Line Follower Simulator (LFS)
by Ron Grant

Sample Tricycle Controller
by Will Kuhnle

USER'S GUIDE   Sep 24, 2020

PROPOSED SIMULATOR FOR DPRG

vChallenge Line Following Contest 2020

Replaces pre-Library version of program User's Guide Aug 11

The library code and bundled demo sketches are included on Git Hub at https://github.com/ron-grant/LineFollowerSimLib.

Table of Contents

Introduction

LFS is a Processing library that provides a framework for developing a line following robot controller through real-time simulation.

An image of a line following course is used as the simulated robots world which is imaged with a down looking camera attached to the robot.

Robot camera images are not obscured by the robot allowing for placement of user sensors without any constraints. Of course sensor placement might need to be a consideration when moving to the real world if your robot used wheels, vs floats.

The first model developed for LFS is that of a differential drive robot, e.g. , a two wheeled robot with caster. This model accepts turn rate and forward speed as inputs to move the robot around in the world which is a 2D surface defined by the course image.

Will Kuhnle's included tricycle robot sketch is a variation that uses a steering wheel angle and wheel steering “speed” to move his simulated robot.

Currently DPRG line following courses use (for the most part) a 12 inch square tile as a common size to contain a course feature, e.g. line segment, 90 degree turn, etc.

LFS is informed of the course scale in “dots per inch” (DPI). A suggested scale is 64 DPI. At this scale a each 12” x 12” tile is 768 pixels. The current Challenge course including 6” borders is 7x13 tiles or 5376x9984 pixels. This is a very large bit map.

Sample course images are provided including advanced level features. Also, an image of the DPRG Challenge course is provided, bundled with example sketches in their data sub-folder.

Depending on the sensor view size (default 800x800 pixels), the area imaged by the simulators camera is about 1 square foot (12.5” x 12.5”) where the center of the robot (midpoint between the drive wheels) is placed at the center of the robot viewport. Sensors including spot and line sensors can be defined which sample the screen image and report their data to the controller program which you write.

Note: Other image processing techniques could be applied to the “camera” image.

Processing is capable of specifying a 3D perspective camera if you are interested in simulating a video camera, but for this implementation I avoided adding in this complexity.

LFS presents sensor data at a fixed stepTime (default value 1/60th second) to the user's controller program (java method) which must provide turn rate and forward velocity values which are used by the simulator to update robots position and heading.

It is important to note this simulation step time is independent of realtime. The simulation can be frozen or step as slow as about 1 step per second. Also, if CPU & GPU are fast enough faster than realtime simulation is quite helpful.

Again, the results of the simulation are the same regardless of how fast it is able or allowed to run. Program code runs on the CPU on a single thread. Most of the image display portions are run using graphics processing unit (GPU) on board your video/graphics card.

LFS provides simple keyboard and mouse interface, single key commands and ability look at an overhead view of entire course showing cookie crumb trail of robots path.

Also, the robot can be driven manually by keyboard commands or by user controller which can be enabled / disabled by key command.

The robot can also be interactively positioned and its heading adjusted.

LFS Environment - Processing (Java based IDE)

LFS uses Processing which is available for free download at processing.org.

Processing is java based, but only a basic understanding of java is needed for a user to write a controller of reasonable complexity. The Arduino IDE is modeled after the Processing IDE, so if you have a little experience with Arduino, the Processing environment should present no difficulty.

A processing sketch (program) is composed of methods (functions) you write including a setup() method that is called one time then draw() which is called at video frame rate, e.g. 30 or 60 times per second. Also you can write your own methods and classes which as part of the sketch are actually subclasses of and outer class which has access to many methods provided by Processing and Java language itself.

More experienced java users may wish to create outer classes, which is described in the Processing IDE documentation where you create .java tabs and where you will need to pass a reference of the application instance to the class instance if you need to access Processing methods.

In the case of LFS, you will write a few Java methods and be required to access sensor data with the program skeleton already written and supported by LineFollowerSim library.

You can group your code into multiple “notebook tabs” (.pde files) as is done with LFS project. The code all exists in a single pool “included” into one program at compile time (when you run your sketch).

Processing Help allows you to access information about the Language (many common methods) environment and other libraries.

Processing also includes a debugging mode which can be quite helpful allowing you to set break points and monitor variables.

Processing Tweak mode allows adjusting program constant values defined typically in methods that are called repetitively (not global constants). Sometimes tweak can be troublesome, but worth giving a try.

Few Notes On Java, If You Have C, C++ Background

If you have a C or C++ background, you will note that the low-level program constructs are identical to C, but do use care integer types, they are all signed in java. Class instances and arrays are passed to methods (by reference)

Certainly Google searches can be a great ally, e.g. “how do I declare and array in Java”

Of course the simulator program itself contains examples of simple lists, arrays and some example graphics drawing if you want to add some annotation to your simulation.

System Requirements

LFS does have some reasonably heavy demands on GPU memory. The change course requires the graphics card be able to handle a large texture map, i.e. approximately 5400x10000 pixel image.

Any machine equipped for light gaming should not have a problem.
For example my machine is equipped with a nVidia GTX-1060 works well.

Low-power laptops with minimal GPU hardware might have to struggle.
Smaller course size would help, the full Challenge course image might be a problem.

Simulation Loop

The following 5 steps are executed per one simulation time step.

1. Draw view of course visible in local area of robot as would be seen from above a transparent robot where front of robot is oriented toward top of screen-window and robot center is center of screen-window.
2. Acquire sensor input from user defined sensors from view drawn in step 1.
3. If in course view mode (Tab key toggles on/off), draw course view over top of robot view.
4. Call user controller (java method) which analyzes sensor data and ultimately calculates robot turn rate and forward velocity (or in tricycle mode, wheel angle and velocity).
5. Update robot location (in world coordinates) including x,y location and heading based on fixed time step.

Currently two robot views appear in the simulation. The largest is the robot view drawn for sensor data acquistion. It is drawn at 1:1 scale where typical course is 64 dots per inch. In this case the default 800x800 pixel area covers 12.5” x 12.5” area. This view supports ability to display where sensor data is being sampled from to allow you to verify sensor location. A second scaled down view is drawn overtop the sensor view. There is an option to cover the original sensor view or gray it out. This may change in the future, but early attempts to render this view off screen slowed frame rate.

Details on Robot Position and Heading Calculations

User input is now specified as targetSpeed and targetTurnRate. where the robot is now goverened by acceleration and deceleration rates and Turn acceleration and deceleration rates that are defined by you and subject to simulator maximum values.

Note: Turn deceleration is fixed (made equal) to turn acceleration rate, so is not specified .

All acceleration/deceleration rates are constant values, thus change in speed or turn rate occurs at a constant rate over time.

If while accelerating or decelerating a target value is exceeded for example, the value will be set at that target value.

For Example: Given simulation dt = 0.01 (to keep math simple) default is 0.0166 sec and given accRate = 20 inches/sec^2 (inches/sec/sec)

In one time interval speed would increase by accRate * dt (20 x 0.01 = 0.2 inches/sec)

speed = speed + accRate * dt

If speed was initially 0 it would ramp up at a 0.2 inches/sec for each simulation step. After 10 steps speed would be 2.0 inches/sec.

Thus a target speed could be specified and if speed is less than target speed, the acceleration process continues. If the speed exceeds the target speed, the speed is set to that target speed and the acceleration stops.

If for example target speed was 2.1, on the 11^{th} step the 2.2 value would be clipped to 2.1 and would remain at that value until a new target speed is set.

Similarly if a target speed is less than the current speed the above process would be applied but using set deceleration rate and a subtraction of decelRate*dt from speed until the new lower target speed is reached.

Likewise turn rate is handled the same way, but of course independently in each time step.

With a current speed (and turn rate) in hand, a second numerical integration action is performed for each simulation time step:

The distance d traveled in a given time step is the product of speed (inches/sec) and time in seconds (dt)

d = speed * dt

Thus for a current speed of 2.2 inches per sec with dt = 0.01 sec
total distance traveled would be 0.022 inches.

The actual change in X and Y location on the course depends on robot heading.
At default heading of 0 degrees the robot moves in -X direction so calculating position is very simple. Y remains constant and 0.022 would be subtracted from X every time step.

Relying on some simple trigonometry we can make use of sine and cosine functions to resolve distance traveled into changes in X and Y as a function of heading.

Given heading in degrees = h, current x,y location, update x,y and heading

```
 h = h + turnRate * dt            new heading based on current turn rate

 if h>=360.0) h = h – 360         restrict to 0..360 degrees, causing wrap around
if (h<0) h = h + 360              for example heading -1 becomes 359, 366 becomes 6

 hr = h * PI/180                  convert heading in degrees to radians, for sin,cos

 x = x – cos(hr) * d
 y = y – sin(hr) * d
```

Sensors

Sensors are defined and placed relative to robot using code defined in userInit() method in the userInit tab. Sensors are placed relative to the robot center with offset distances in inches. Positive X is distance in front of center point, positive Y is distance to right of robot center.

The sensors are used to sample the visible region around the robot where the simulator renders the view as would be seen from a camera placed above the robot, looking down, without the robot obscuring the view. (Invisible robot).

Two types of sensors can be defined.

1. Spot Sensor samples a rectangular region of the course and returns a single intensity value ranging from 0.0 ,e.g., black line, to 1.0 white background.

2. Line Sensor is linear array of spot sensors and returns an array of intensity values.

Line sensors now have two available modifiers. setRadius(radiusInches) turns line into 1/2 circle with center at offset specified for sensor center. Also setRotation(degrees) rotates line or 1/2 circle about center. Negative values clockwise, positive counter-clockwise.

Line sensor data is read using readArray() method which returns array of floats ranged from 0.0 (black) to 1.0 (pure white). Spot sensor data is returned as a single float value using read() method.

See Sample controller code for example of how to access sensor data.

Getting Started With LFS

1. Download Processing from www.processing.org website.
2. Play with the processing IDE, there are many good tutorials
3. Get the LFS library code from git hub.

The library code and bundled demo sketches are included on Git Hub at https://github.com/ron-grant/LineFollowerSimLib.

Download the LineFollowerSim-XXX.zip file to your computer.
Extract the LineFollowerSim folder from zip file and copy to the libraries sub-folder of your Sketchbook. To locate your Sketchbook folder, start up Processing and use menu command File->Preferences. The Sketchbook location appears at the top of the Preferences dialog box.

After copying the LineFollowerSim folder to the libraries folder, close down and restart Processing. You should now see the LineFollowerSimulator library after issuing File>Examples menu command. The example sketches will appear when you click the [-] on the LineFollowerSimulator tree entry. Click on any of the examples (two at the moment) to load the sketch and press run button (Circle with triangle).

Note this rendition of the simulator & examples use simple keypress. You may need to click on the window outside the course view or robot view to give the application focus.

For now the command pattern on a contest run is (Press R) to run, space bar to stop, "F" to finish the contest run, including adding or appending to contest.cdf file in sketch data sub-folder.

Also, in sketch folder a screen shot is saved for the "Finished" run named with user and current time of day FirstnameLastnameRobotname-hr-mn-se.png

The demo sketches should be quite useful in seeing how to use LFS.

The LFS library application interface is documented in reference sub-folder of library as html pages generated by javadoc utility.

At present every LFS sketch includes some standard high level code in the main program tab then additional tabs (.pde files) with user code. Later releases of the library are likely to have changed the main tab code, but require little or no change to your code. Therefore if you install an updated library you may need to copy main tab code from an example sketch bundled with the library.

Key Command Summary

LFS accepts keyboard commands when the program has focus. When the program does not have focus, it displays a message:

“Click in application window to give focus for key command response.”

At present you may want to click outside robot and course viewport when clicking to give focus to avoid moving the robot.

Horz. Cursor Arrows   Turn robot when controller OFF
Vertical Cursor Arrows   Change speed of robot when contoller OFF (or with controller enabled if it does not adjust robot speed.

SPACE   Toggle Controller ON/OFF – If Contest Running STOP Contest
S   Stop Robot (turns controller OFF, if ON)

C   Toggle user controller On/Off. When off arrow keys can be used to drive robot.

G   Go – similar to run contest, but intention is to allow modification of robot position with mouse while robot controller is running. “Finish” not available

R   Run Contest. Robot moved to starting position, stopwatch cleared, cookie crumbs are cleared and user controller is enabled.

F   Finish Contest. Contest stats are recorded in contest.cdf file in sketch data folder.

TAB   Toggle between robot view on top and course view on top view.
ALT   Toggle rotation of course view 0 and 90 degrees

Mouse Commands

When simulation is not in run contest mode (Run command issued):

Hold down left mouse button in either robot view or course view and drag to move the robot. Hold down right mouse button with mouse positioned in either view and move horizontally to rotate the robot to a new heading.

Example Robot Sketches Included

LFS_SimpleBot

Simplebot is a simple controller more suitable for the advanced course. It is not designed to handle reverse images, acute angles...

The controller calculates the centroid of a dark area on a line censor with respect to the center of the sensor returning a signed value which indicates the sensor centers offset from the center of the line, which I will call the error.

The the error is in pixel units which at 64DPI, equates to 64ths of an inch per unit.
For Example a value of 4 = 4/64 = 1/16 of an inch.

The error and its first derivative (change since last time step) is used to control robot turn rate implementing a PD controller.

The robot velocity is controlled, optionally by a line of code which sets the robot drive speed to a value related to the inverse of the error. If commented out, you can manually set robot speed with up and down arrows when the controller is enabled.

LFS_TrikeX (where X = revision date)

This sketch implements Will Kuhnle's tricycle robot controller.

His claim at the moment is “The code is a work in progress – not refined, messy..
Will's trike (tricycle) operates like the tricycle you had as a kid, except in reverse.
The steered and driven wheel is at the back and the inputs are wheel angle and wheel velocity. You could replace his controller code with your own while leaving trikeMode = true to provide the trike functionality vs differential drive robot functionality.

An interesting note from Will, and see his code, a given trike wheel angle and speed is easily transformed to a turn rate and speed of a differential drive robot.

Writing A Controller

To write a robot controller you need only access sensor data, and generate target turn rate and target drive speed information. Ultimately see example sketches, but here are a few notes:

In UserInit tab:

1. Load Course Image file which should be located in data subfolder of processing sketch.
2. Define robot instance which sets robot position, heading and initial velocity.
3. Specify course scale in dots per inch if not 64, e.g. courseDPI=32, Most sample courses are 64 DPI.
4. Init Spot and Line Sensors (see example code)

In UserControllerDiffDrive tab controllerDiffDrive(Robot robot, float dt) is called every time step. In this method or methods you call from this method, you will read sensors then ultimately set robot.speed and robot.turn rate variables. In Trike Mode, you set wheel angle.

The simple DiffDrive controller is not suitable for handling reverse images. You might want to look at Will's code which looks at transitions spanning a line sensor to help handle reverse polarity (white on black tiles).

The complexity of the code will depend on what problem you are trying to solve. The challenge course is likely to require a significant coding effort unless you are very clever with sensor placement then your code complexity will be reduced.

If trikeMode set to true, then Will's tricycle code is called or code you replace it with. This is controllerTrike(Robot robot, float dt) called in UserControllerTrike tab.

Description Of LFS Files (IDE Notebook Tabs)

All files contain .pde extension. The tabs prefixed with “User” are the ones you will typically modify or replace with your code.

LFS_XXX Main Program contains LFS skelaton including calls to many high-level library methods. Typically you don't need to modify this method.

It is suggested when you load a demo sketch and decide to modify it, execute a Save As to a meaningful name with LFS_prefix recommended, e.g. LFS_SpeedyBot5

UserInit

Implements userInit() method which is called when LFS starts and typically when when userControllerResetAndRun is called. This method defines contestant name, robot name, course to be used, acceleration deceleration parameters, and sensors including their type (spot or line) and placement. Also, modifiers can be used to turn line sensors into 1/2 circle arc sensors and rotate both line and 1/2 circle sensos.

UserReset Contains method **userControllerResetAndRun** called when robot commanded to start running contest (R-Run contest command key pressed).

Usually userInit method is called from this method to reset/redefine sensors and robot acceleration / deceleration constants.

In addition it is recommended to place code here to reset the state of your controller. If this is not done, sketch will have to be re-started every time you want to start a Run.

UserCon Contains **userControllerUpdate** method which is the heart of your robot controller. Called by LFS upon each timestep after it has made sensor data available. Here you will include code to analyze sensor data, and most likely recognize various patterns in case of Challenge Course. Finally you will update as needed your robots target speed and target turn rate.

```
lfs.setTargetTurnRate(t);
lfs.setTargetSpeed(s);
```

UserDraw Optional overlay drawing method calls. This method userDraw() must be present, but can be empty if you have no need of overlay graphics. Two LFS methods may come in handy.

```
lfs.setupUserDraw();          // sets up transforms origin robot center scale inches
lfs.drawRobotCoordAxes();  // draw robot coordinate axes
```

Standard processing drawing methods can be then called. Keep in mind after lfs.setTargetTurnRate() coordinates are transformed to match robot coordinates with +X extending forward (up) and +Y extending to right.

UserDrawPanel Miscellaneous screen drawing other than course views and robot views drawn by main tab using library code.

UserKey

Keyboard command decoder - single keypress commands for simplicity.
Might consider moving some key decoding into main tab leaving user key decode here, but for now all keyboard command keys decoded in this module.

Other things worth noting.

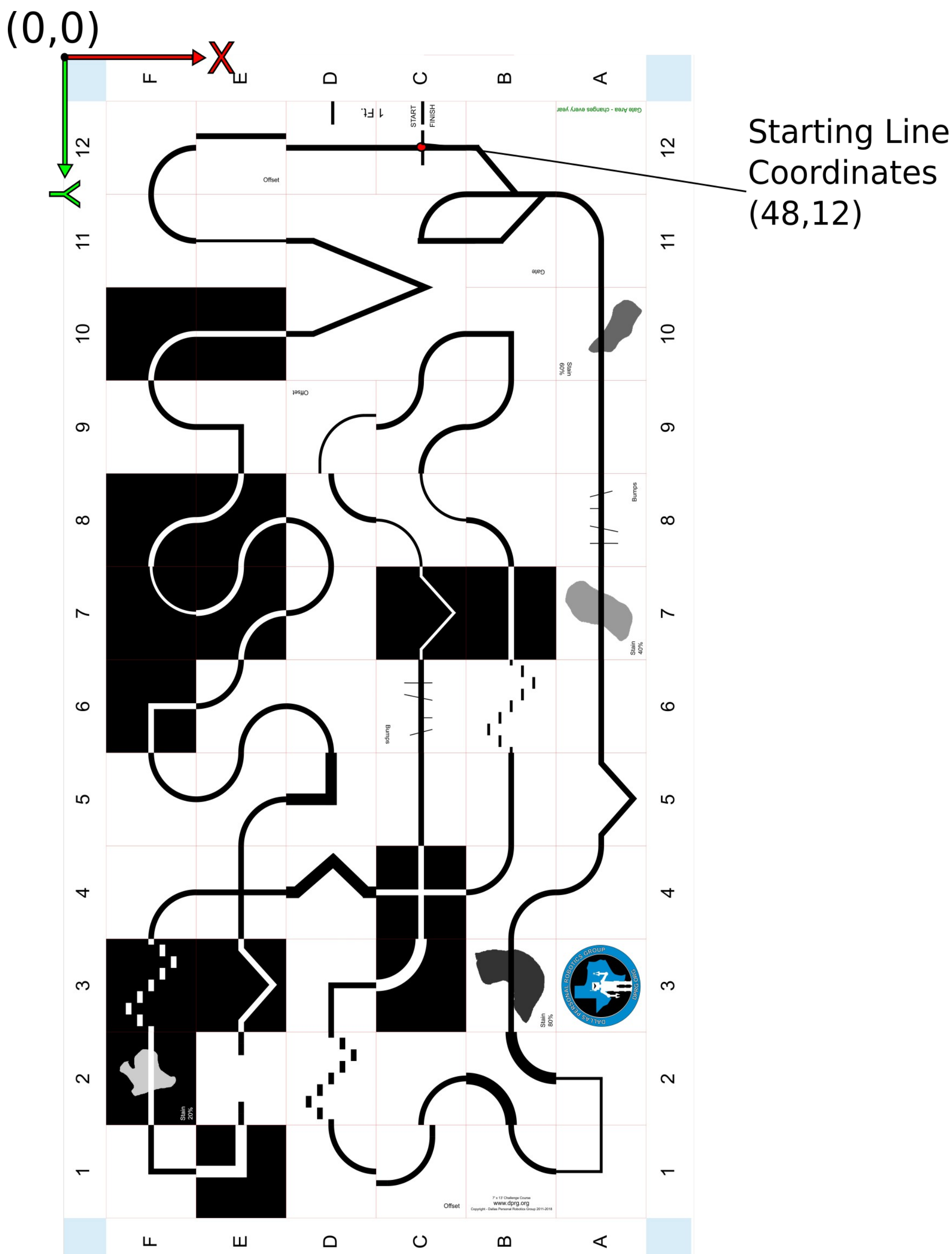
Processing provides console output window which is quite handy when you just want to print stuff.

Also worth consideration is Processing debug mode and also Tweak mode which allows you to alter constants defined within program loop, i.e. anything in draw() or called by draw.

Be sure to locate the API documentation if more attention has not been brought to it. It is located in LineFollowerSim reference sub-folder. index.html
Most all LFS functionality is accessed through a single instance lfs of LFS class and the sensors you define.

LFS World Coordinates

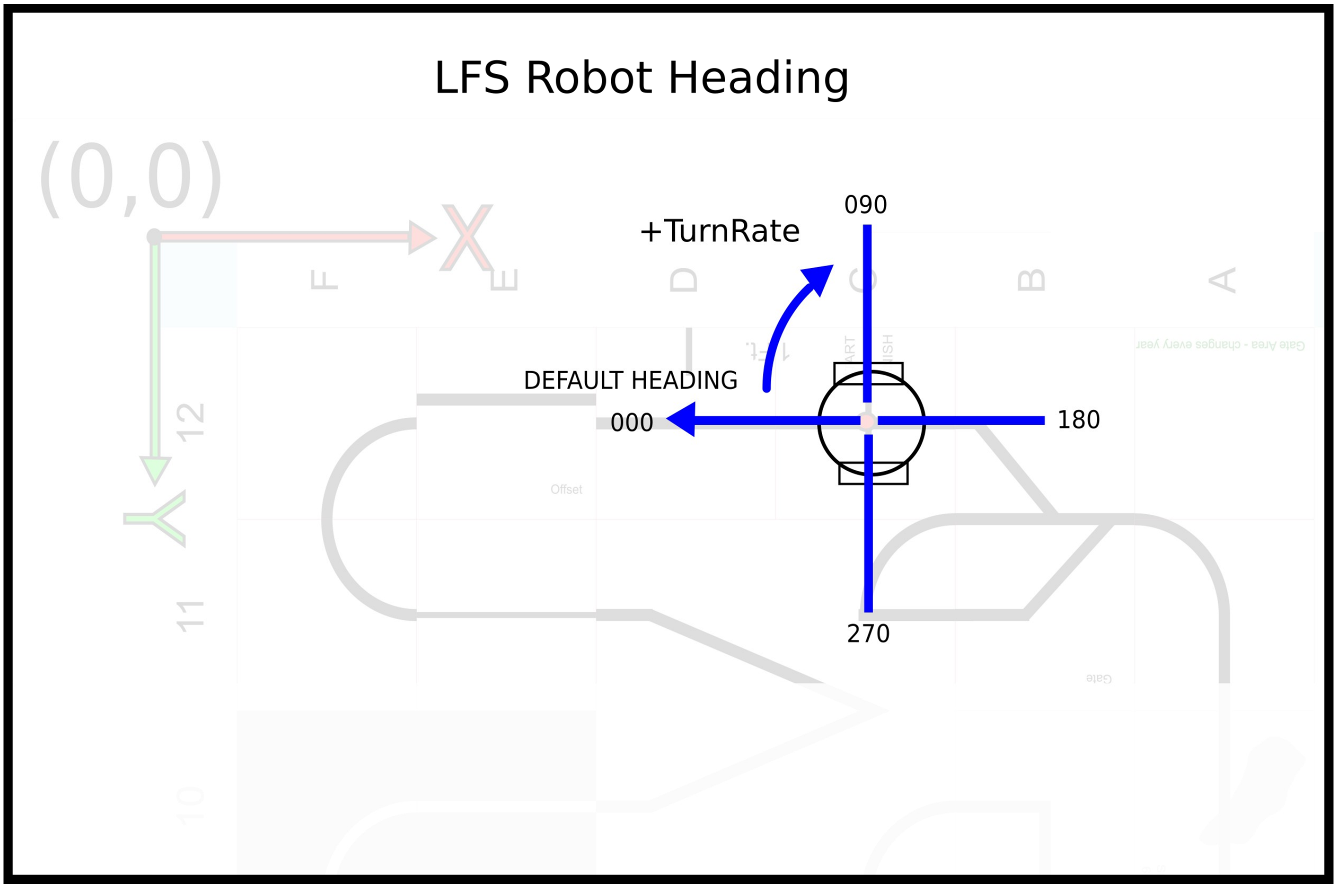
Origin is Course Image Top-Left Corner (DPRG Challenge Course shown here)
Tiles are 12x12 (inches) at 64 dots/inch (DPI) = 768x768 pixels per tile
Borders are 6 inches. Robot is located in world coordinates.



Robot Heading

Robot default heading 0 degrees.
Positive turnRate will turn robot in direction of arc (right turn) with increasing heading, while negative turnRate will turn robot the other direction (left turn) with decreasing heading.

Heading values are confined to the interval 0..359.99,
e.g. decreasing heading crossing 0 (and going negative) has 360 added to its values.
Likewise, increasing heading exceeding 360 (359.99) will have 360 subtracted from its value.

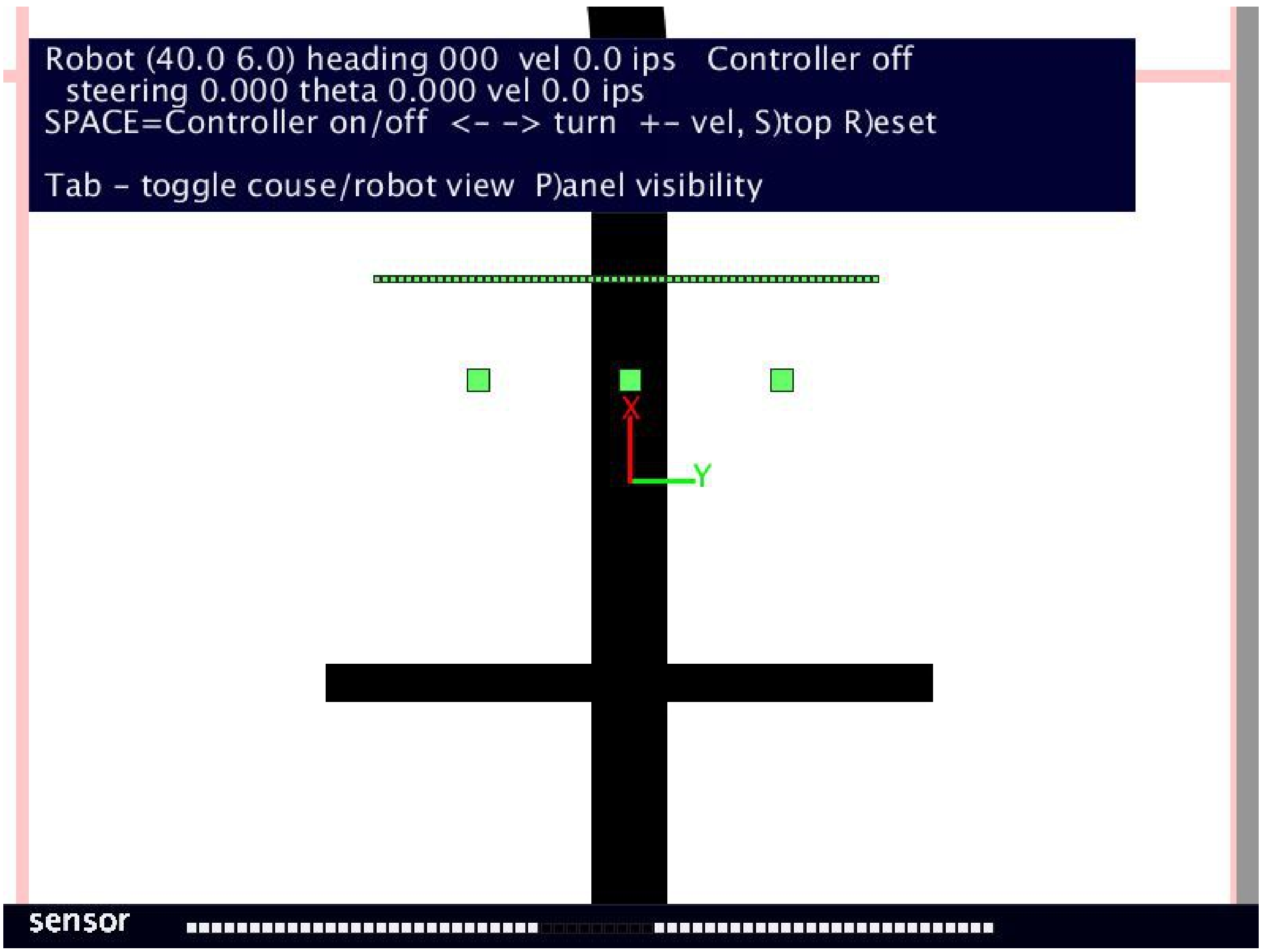


Robot Coordinates / Robot View

Robot origin (0,0) is midpoint between wheels, center of screen in robot view.
Robot position in world is such that robot origin is placed at current world (x,y) location.

The screen image below shows robot overhead view (robot is invisible).
Location is 4 inches forward of position shown in LFS World Coordinates diagram.
Note: In a contest you would start behind the "start/end" line – not in front of it.

Sensors are located in robot coordinates, e.g. , bar sensor location is at (2,0)
Left spot sensors is located at (1,-1.5), center at (1,0) and right at (1.0,1.5)
in inches.



Note: Sensor data shown at the bottom of this screen may not appear or appear differently. Current goal is to create a viewport overlay with sensor data. -

Notes