dead Reckoning

goals

- Apply coding fundamentals to devices that move in the real world
- Translate block-based code to text-based code to learn about syntax
- Get started with VEX Coding Studio
- Develop programs collaboratively to identify what abstracted concepts in block-based languages look like in a text-based programming language

description of task

Create a program to navigate a well-defined environment with limited use of sensors

essential questions

- What challenges are there in programming a vehicle to navigate an environment even if the environment is well defined?
- Why are different languages sometimes better suited for expressing different algorithms?
- How are abstractions managing complexity in a program?

essential Concepts

- Broadcasts
- Programming Language Abstraction
- Algorithms, Variables, Arguments, Procedures, Operators, Data Types, Logic, Loops, and Strings

Materials

- VEX Coding Studio
- Self-driving vehicle
- Grid mat
- Ruler

Plan the Path

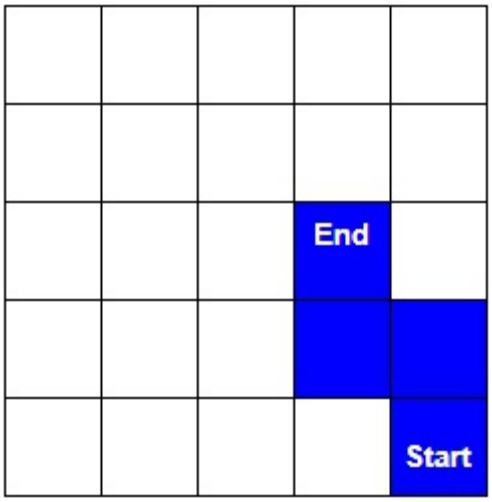
Navigating from one place to another might be an easy task for a person, but for a self-driving vehicle (SDV) each motion must be considered. In this activity you'll use dead reckoning navigation and algorithmic thinking to code a set sequence of motions to maneuver a self-driving vehicle along a specified path.

Dead reckoning is a navigational method used by SDV only in the absence of data from sensors. While dead reckoning navigation does not benefit from the collection of data from the environment, it allows for the creation of low-cost SDV that are not dependent on sensors.

For dead reckoning navigation, the vehicle's path, from a starting point to a target location, must be planned in advance. An SDV without sensors relies solely on a carefully coded program that considers the distance traveled, timing, and necessary turns along the path.

To create your SDV's program, you'll work in a team of three to determine the distance your vehicle will need to travel to navigate a specified path on a grid mat. Your SDV's rear wheels each have a circumference of about 6.22 inches or 15.80 centimeters. This is also an estimate of the distance your SDV can travel straight forward with one revolution or turn of the wheels.

1. Use the path shown in blue to plan out the sequence of directions your SDV will need, to navigate it on the grid mat.



2. Use a ruler to measure the distance, in inches, that the SDV's rear wheels will need to travel forward from the starting square to end within the second square.
3. Use the circumference of the rear wheels to determine how many times a rear wheel will need to make a complete revolution to move forward one square.

PLTW DEVELOPER'S JOURNAL Record your measurements and the number of rear wheel revolutions needed to move the vehicle forward a distance of one square on the grid.

4. For any 90-degree turns in your path, consider how many revolutions, or turns, are required and the turning direction of each wheel as the SDV pivots. Each revolution requires a wheel to rotate 360 degrees.

PLTW DEVELOPER'S JOURNAL In your journal, record your best estimate of each wheel's revolutions, in degrees, required to turn the SDV 90 degrees with an appropriate directional label, forward or reverse.

5. As a team, continue planning out your movement sequence, distance measurements, rear wheel revolutions in degrees, and wheel directions required to complete the path until your SDV is inside the square at the end.

PLTW DEVELOPER'S JOURNAL Record your team's movement sequence plan and refer to it while creating a dead reckoning program for the SDV's path.

Create the Program

In the previous activity, you began using a text-based programming language. You'll begin using a text-based language in the VEX Coding Studio to create your dead reckoning program. Use the Blocks and Text toggle buttons at the top of the programming environment to work in either a blocks-based or text-based language as you code.

With your team, create a dead reckoning program that will navigate your SDV through the blue path highlighted on the grid in the previous Plan the Path section. Use the pair programming technique with a third team member managing the SDV. Rotate roles as indicated by your teacher. Use your navigation plan and notes as you complete the following steps to assist you with your code.

Broadcasting

You'll place the code for each movement (Examples: LEFT, STRAIGHT, RIGHT) of your SDV an event handler. To call on each event that contains code for these movements, you'll use a *broadcastAndWait* command. Include the *broadcastAndWait* commands within a main string of code that calls on each movement as identified in your sequence plan.

6. Create a main string of code that calls on events

using *broadcastAndWait* for each movement identified in your planned sequence.

7. Using a *when* command, create multiple event handlers for each movement your main code calls on. To turn your SDV left and right, you'll use two event handlers, each of which controls one motor.

Each *when* command should be labeled with one of the *broadcastAndWait* movements. These event handlers will serve as functions for each of your SDV's movements.

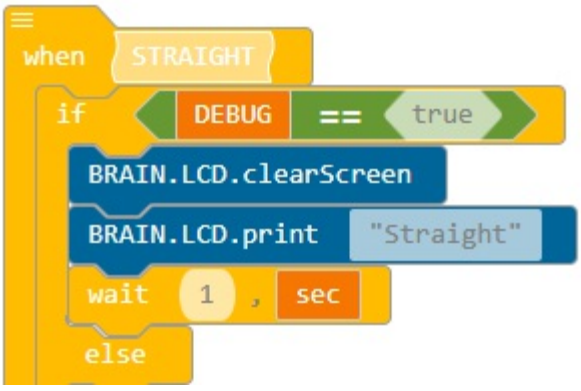
Debugging with Strings

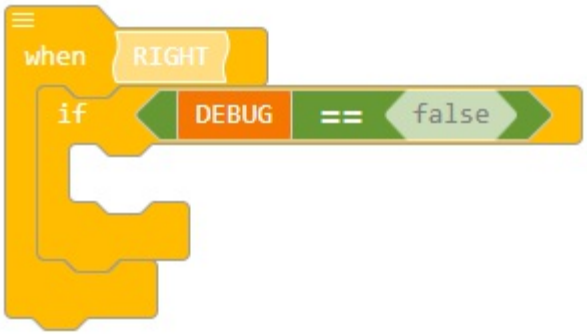
As you continue creating your dead reckoning program, you'll find it useful to get some feedback about what your code will do before you actually run the SDV on the grid. The following steps outline one method of creating such a means of debugging.

8. Create a global variable DEBUG and set its Boolean value to *true*.
9. Add a conditional to each movement event handler to check if DEBUG is set to true.

   Use *if-else* commands for the STRAIGHT, and one LEFT and one RIGHT, event handlers. Use *if* only for the other LEFT and RIGHT event handlers.

10. Set the *if* part of your *if-else* commands so when DEBUG is true, output commands will only be printed on the SVD's LCD screen rather than turning the wheels and actually moving. You may choose to clear the LCD before each printed movement or print the movement command on a new line to show the whole sequence as it's executed.
11. Set the *if* only commands so when DEBUG is false, the program will run the motor commands that you'll create next.



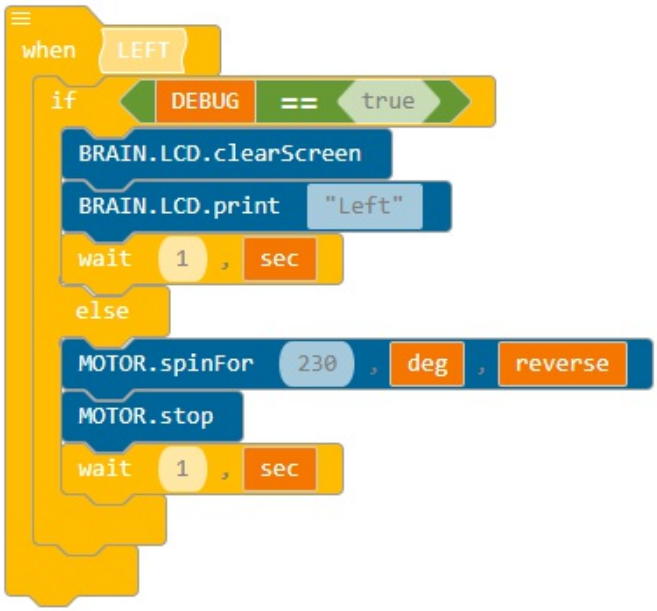


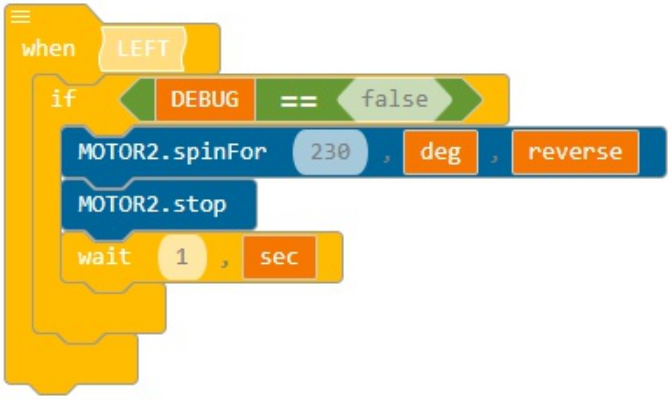
This debugging approach allows you to test your planned sequence for the navigation and order of events in the main code. The *else* part of the conditional will include the code that moves the SDV.

Get Those Motors Spinning

For each movement event code, let's look at the *else* part of your conditional and the movement commands. If the DEBUG variable is FALSE, then the program runs the motors and drives your SDV along the path. Using arguments in the motor commands, you can specify the degrees the wheels turn before the motors stop.

12. Use the *DRIVETRAIN.driveFor* command to move the SDV forward the STRAIGHT event handler.
13. Use your planned path and notes to set the arguments for the number of inches and wheel direction needed for the straight movement.
14. Add a *DRIVETRAIN.stop* and *wait* commands to pause the SDV after it moves and ensure it stops all motors before executing the next motion.
15. Inside each LEFT and RIGHT event handler, use the *MOTOR.spinFor* command.
16. Using your notes on the degrees and direction each wheel turns when the SDV rotates 90 degrees, add arguments to the command.
17. As with the STRAIGHT event handler, add a *MOTOR.stop* and *wait* commands to fully stop the motors before the next procedure in the sequence begins.





Test and Adjust

After your initial program is complete, your team will test it. The team member who is managing the SDV downloads and runs the programs. All team members analyze the program's execution and make necessary code adjustments. The process is iterative and should be repeated as necessary.

18. Set your DEBUG global variable to *true*.
19. Save your program as *A212_Path1_LastNames* and download it to your SDV.
20. Run the program on your SDV and use the printed movement commands on the LCD screen to test its execution.
21. As a team, make any necessary changes to your program based on your testing results.
22. Set your DEBUG global variable to *false*.
23. Save your program and download it to your SDV.
24. Run the program with the SDV on the grid and test its path navigation.

PLTW DEVELOPER'S JOURNAL Record your observations of each test and note any improvement ideas or adjustments you made to the code.

25. As a team, make adjustments to motor speed, direction, and/or revolutions in your code based on your testing results.

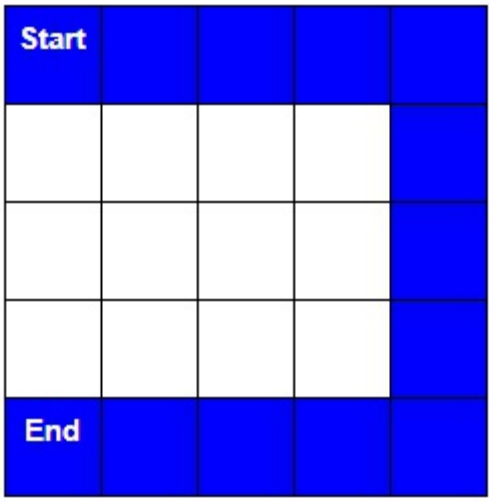
Challenge Paths

Using text-based programming in VEX Coding Studio and additional coding concepts, create two

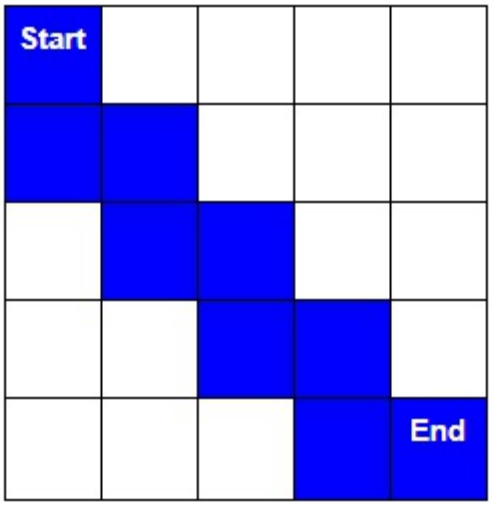
additional programs with your team to navigate your SDV along two new paths.

PLTW DEVELOPER'S JOURNAL Remember to use algorithmic thinking and plan out the movements for each path before coding. Include an event sequence and code ideas as necessary.

26. For the first path, consider using a *while* conditional loop and variables that iterate the necessary SDV movements three times. Save your final program as *A212_Path2_LastNames*.



27. As a team, consider ways to program four iterations of movements to navigate the final path. Save your final program as *A212_Path3_LastNames*.



Conclusion

1. What can impact dead reckoning navigation accuracy?
2. How can you compensate for navigational errors with code?
3. How might dead reckoning navigation be beneficial when programming self-driving vehicles?

4. How did you interpret and respond to the **essential questions**? Capture your thoughts for future conversations.