The Intelligent Robot Simulator

MindRover™

The Europa Project™

Strategy Guide

CogniToy

MindRover: The Europa Project Strategy Guide

By Eric Ellingson
and Jeff Scott

International Standard Book Number 0-9716816-0-0

MindRover: The Europa Project Strategy Guide

Chapter 1

Introduction

MindRover is an exciting robot-simulation strategy game with a growing community. This strategy guide was written to provide a more in-depth view of what you can do with MindRover.

It is assumed that you already have the game “MindRover” (v1.07 or later) installed on your computer and that you know how to use it. If you are experiencing problems installing the game, contact CogniToy by email at support@cognitoy.com or by phone 1-888-788-1792.

Once you have MindRover installed, refer to the provided user manual for basic information on the user interface and how to create rovers. This guide also assumes that you have installed the v1.08 add-on component pack. If you don’t have this pack, you can download it from www.mindrover.com. The add-on pack can also be found on the CD with the MindRover v1.08. Look for the folder labeled "MRExtras".

At the MindRover website, you can discuss robots, strategies, programming and other topics with other robot enthusiasts at the Player forums. You can also download updates and add-ons and find out what’s new with MindRover at the website. There are a number of fan sites that run contests and collect interesting MindRover vehicles.

Chapter 2

Using the Interface

This chapter provides some hints and tips for using the MindRover interface to build robots and wire them up. Once you have logged into the game, you are in the console interface.

The first screen you see is the Scenario Selection Screen. On the left are buttons to get you to the Vehicle Selection, the Component Selection, the Wiring Screen, and finally the 'GO' button to start a competition.

Wiring Screen: Colored wires and the Mode Switch

Colored wires are not just for show. They are an integral part of the mode switch functionality. The mode switch is a component that turns on and off entire sets of wires based on their color. A colored wire will only be active if the mode switch is set to that color. This means that if you don't have a mode switch, ONLY THE GRAY WIRES ARE ACTIVE. For more tips on using the mode switch, see the advanced strategy tips section.

Navigating the Holobox

There are several ways to navigate the holobox. You can simply use the arrows, you can use the dropdown box to jump to a specific component type, or you can click and drag with the RIGHT mouse button to scroll quickly through the whole box.

Wiring Screen: Adjusting the component values

There are three ways to use a mouse to manipulate the value settings for any component. The first is to grab and drag the slider. This allows you to quickly change the setting to the approximate value desired. You may then adjust it further by using one of the following methods:

- If you want to adjust the value in discrete increments (usually in steps of 1.0 or 5), left click on either side of the slider.
- To adjust the value in smaller increments (usually in steps of 0.1 or 1), right click on either side of the slider.

Keyboard Shortcuts

To speed up navigation throughout the game we have added some keyboard shortcuts that cut down on the time that it takes to do certain activities.

In the Holobox:

- Alt G - Go
- Alt S - Scenarios
- Alt V - Vehicles

Within the competition:

- ESC - pop up window
- Alt I - Instant replay
- Alt P - Play again
- Alt C - Continue
- Alt R - Return to console

Others:

- Alt Q - Quit
- Alt C - Credits
- Alt O - Options

Using the Command Console

At the bottom of the design window is the command console. The console displays information about what the game system is currently doing (i.e. when it saves your rover or loads a scenario). The console window also allows you to enter commands. Click inside the frame to get a command cursor. Type a command and press enter to execute it. Type "help" for a list of commands.

A list of commands available is below:

bind [key] [action]

Allows you to run commands while running a scenario. By pressing the [key] the [action] will be excecuted. Ex. "bind q screenshot" allows you to press q in the scenario and have a picture of the screen saved to disk.

clearslots

Clears the vehicles from the slots. Used within a script when running multiple scenarios with the same vehicles in opposite locations. Clear the slots before loading the second set of vehicles.

debugmsg [message name] [text]

Brings a debug message up on the screen. If no message with the specified [message name] exists, a new message is created with that name. If a message with the specified [message name] exists, its current text will be replaced with the specified [text]. If a message with the specified [message name] already exists and the [text] argument is omitted, the specified message is deleted.

drawboundingboxes [1/0]

Shows the collision boxes while running a scenario. This slows down the performance greatly and was meant to be used by the developers.

drawdebughud [1/0]

This command is outdated and not used anymore.

exec [script name]

Executes the given script. This is a subroutine call; when the exec terminates, the previous script is continued.

freezecam [1/0]

This was used in previous builds of MindRover and is now obsolete.

help [command]

Brings up a list of commands and if one is specified help will give you a detailed description of that command.

loadscenario [scenario name]

Makes the named scenario the current scenario. Scenario name must be the full scenario filename, including the path to the scenario. Looks like this:

loadscenario scenarios/s_goldrush.vmf

Note: Linux systems may require proper filename capitalization for loadscenario and loadvehicle to work.

loadvehicle [team] [slot] [rover name]

Loads the named vehicle into the team and slot for the current scenario (teams are 1 and 2, slots are 1 and 2 per team). This is used within a script that automatically starts a scenario. Vehicle name is the full filename:

Vehicles/username/vehiclename.vmf

login [username]

Log in as the given user (must be before the first LoadScenario). This is used in a script that can be run from a DOS window when starting up mindrover (before you log in):

```
c:>mindrover /exec [script name]
```

For more information on scripting, there is an example immediately after this list of commands.

playtrack [number]

Plays a specific track on the CD. You can swap any CD in after MindRover has started and this command allows you to select which songs you want to listen to.

quit

Exits back to the operating system.

rem

This command is ignored. Used for placing remarks in a script file.

returnonendscenario [1/0]

If true (default is false), when the scenario ends the scenario automatically returns to the console.

screenshot

Takes a picture of the current screen and saves it to MindRover's install directory.

setconsolelog [filename]

All messages sent to the console get recorded in [MindRover install directory]\[filename].log. If [filename] is empty, console messages are not recorded (the default).

setgamma [value from 0-2]

Changes the gamma level. Values range from 0 to 2.

setscenariotimeout [Numbers seconds]

Will force a scenario to end after the specified number of seconds.

setupscreenshots [N] [filenamebase]

This is a setup for "togglescreenshots". When that command is run, a screenshot will be taken every N frames and saved to

MindRover's install directory with the filename "filename-baseXX.tga"

startscenario [seed]

Equivalent to hitting the GO button with the specific seed. The seed is an optional integer. Useful if you want to run all of the matches in a tournament with the same starting configuration.

togglealphadrawing

Toggles the drawing of alpha objects during simulation (for debugging only).

toggleboundingboxes

Toggles bounding boxes.

togglepause

Pauses or unpauses the scenario.

togglescreenshots

Used in conjunction with "setupscreenshots" to take many screenshots in a row. When activated, screenshots will be taken on a specific interval, set beforehand, until "togglescreenshots" is turned off.

toggletutorial

Toggles the tutorial help.

unbind [keyname]

Undoes a previous binding for a key.

Special Commands:

authorizeall [1/0]

Allows all chassis and weapons to be used.

nolimits

Overrides MindRover's point and weight limits, allowing you to place as many components on a chassis as you can fit.

Note: Although some of these commands many not seem useful, most are essential for scripting. Here's an example of a script you can run, using the either the /exec command line parameter or the "exec" console command:

```
Login Tournament1
ReturnOnEndScenario 1
SetConsoleLog Tournament1.log
SetScenarioTimeOut 300
LoadScenario scenarios/s_hotpotato.vmf
LoadVehicle 1 1 vehicles/Tournament1/contestant1.vmf
LoadVehicle 2 1 vehicles/Tournament1/contestant2.vmf
StartScenario
Clearslots
LoadVehicle 1 1 vehicles/Tournament1/contestant3.vmf
LoadVehicle 2 1 vehicles/Tournament1/contestant4.vmf
StartScenario
Quit
```

The command file MUST reside in a subdirectory of MindRover (you may want to create a subdirectory for this purpose), and you must specify its path relative to the MindRover root:

```
mindrover /exec commands/test.cfg
```

Or pop up the command console within MindRover(found at the bottom of the command screen) and type:

```
exec commands/test.cfg
```

Loading, saving, and copying vehicles

To load a vehicle, drag it from the holobox at the top of the screen into an empty slot. If there is a rover in the slot you want to use, you can drag that rover out of the slot to empty it. Since the default row in the holobox is a list of empty chassis, you will need to scroll down to find your vehicle.

Vehicles are automatically saved any time you run a scenario, go to the vehicle selection screen, or log out, so you should never have to worry about making sure your vehicle is saved. However, since there are times that you may not want your changes permanently saved (when you just want to test a change, for example) you may wish to copy the vehicle first so you still have an unchanged version.

To copy a vehicle, drag the rover from the holobox into two empty slots. The rover in the second slot will automatically be renamed and is now a new rover.

Importing and exporting vehicles (sharing with your friends)

Vehicle files are saved as files in the [MindRover install directory]\vehicles\[loginName] directory. Each vehicle has three files associated with it: A [vehiclename].wst, [vehiclename].ice, and a [vehiclename].vmf file. For Linux players, the vehicle directory is: ~/.loki/mindrover/Vehicles/<loginName>

Using a vehicle in a scenario requires only the vmf file. Thus, you can add other players' vehicles to your system (import them) just by saving their vmf files to your vehicle directory. Likewise, you can share your own vehicles by giving the vmf files to other players.

Editing a vehicle, however, also requires a wst and ice file. If you would also like other players to be able to see the wiring used to create a vehicle, you will need to send them the wst and ice files, in addition to the vmf.

Note that MindRover only scans the vehicle directory when you first start the program, so you should exit and restart the program any time you import a vehicle.

Reloading ICE rovers

If you prefer to create rovers using the ICE programming language instead of the wiring screen, you will now be able to save a lot of time while compiling your rovers. Instead of quitting MindRover and editing your rover, you can just ALT-TAB out to a text editor and change your rover. Next, ALT-TAB back into MindRover and type: "reloadvehicle <vehiclename>" in the console. This will recompile your rover and it’s ready to be tested!

Chapter 3

Basic Strategy Tutorials

The following tips provide complete instructions on how to create a basic rover to accomplish the task in a given scenario.

Tutorial 1: Using TrackSensors

This tutorial is intended to teach you how to keep a wheeled rover on the race track. This particular method uses two TrackSensors pointed to the left at the same angle to maintain a fixed distance from the edge of the track. One sensor rides outside of the track and the other rides just on the track. This is a way to learn the basic controls for wheeled rovers and after completing this tutorial you will be able to beat the default opponents.

Select the Race category and select the Hallway Racing scenario.

Now select the vehicle page.

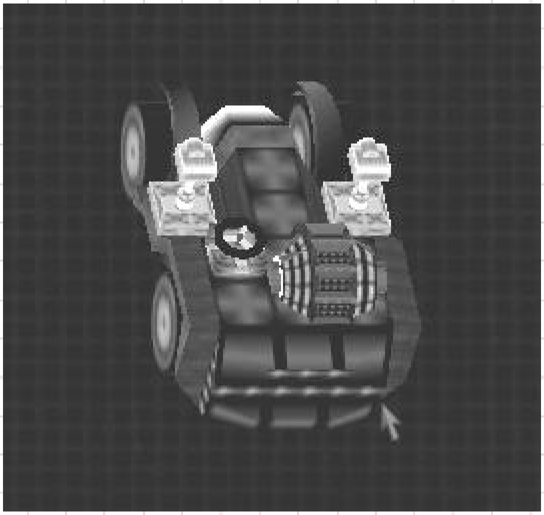
All the chassis are available in this scenario, but for now, use the small wheeled one. This is a race, after all, and the small-wheeled chassis is the fastest of all the vehicles. To select it, drag the small wheel (WheelS – the blue chassis) from the holobox into the team 1 slot. This will enable the component and wiring pages.

At this point, you may wish to rename your rover. The default name will be something like [username]_WheelS[count]. You may also keep the default name for now and rename it later. To rename your rover, click on the “name edit box” on the right hand side of the screen (you may have to select your rover in the Team 1 slot if it has been deselected somehow). This places a cursor in the edit box and you can enter a new name.

Adding components

Next select the component page so the attach points for the small wheeled chassis are shown. The WheelS has 10 attach points; eight are in a large group in the center, which allows for placement of items requiring multiple spaces. The remaining two are single spots (one on each side) that can only be used for components that require only one attach point.

In the holobox, go to the movement category and select the LargeEngine.



Drag the engine onto your chassis (anywhere in the lower section will do).

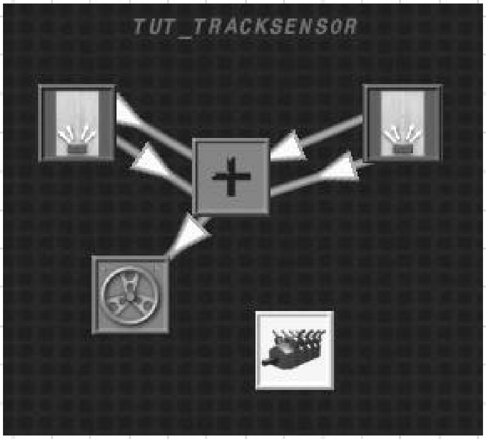
Next, select a steering wheel (also found in the movement category), and drag it on to your chassis next to the engine.

Now that the rover can move and steer, it needs sensors so it knows where to go.

Select the Sensors category and find the TrackSensor component. Drag two of them onto your chassis, placing them in the two single component spots on the front corners.

Wiring it up

Now move onto the wiring page.



Select the engine and set the throttle to 10 (you can increase it later, but keep it slow for now so you can see what is happening).

Point both TrackSensors 45 degrees to the left and leave the range at 1.0 meter. Your rover will be set up to try to keep the edge of the track between the ends of the two sensors.

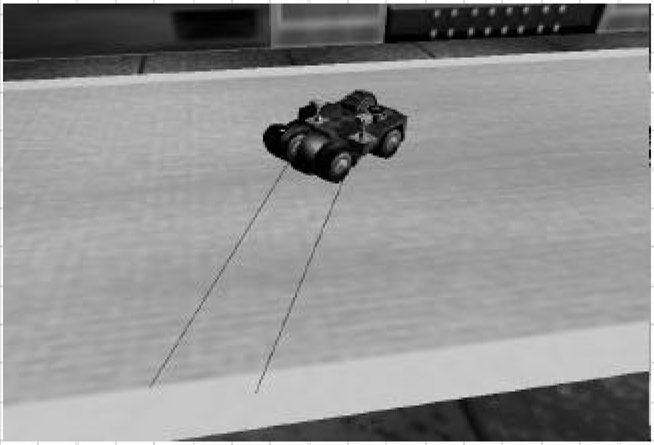
Each TrackSensor has two states: OnTrack and OffTrack. The goal is to get the left sensor to be off and the right sensor on. Since you will need to combine the readings from the two sensors, choose an Add component from the Math category and drag it into the work area.

Drag a wire from the left TrackSensor to the addition component (you can create a wire by dragging one component on top of another). The default wire should trigger on event TurnOff and set Input1 to the Value 0. This is what you want, so no changes are necessary.

Next, drag a second wire from the left TrackSensor to the Add component. This time it is necessary to change the values from the defaults. Change the event to be the TurnOn event, set property to be Input1, and change the Value to be 10. Since the left sensor will turn on when the rover has moved too far to the right, a setting of 10 will steer left and get the rover moving back toward the edge of the track.

Now you need to make two similar wires from the right TrackSensor to the addition component. The first wire will prevent the rover from driving off the track by steering back to the right when the rover has drifted too far left. The wire's defaults are almost correct (TurnOff, Input2); you just need to set the value to –10 (turn to the right).

The second wire triggers once the rover is back on the track and simply resets the Input2 value back to zero (steer straight). The wire needs to trigger on the TurnOn event and the Set Property needs to be changed to Input2. You can leave the value at 0.



The final step is to drag a wire from the addition to the steering component. The defaults will work just fine (event Change, set Angle, To Source Property Output).

You can now test out your new rover. Due to the low engine setting, it will not be particularly fast, but it should still defeat the default opponent, and in the process, you can see how it steers down the track.

You are now ready to increase the rover's speed. As it goes faster and faster, you may notice it start to bounce left and right. Setting the steering angles smaller will help to prevent this. You can also reduce the length and angle of the TrackSensors in order to hug the edge of the track and cut the corners more closely.

Tutorial 2: Using a WaypointSensor

In this tutorial you will learn how to use a WaypointSensor by building a hover rover designed for the sumo mat. This tutorial involves using a SpinThruster, which continually changes its direction to aim toward the waypoint and stay on the map. Once that is working, you will use a combination of logical components to improve the rover's performance.

Select the Sports category and select the Sumo Hover scenario.

Now select the vehicle page.

The three hover chassis are displayed in the holobox at the top of the screen. Drag a medium hover (HoverM - the gray chassis) from the holobox into the team 1 slot. This will enable the component and wiring pages.

Select the component page to display the attach points for the medium hover. The holobox now displays the physical components that are allowed in the Sumo Hover scenario.

Scroll down to the movement components (or use the dropdown box) and find the SpinThruster.

Drag it on to the center of your rover.

Scroll to the Nav/Comm components and find the WaypointSensor. Drag it next to the SpinThruster.

Select the wiring page from the menu on the left.

Click on the SpinThruster and set the thrust to 100 using the slider.

Click on the WaypointSensor and change the waypoint list to be just the single letter 'a'. This will prevent the WaypointSensor from trying to find the waypoint 'b' after it hits waypoint 'a'. Now, when waypoint 'a' is hit, the WaypoinSensor will try to locate the next waypoint but since the rest of the list is blank, the WaypointSensor will start the list over and continue to track waypoint 'a'.

Now drag a wire from the WaypointSensor to the SpinThruster.

Change the "Trigger Event" to be 'Change'.

Change the "Set Property" to be 'Angle'.

Change the "To" to use a source property (which should be 'Bearing' by default).

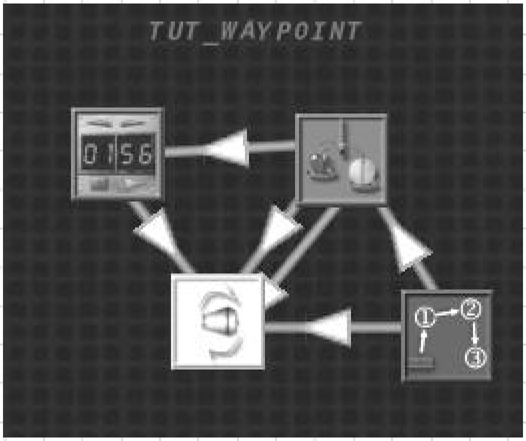
Hit Go and take a minute to try out your rover. Against the default opponent, you might even win! However, as it stands now, your rover will keep overshooting the waypoint and end up flinging itself out of the ring within a few cycles. This problem, however, is easily fixed.

In the holobox, select the Logic category and find the Compare component. Drag it onto the wiring area. This Compare will be used to compare the rover's distance to waypoint 'a' and a set distance at which you want to turn on the thrusters to stay on the mat. If the distance to waypoint 'a' is greater than the set amount, the thrusters will turn on. If the distance is less, they will turn off.

Drag a wire from the WaypointSensor to the Compare.

Again, you will need to change the trigger event to 'Change' and select it to use a source property.

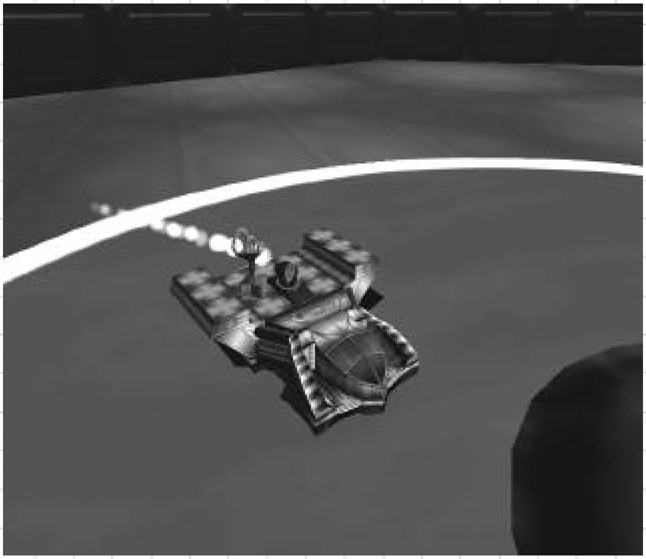
For the source property, select Distance.



Click on the Compare and set Input2 to the threshold distance. A value of 0.7 should work. Drag a wire from the Compare to the Thruster and change the trigger event to LessThan.

Set the "To" value to be -100 to reverse thrust (this will slow down the rover).

To prevent the rover from completely reversing direction, add a Timer (found in the timing category) to turn the thrusters off before that happens.



Drag a wire from the Compare to the Timer to set the tick time to 0.1 on the LessThan event.

Drag a wire from the Timer to the Thruster to set the thrust to 0 on the Tick event so that after a short reverse burst, the thrusters will shut down.

Finally, drag a wire from the Compare to the thrusters to set the thrust back to 100 on the GreaterThan event so the rover doesn't drift off the mat.

Your rover should now stay on the mat. Of course, simply staying on the mat will not be enough to defeat your opponent. You will also need to be able to push him off the mat, so the next step is to add sensors and extra logic to try forcing the opponent off. You may also wish to add more thrusters.

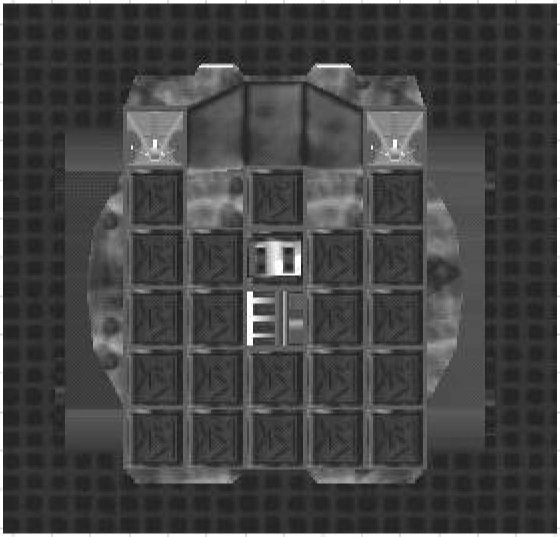
Tutorial 3: Using Multiple Radars

This tutorial is intended to introduce you to using multiple filtered radars to quickly locate and track an opponent. Here we use two radars that are both pointed forward and slightly overlap. By filtering and comparing their outputs, you will create a simple yet effective tracking system.

Select the Battle category and select the Word War I scenario.

All the chassis are available in this scenario, but for now choose the large treaded one (TreadL) for its high health and fast turning ability.

Now select the Components page and add a SmallEngine and a TreadControl so your rover will be able to move (both are located in the Movement section of the holobox).



Next, add two medium radars (these can be found in the Sensors group). Place the radars on each of the front corners of the rover.

Now switch to the Wiring page.

The first thing you will need to do is add the filters. In the Logic section, find the Filter component and drag two filters onto the work area, placing each near radar.

To hook up the filters, simply drag a wire from each radar to the filter. You do not need to adjust any of the default wire settings. Should you accidentally change one , just delete the wire you changed and create a new one.

When using filters, keep in mind that although a component can be wired to multiple filters, a filter can only have one component wired to it. Therefore, your radar could have five different filters. However, you can NOT have two radars use the same filter. This is because the filter actually hooks back to the original component and the hook will be overwritten if a second component is wired to it.

Now, select each filter and set the EnemyVehicle switch to 1 (on). This will allow your rover to detect enemy rovers without being sidetracked by projectiles (or friendly rovers in scenarios where you get multiple rovers). Make sure all the other switches are off to prevent the filter from triggering when it sees those types of objects.

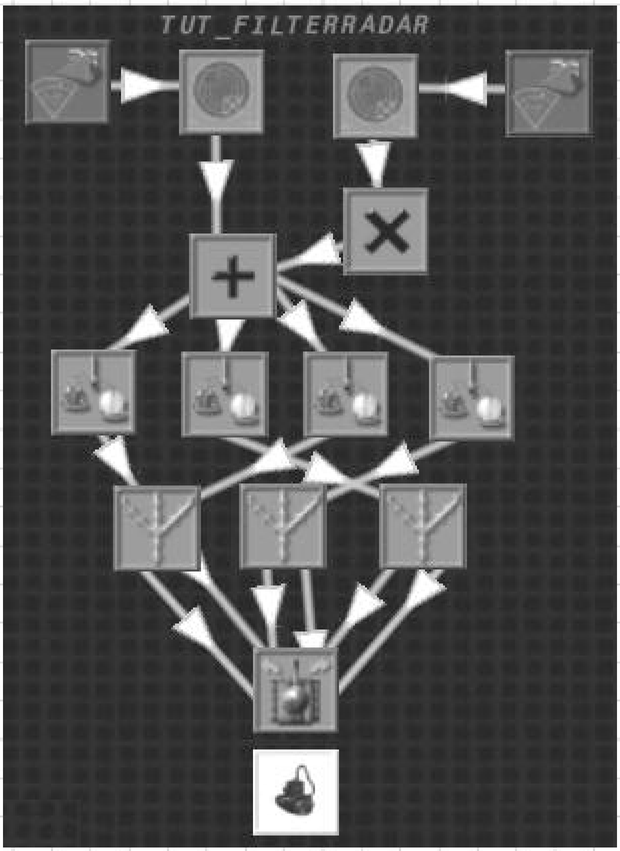
The next step is to wire the steering. Since tread controls always require two wires to set the rover's motion (one wire for each tread), the screen can become cluttered very quickly. To keep the wires organized, add three Broadcast components. Each broadcast will set both inputs to the tread control. By triggering the broadcasts, you will change the rover’s motion to turn left, turn right, or stop turning.

From the broadcast for turn left, drag two wires, one that sets the LEFT tread to -100, and another that sets the RIGHT tread to +100.

From the broadcast for turn right, drag two wires, one that sets the LEFT tread to +100, and another that sets the RIGHT tread to -100.

From the broadcast for stop turning, drag two wires, one that sets the LEFT tread to 0, and another that sets the RIGHT tread to 0.

Now you are ready to add the logic for turning based on what your radars sense.



First, aim the radars so they overlap. You can do this by selecting the left radar and aiming it at -45 (to the right), and then selecting the right radar and aiming it at +45 (to the left). You should also increase the scan width to the full 90 degrees available for a medium radar.

Now, use the combined output from both radars to control the vehicle's movement. This leaves you with four possibilities: both radars are on, just the left is on, just the right is on, or neither radar is on . To convert the combined on/off states to modes, use a multiply and an addition operator. Wire the output of the left filter to Input1 of the Add. Set the trigger event to Change and set the To Source Property to 'State'. This will cause Input1 of the Add to equal 1 when the left radar (which looks to the right) detects an enemy and 0 when it does not. Then wire the output of the right filter to Input1 of the Multiply, setting the event to Change and the To Source Property to 'State'. Set the Input2 of the Multiply to 2 and wire the output of the Multiply to the Input2 of the Add. Now, Input2 of the Add will equal 2 when the right radar is on and 0 when it is not.

At this point, the Add will have an output of 0, 1, 2, or 3. It will be 0 if both radars are off, 1 if only the left radar detects an enemy, 2 if only the right radar detects an enemy, and 3 if both radars detect the enemy.

To set the tread control, you need to test the output value of the addition component. To test it drag four Compare components from the holobox into the work area (one Compare for each possible state). Set the Input2 properties of the Compares to the values you want to test for: 0.0, 1.0, 2.0, and 3.0. Then drag a wire from the addition component to each Compare (the default settings are fine). This will cause the Compares to trigger their Equal event when the Add outputs the value you set for Input2.

Now drag a wire from the =0 Compare to the turn left Broadcast. This will cause your rover to start spinning around to search for the enemy, should it ever lose track.

Next, drag a wire from the =1 Compare to the turn right Broadcast. In the event that only your left radar (looking to the right) sees the enemy, your rover will now turn right to get it in its sights. Do the same for the =2 Compare to the turn left Broadcast.

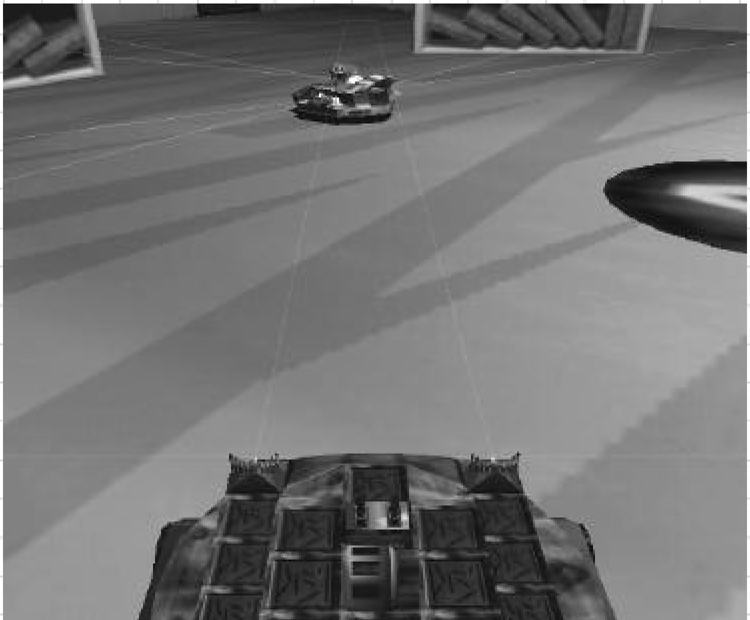
Finally, drag a wire from the =3 compare to the stop turning broadcast, since if the enemy is in both radars, it must be directly ahead of you and you do not need to turn any longer.

The final steps are to set the engine to full throttle and give the rover an initial spin. Since a treaded rover's movement is really controlled by the TreadControl, you can set the engine to 100 and not have to worry about it again. For the initial spin, set the tread control to have one tread at +100 and the other at -100 (it doesn't really matter which tread is set to which value).

You can now try out your rover. You should see that it very quickly locks onto the enemy and tracks its movement.

Of course, you haven't added any weapons yet, but that can easily be done. And you already have a component to trigger them. Whenever the =3 compare triggers, you know the enemy is directly in front of you, so fire away…

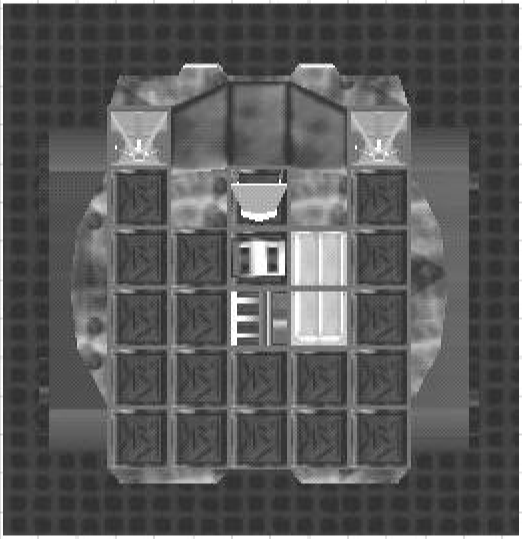
Note: This task can easily be accomplished using a Demux instead of the four Compares to control each input and response.



Tutorial 4: Using Sonar to See Walls

Radars are an excellent way to quickly find and track an opponent. They have a large scan area and can even see through obstacles like walls and rocks, but if your opponent is hiding behind a wall, your weapons still won't be able to get to them. You can add a sonar to the rover you built in the filtered radar tutorial to detect walls and other obstacles and allow your rover to move around them.

Open the Word War I scenario and load the rover from the filtered radar tutorial. To load the rover, select the vehicle page and select the Battle category from the drop down menu in the upper left. This will display all the Battle rovers you've created so far - one of which is the one you made for the previous tutorial. Drag it into a vehicle slot.



On the component page, add a Sonar component to the front middle of your rover (the sonar is located in the sensor category).

Switch to the wiring page and, if necessary, move the sonar's icon to an open spot.

Before you start wiring the sonar, add some forward movement to the rover by creating a 4th broadcast that sets both treads to 100% forward, and have the compare =3 component trigger it. Consequently, your rover will close in on the target when it is straight ahead. Try out the forward movement change and see how your rover now chases the opponent (albeit slowly).

Now you are ready to add the sonar wall avoidance. To set up the sonar, you need to give it an input trigger since, unlike most other sensors, the sonar does not update on its own. Instead you need to trigger the sonar's Fire event. An easy way to do this is to add a Loop Timer (located in the Timing category), set the TickTime to 1.0, and drag a wire that triggers on the loop timer's tick event and activates the sonar's Fire event. This will cause the sonar to generate either a ping or a no-ping event, depending on whether or not it detected an object.

Next add a logical filter to the sonar. Set the Other type to 1 and leave the remaining types off (0), since at this point, you are only trying to avoid walls. (The rover is too slow to dodge rockets and the current goal is to attack the enemy, not dodge it. You could also avoid teammates, but you don't have any in Word War I).

Drag a wire from the sonar to the filter to get it 'plugged in'. Do not change any of the default settings. Now the filter will generate TurnOn/TurnOff events, much like a radar does. The disadvantage to this is that you will no longer have access to information regarding an object's distance once the filter turns on. Thus, if the distance of an object changes between two pings, there won't be any indication that it has done so (the filter will simply remain on and not generate any additional events). However, for our purposes, this is not a problem.

Select the sonar and reduce the range to 1.0 meter. This will prevent your rover from attempting to dodge walls that are not even close to you. You may find it necessary to come back and adjust this value later to tweak the performance of your rover.

When the sonar's filter turns on, the desired effect is to have the rover turn right until the sonar turns back off, then turn right just a bit more, then move forward, and finally turn left and switch back to search/attack mode.

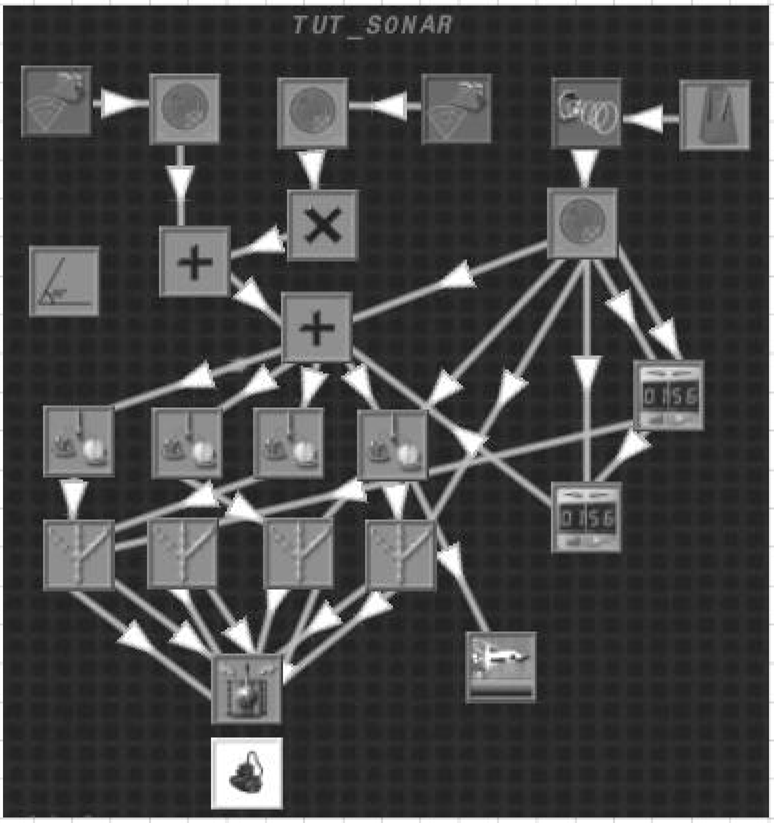
The first thing to do is disable the normal search/attack mode. Since your rover bases its movement on a single value in the range 0-3, one easy way to disable that movement is to push the value outside the 0-3 range. To do this, insert an addition component between the original addition component and the compares. Pass the output from the original addition to input1, and set input2 to the value '0'. Later, to disable the search/attack mode, you can set input2 to a value such as 10, moving the range from 0-3 to 10-13. Since there aren't any Compares that trigger on the values 10-13, any changes in the radar state will be ignored. Setting input2 back to 0 will re-enable the search/attack mode.

To create the dodge pattern, create two wires from the sonar's filter that trigger on the TurnOn event. One wire should set input2 of the new addition component to 10 (to disable the search/attack mode), while the other wire triggers the turn right broadcast so the rover will turn to the right.

When the sonar's filter turns back off, you'll want your rover to stop turning. You will actually want to turn a bit farther so the rover doesn't run into the wall again, but since you are only pinging the sonar once a second, the rover will already have overturned a bit. You'll need add a timer before creating the wire, the Timer is part of the Timing category of components. Create two more wires, this time that trigger on the TurnOff event, one that triggers the move forward broadcast, and another that sets the timer (note, this is different than the loop timer and is frequently referred to as a 'one shot timer' to differentiate it from a loop timer). The timer should be set for the maximum amount of time you'd like to be moving forward (it might get interrupted by the sonar detecting another wall). For now, set it to 2.0 seconds.

When the timer triggers its Tick event, you'll want to turn left for a bit to try to reacquire the original target, You can program your rover to do this by dragging a wire from the timer to the turn left broadcast.

To keep your vehicle from turning too far left, add another timer and wire the first timer to set it to 1.0 second. When the second timer triggers, have it re-enable the search/attack mode by setting the input2 of the addition component back to 0.



Your rover is nearly ready, but requires one final bit of cleanup. If you are in the move/turn left part of the dodge and you detect another wall, you need to shut down the timers. You can do this by adding two more wires from the sonar's filter that trigger on TurnOn and set the tick time of each timer to 0. If the timers were already at 0, setting them will have no effect. If either timer was ticking down, setting it to 0 will stop it **without** causing it to trigger.

Testing the new setup will be a bit trickier, since to test it properly, you'll need an opponent that ends up behind one of the bookcases. The default hover opponent will occasionally end up outside the bookcases with some help from a rocket explosion, so add a single rocket launcher to your rover (a good location is just to the right and behind the sonar). Then run the scenario and hope for the best. Seed 501111 seems to work well, so you could try it (assuming your rover is set up in the previously specified manner) by clicking on the console bar at the bottom edge of the screen, typing 'startscenario 501111' and pressing the Enter key. You can also just press the Go button and within a few trials, you should see your rover dodge around a wall.

TIP: If you get a good test case after starting a test using the Go button and would like to use it again, you should immediately exit out of MindRover and examine the contents of a file called "mindrov.log" in the MindRover installation directory. Near the end of the file you should find a line like 'Random seed is: 501111.' You can repeat the test by starting up MindRover again, loading the same rovers, and entering 'startscenario NNNNN' (where NNNNN is the number you got from the log file.) With the same scenario and rovers, it will produce identical results each time you run it. Of course, any changes to either of the rovers will affect the results at the point that the changes have some effect, but the starting positions will always remain constant

Chapter 4

Advanced Strategy Tips

The following tips are ideas that you can add to your growing arsenal of techniques:

Tracking a target with a single radar

In the course of playing this game, you have probably come across several situations that have left you wishing you could get rid of one of your radars so you can use the space for other components. You will no doubt be happy to know that it is indeed possible to make a good rover, capable of finding and tracking an opponent, using only one radar. In fact, finding an opponent with a single radar is not necessarily any harder than when using two, and may actually be slightly easier since you don't have to worry about keeping it synchronized with anything.

The obvious problem with using only one radar is that when the radar turns off you do not know if the target moved to the left or right. However, if you were to move your radar with the intention of making the target leave the radar scanning area, you would know which side the target is on. Of course, if the target moves in the direction that you shifted the radar, then the radar will need to be shifted in the same direction again (repeat until the radar turns off). Once the radar turns off, reverse the direction until the radar turns on. The net effect is that the radar will continuously bounce left and right with the target being just on the edge of your radar's scan area.

This allows you to point your weapons (or orient your rover) toward that radar edge and fire away. A side benefit to using this method (as opposed to using a fixed single radar) is it doesn't matter how much area your radar is scanning because the edge of the radar is always just a line (and that is where the target is).

The biggest problem you will have is that a fast moving opponent will require large changes in radar position to keep up, while you'll want small changes to precisely track a slow moving target. A good solution is to use some form of accumulator to get an average of the required change to track the opponent and dynamically update the sweep increment throughout the combat.

Tracking a target with Sonar

A sonar makes an excellent tracking system since it gives the target's distance, as well as a very precise angle. The downside is that it does not scan as much area per ping as a radar (keep in mind that the target has some width and the sonar can detect any part of it, so the sonar's scan width is the apparent width of the target, which unfortunately will vary with the distance the target is to you). Using two sonars can be more effective because you can update the ping location based on the sonar with the smallest distance (looking for the target's near corner). Two sonars will, of course, also cover a larger scan area.

To maximize the scan rate you can dynamically change the angle increment based on the last ping distance (be it wall, obstacle, or target). That way you can quickly scan past walls that are near you, while carefully examining the large expanse.

Of course, using a radar along with a sonar is always a good way to quickly locate and then track the target.

Replacing the WaypointSensor with a XYFinder

The WaypointSensor and the XYFinder components really do the same thing, but in different ways. The WaypointSensor reads the target XY coordinates from the scenario, while the XYFinder requires you to set them. The WaypointSensor can automatically increment to the next waypoint, and while the XYFinder can not, a simple compare using the output distance from the XYFinder will have the same effect.

Of course, the BIG advantage of the XYFinder is that you can place your XY waypoints anywhere you want, so your rover is no longer limited to the built-in scenario waypoints.

The hardest parts of using the XYFinder are getting the coordinates to input, and then storing the coordinates in your wiring so you can update the XYFinder during the competition.

To get the coordinates, it is usually best to start by creating a simple rover that uses a non-XY method of navigation, such as a WaypointSensor, TrackSensors, or a sonar. You can even create a hover that simply thrusts and bounces around randomly. Then place a XYSensor on the rover and hook it up to two DebugMessage components. That way as the rover moves around, you will get a display of the XY position of the center of the rover. Draw a map and mark down the various coordinates you're interested in. We have included maps for a variety of levels in this guide to make this process easier.

Once you have the coordinates, you have to store them. It is generally best to use a variable that stores the current 'XY waypoint' index, and then use a set of compares to test which index is currently set (this is similar to what is done in the filtered radar tutorial). Each Compare will test for a specific index, and when found will set two variables (one for X and the other for Y). For ease of setup, set the X and Y values to 10 times the intended value (so 2.7 would be 27), then use a divide component to divide the variable's output by 10 and use the divide's output to set the XYFinder. The result will contain a lot of wires, but by giving plenty of space between the compares and the X and Y variables, the correct wires can be located without much trouble.

Since the XY waypoint index is contained in a variable, you can use the variable's IncrementBy feature and have the 'range test' compare (attached to the XYFinder) increment the variable by 1 when your rover gets close to the waypoint. It is also useful to setup an extra index compare that, when triggered, sets the variable back to 1. One way to set the initial XY coordinates is to have a Startup component to set the variable to 1.

Preventing hovers from orbiting a waypoint

You can stop a hover from orbiting by using an XYSensor to calculate your current direction of travel and then modify your target angle with an opposing force.

An easier way is to use a brake and just pulse it on for a moment, once every few seconds. The brake will stop any movement that the thrusters aren't providing, so orbiting, or generalized spinning will be kept to a minimum amount. The intended direction of travel will also be slowed, but not by much (if you keep the brake's on time low).

Replacing the XYFinder with a XYSensor and compass

At some point, you may find that it would be useful to know your current location and heading (or spin rate) in addition to traversing a set of waypoints. If space/weight isn't a problem, simply adding an XYSensor and a compass may be the easiest solution. However, since weight, and even more often space, are frequently limited, you can save a space by using the XYSensor and compass to do the same job as the XYFinder.

The basic idea is that if you know where you are, and what direction you are facing, then you will know the direction and distance to any other point.

To find the distance, you can use a bit of trigonometry ($a^2 + b^2 = c^2$). Subtract your current X position from the target X position, and do the same for the Y positions. Multiply each result by itself, and add those results together. Then use the SquareRoot component to get the actual distance to your target.

Note: Square roots take quite a bit longer to compute over other mathematical operations, and oftentimes they are not necessary. If you are trying to compare a distance (you want to know when you are 2 units from a wall), you can take the output before you take the square root and compare that to the required distance, squared (so you compare $a^2 + b^2$ to 2^2).

To find the angle, you can start by passing the difference in X locations and Y locations to the AngleFinder component. That will give you the angle from the positive X-axis to the target location. You can then add the Bearing output from the compass (which is the angle from your heading to the positive X-axis) to give the total angle between your rover and the target. One potential problem is that each angle is in the range -180 to +180, giving a combined range of –360 to +360. A range test, a multiply, and another add component can easily fix this, or you can get one of the user created components (like WrapVar) that were made to specifically address this issue. You can find user created components by following the fan-site links listed on the MindRover web page at www.mindrover.com.

Using fixed thrusters instead of a rotating thruster

A rotating thruster is very convenient for heading in the correct direction, but for the same space usage, you can get two fixed thrusters (though these do weigh a bit more). By aligning one of the thrusters perpendicular to the other, you'll still have at least 100% thrust in each direction, and while traveling at an angle, the combined thrust can reach up to 141% of a rotating thruster's output. Additionally, if you are replacing two rotating thrusters, you gain the ability to control your rover's rotation without sacrificing any forward thrust by using the set of thrusters that aren't required for forward thrust for rotation.

To replace the rotational thruster with two fixed, simply pass the angle you would have sent to the thruster to a Trigonometry component (the sine/cosine wave one). Then multiply the sin and cos output by 100 and pass the results to the thrusters (cos goes to your forward facing thruster, sin goes to the left facing thruster). This will give the correct direction. To get the maximum thrust possible, you need to vary the scale so that the thruster with the larger input is exactly 100 (at 45 degrees, both thrusters will be 100 and the combined output will be 141%). A Compare and some math will serve you well for doing the scaling.

Preventing wheeled rovers from oversteering

To understand the solution to oversteering, you need to understand the problem. Oversteering occurs when you tell the rover to turn, but do not tell it to stop turning at the point where it is aimed at the target. Usually, this happens because your sensors have not updated yet, so your rover does not even know it is pointed at the target. A quick method of getting around this is to simply divide the angle to your target by 5 and pass that to your steering. This makes the assumption that for a rover moving at maximum speed, steering at the target angle/5 will not cause the rover to turn more than ‘2 x target angle’ degrees in the standard sensor interval (1/4 second).

A better performing solution is to only steer for 1/5 of the time. Since the standard sensor interval is 0.25 seconds, steering at the full ‘target angle’ degrees for 0.05 seconds will get a full speed rover pointed in the target direction and it will drive straight towards the target for the remaining 0.20 seconds. You can implement this by setting a timer at the same time you set the steering. The timer should be set to 0.05 seconds and on its tick event it should set the steering to 0. And with a bit more logic, you can deal with the cases where the target is >30 degrees by increasing the timer time (and dividing the target angle by the number of 0.05 time units you’ll be turning).

For even better performance, you can calculate your speed and factor that into your turn. For example, the assumption is that your rover is moving around 5 m/s. If you are only moving 2.5 m/s, you will need to either turn twice as long or twice as hard to get the same amount of turn. So by pre-multiplying the target angle by 5/[actual speed], you’ll get better performance at slow speeds. Keep in mind that if you are accelerating, you speed will be changing. Of course, if you do the majority of your turn in the first tick (0.05s), that should not be a problem.

Mode Switch

The mode switch can make your life easier by allowing you to wire your rover one way for one set of circumstances and a different way for another set of circumstances. The most obvious use for this is in a contest like Hot Potato where you are alternately chasing the opponent and running away from it. It can also be useful when you have stages of progress, such as in a battle where you either are looking for your opponent or have found your opponent.

The mode switch has two disadvantages. One is that you can only have one of them, so coming up with a general solution that involves the mode switch will mean the idea will not be usable on many of your other vehicles. The other problem occurs when you change into a new mode. Since the mode switch works by disabling wires (not components), when you switch to a mode, components are not automatically updated with the any old values that were not sent while the wires were disabled. Instead, you will have to add a way to save and then restore the desired values on mode change.

There is no way to get around the first problem (only one mode switch per rover). You will just have to decide if the limitation is acceptable to you. If it is not, you can create similar effects to the mode switch by using switches, queried variables, and multiplication/ addition trees (see below).

The mode switch's second problem can be dealt with by either carefully adjusting your logic, or by adding additional variables to cache any unused values and then query the variables when you switch to the mode.

The mode switch works by enabling all the wires that are a specific color (or gray) and by disabling all the wires that are not that color. Therefore, if the mode switch is set to 'Blue' mode, then all the Red wires act as if they weren't there. Gray wires are always active. You can change a gray wire into a colored wire by selecting the wire and choosing a color for it from the lower right-hand side of the screen (under the label 'Wire color:'). If you prefer, you can also select one of the colors on the left-hand edge of the work area. When a color is selected, only wires of that color and the gray wires will be visible. Additionally, any new wires you create will be created as the selected color (though you can change them later, of course).

Setting its mode color controls the mode switch. The default gray mode leaves only the general-purpose gray wires active. All colored wires will be inactive while in gray mode.

Setting the mode switch to one of the colors will cause any wires of a matching color to become active (while at the same time, the gray wires remain active). Remember, though, that the colored wires that are now active will only affect future events. For example, if you are setting your steering based on a WaypointSensor that passes its value using a blue wire, setting the mode to blue won’t update the steering until the WaypointSensor generates a new event (which could be as much as ¼ second later).

To get an immediate update, have the WaypointSensor pass the value to a variable using a gray wire. Then have the variable set the steering using two blue wires, one triggering on the change event and the other on the query event. Set the mode switch to query the variable when the mode changes. If you make the wire blue, then the variable will only be queried when the mode changes to blue, which is fine. You can also make the wire gray, and the query will still only get propagated to the steering when the mode is blue because the wire from the variable to the steering is also blue.

If you would like to avoid some of the limitations of the mode switch, here are some tips on how to create some mode switch replacements:

The switch component is the most direct replacement for the mode switch. Simply put a switch into the middle of each wire you would have set to a color, and turn it on to simulate being in that mode and off otherwise. The disadvantages to using them are that you can’t pass a value through the switch, and if you have more than a few colored wires to replace, it will quickly become a bookkeeping nightmare.

To get around the value problem (with the switch), you can add a variable. Set the variable instead of triggering the switch, and then have the variable's SET event trigger the switch. If the switch is on, have it query the variable. Then the variable will pass its value to the original target component when queried.

Another way to replace the mode switch is to use multiplication/ addition trees. This is only applicable to the output end of your logic, but quite often that is all you really want.. To use them, create a set of logic and, just before sending the final result to a control component (like steering, engine, thruster, or even radar angle), pass it through a multiply. Then create an alternate set of logic and pass its value through a second multiplication component. Add the results of the two multiplication components and pass that value to the control component. By setting only one of the multiplies to multiply by 1 (set the others to multiply by 0), you ensure that is the only value that will be passed to the control.

This method has several benefits that the mode-switch does not. Primarily, it can be extended to any number of modes (just keep adding multiplies and adds to the end of the chain). Second, you will not need to worry about losing values when changing modes, The old values have already been stored in their multiply, and as soon as you turn it on, the value is sent to the control. The downsides are that it is limited to value propagation and, like the toggled broadcasts, can become a wiring nightmare if you have a large number of controls.

Radio Communication

Radio communication is only applicable when you have more than one rover on the same team. Radio messages are not transferred between teams, so you need not worry about other rovers listening to your message and the other team can't interfere with your messages in any way.

Communication is very simple. Select a channel for communication (in most cases, channel 1 will be sufficient). Have one rover with a transmitter and another with a receiver (both rovers will need transmitters and receivers if you prefer two-way communication). On the transmitting rover, wire a component to the transmitter and set a value to send. The value will immediately create events on all teammates (including the sending rover) that have receivers set to the broadcast channel.

If you want to do two-way communication, set one rover to transmit on channel 1, and the other rover to transmit on channel 2 to prevent both rovers from hearing their own messages.

Because the communication is immediate, you can send multiple values in a stream. The difficulty lies in unpacking the data. For example, you can wire a XYSensor to send both the X and Y coordinates, but the other rover needs to be able to determine which coordinate is X and which is Y. One strategy is to use synchronization. For example, if by trial and error you determine the X is always transmitted before the Y, then the first signal received will be an X position and the second will be a Y. You can direct the signal by having the receiver set a variable, which sets another variable, which toggles a toggled broadcast. When the toggle turns on, have it query the first variable. When it turns off, have it query the second variable. The first variable should send its value to the X location when queried and the second variable sends its value to the Y location when queried.

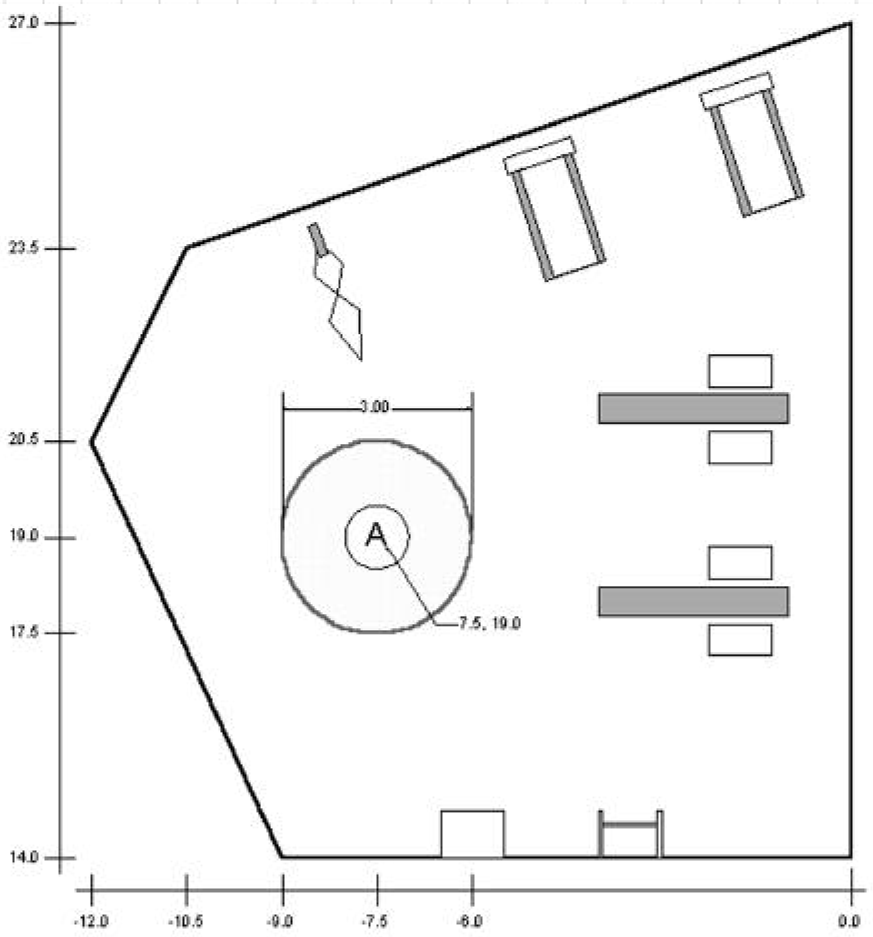
Of course, there are many other ways to unpack the data also. Another possibility would be to use ranges. If you know the X and Y values will always be in the –50 to +50 range, then simply add 101 to the Y value before sending. Thus any received value in the –50 to +50 range will be an X value, and a value in the +51 to +151 will be the Y value +101 (so subtract 101 to get the actual Y value).

Chapter 5

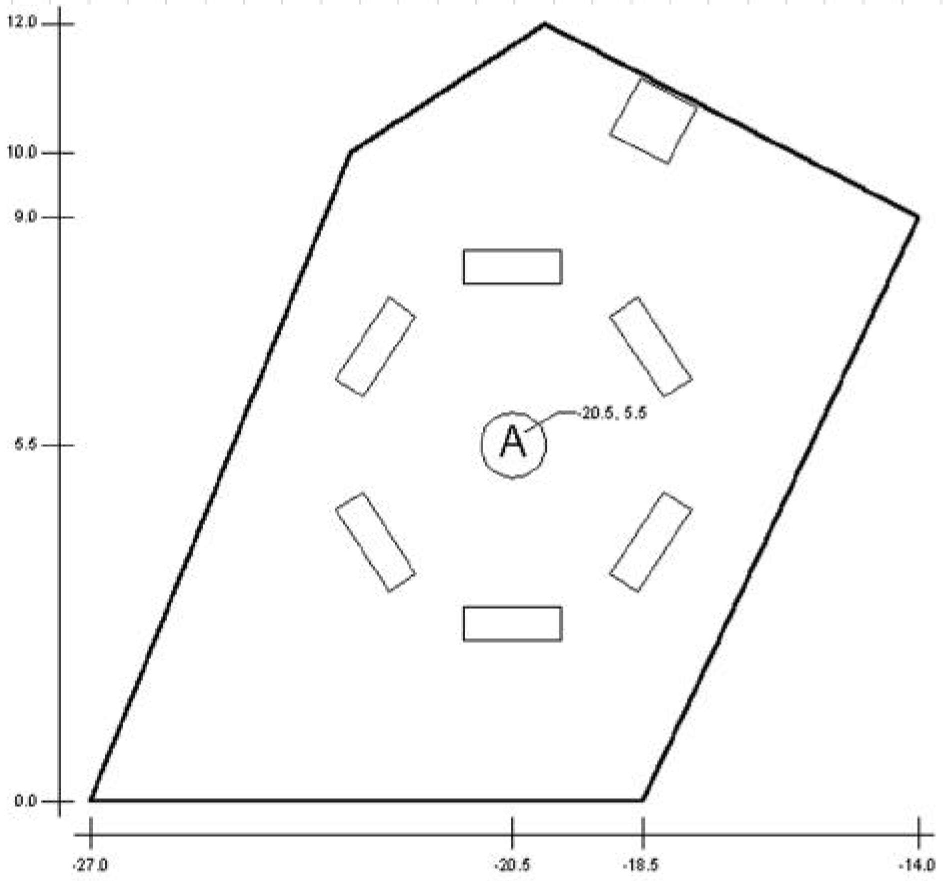
Arena Maps

Images of each of the arenas with XY coordinates and waypoint locations.

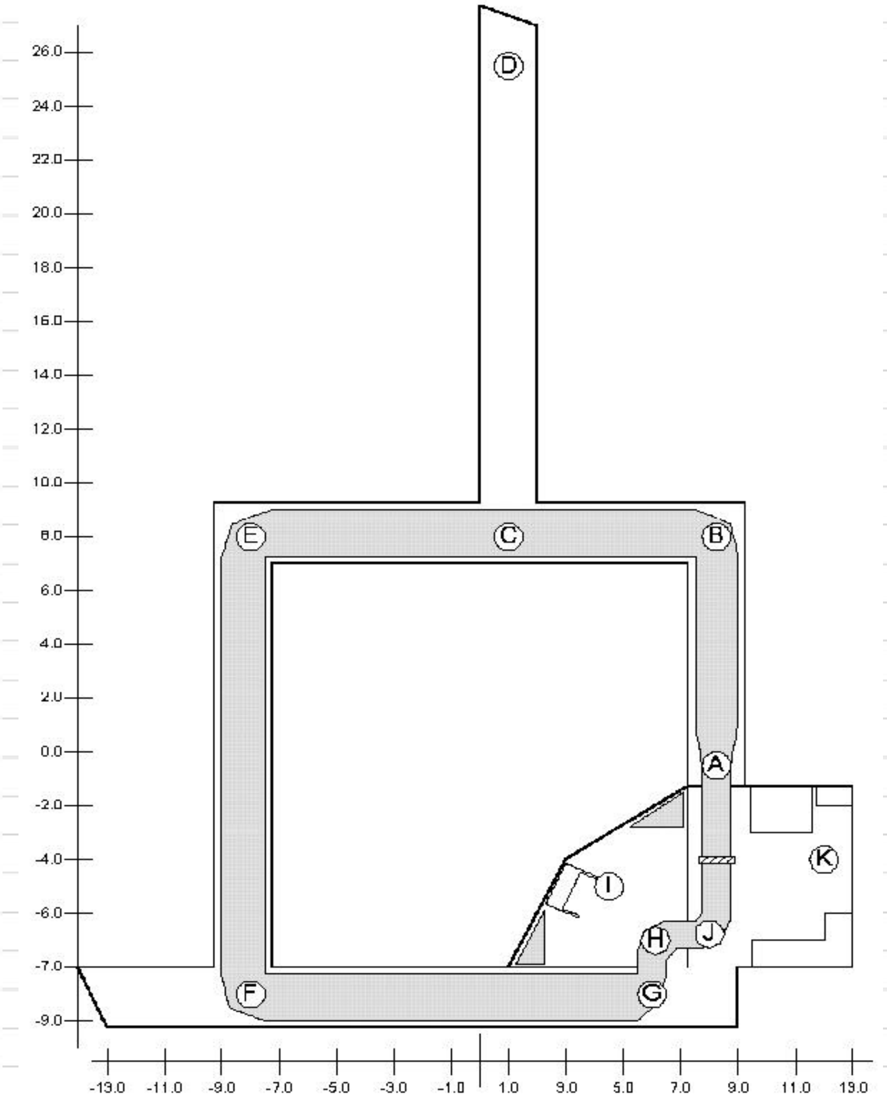
Gym



Library



Hallway



26.0
24.0
22.0
20.0
18.0
16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
-2.0
-4.0
-6.0
-7.0
-9.0
-13.0
-11.0
-9.0
-7.0
-5.0
-3.0
-1.0
1.0
3.0
5.0
7.0
9.0
11.0
13.0
D
E
C
B
A
K
I
H
J
F
G

Bunk Room

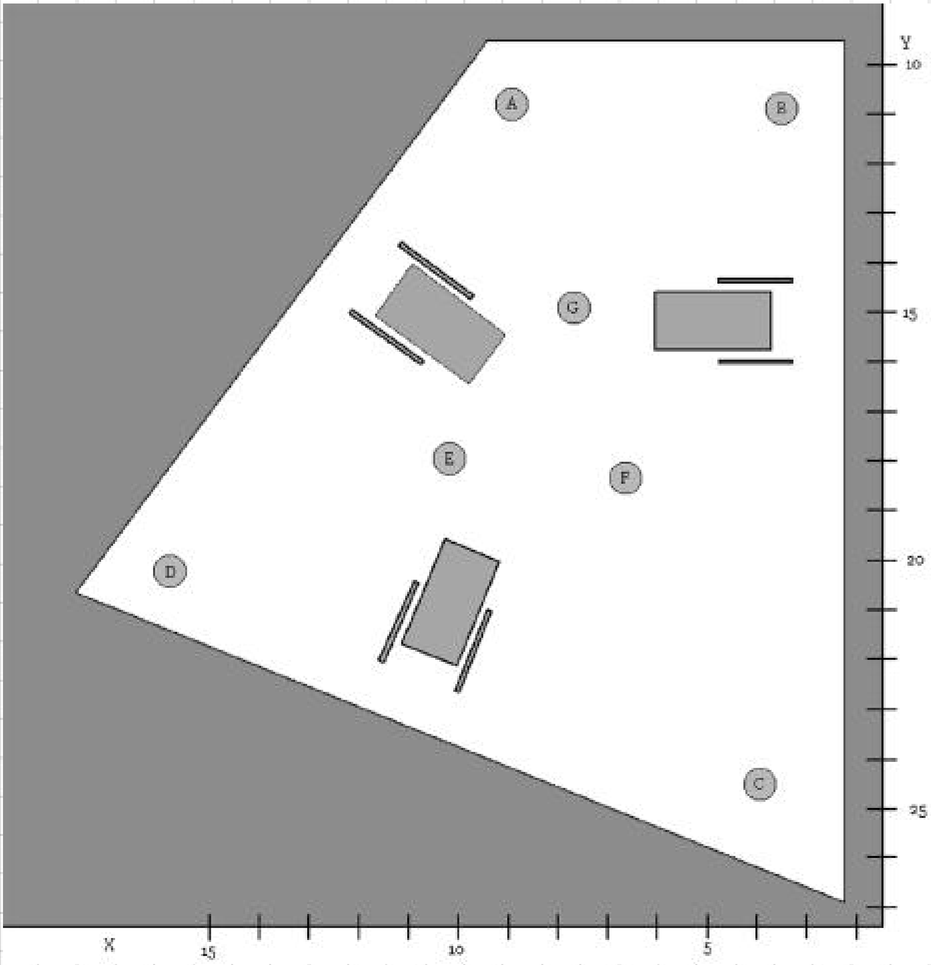
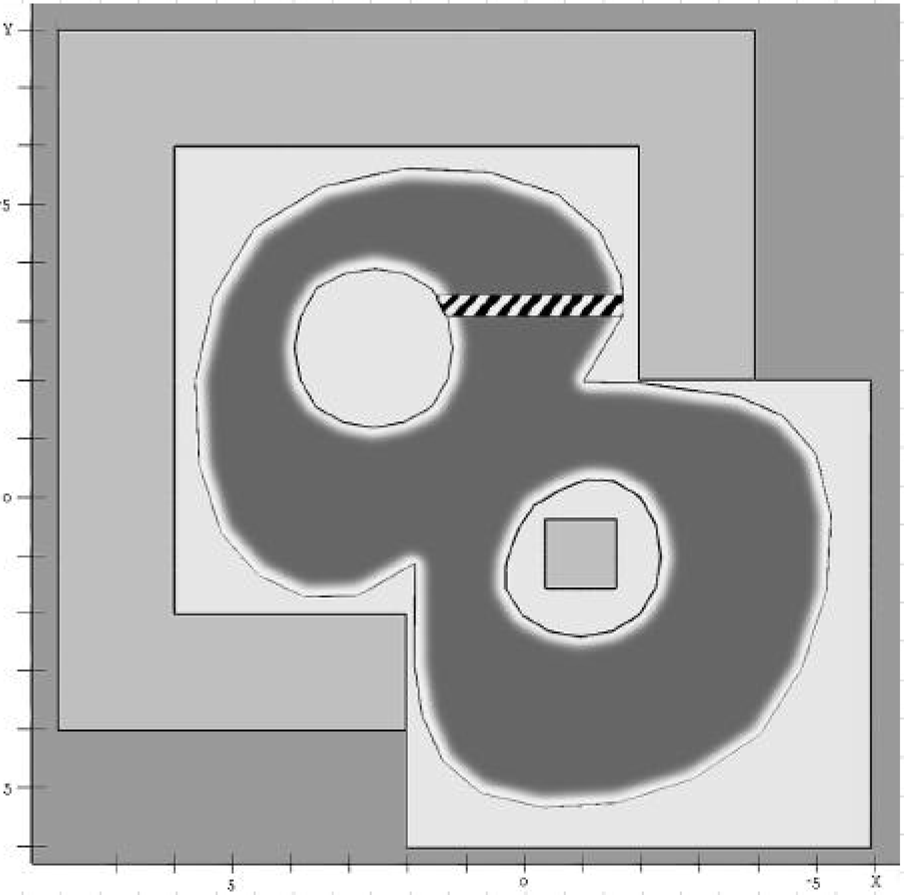
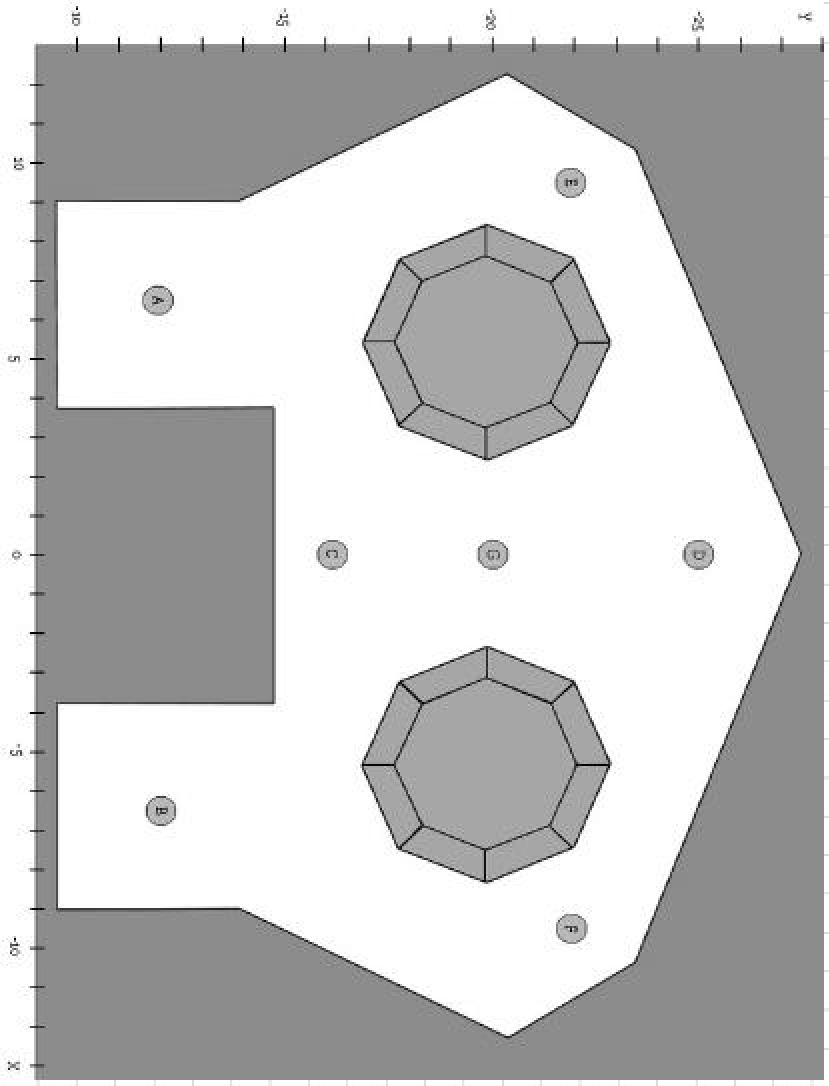


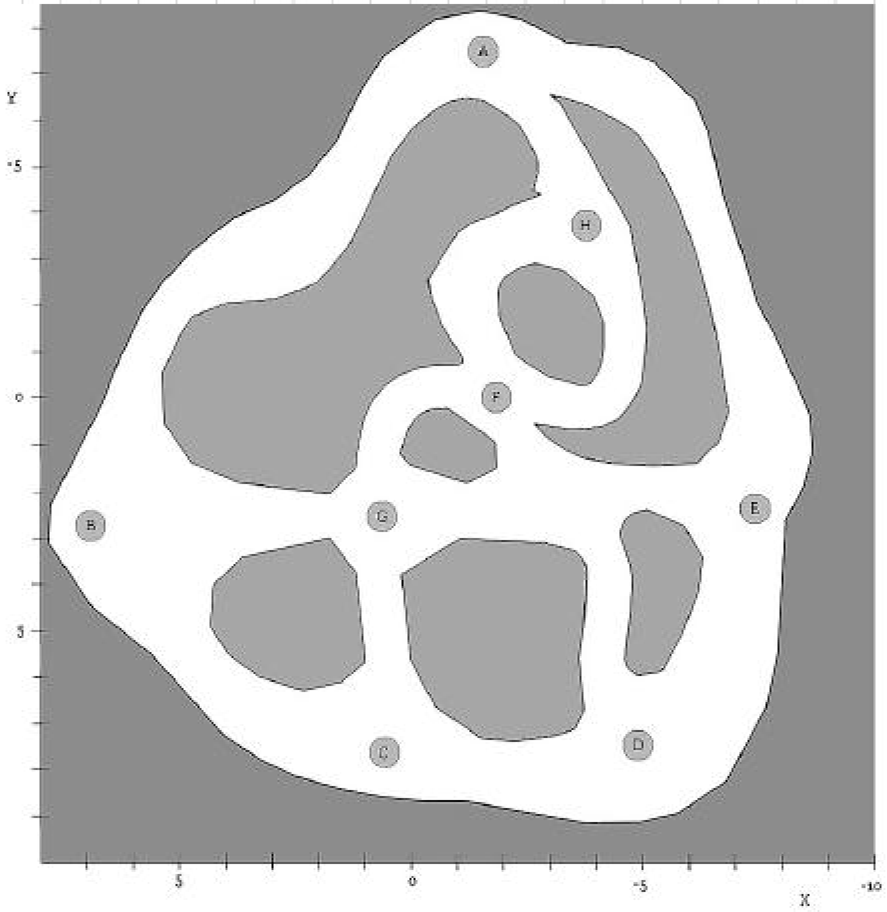
Figure Eight Room



Hydroponics Lab



Quarry



Chapter 6

Component Reference

How To Use

All the officially released components available in MindRover are listed in this chapter in alphabetical order. It should be easy to find a component if you know its name.

In this reference you will find the component description, its icon, its points and weight, and usage notes. Some components will show examples of use with a wiring diagram.

This information is also available within MindRover by selecting the component and pressing the F1 key.

If you don’t have all the components listed here, you can download the add-ons pack from the downloads section on www.MindRover.com.

Add

A component that adds its two inputs and generates an output equal to the sum.

Category:

Logical

Properties:

Input1, Input2, Output

Events:

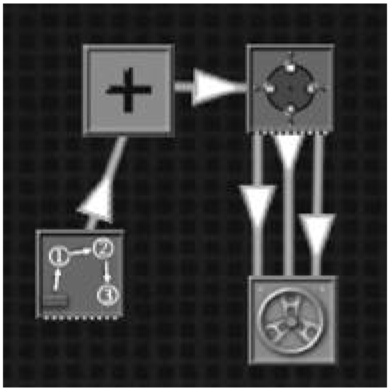
Change, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

Description

This component adds its two inputs, **Input1** and **Input2**, and gives you the **Output** as the sum of the inputs.

The **Change** event is triggered whenever the output changes. The **Set** event is triggered whenever an input is set.



Usage

Instead of steering directly for a waypoint, steer 20 degrees to the left and circle the waypoint by adding a constant to the waypoint bearing.

For this example, you will need a wheeled vehicle, a *Steering* and *MediumEngine* component, a *WayPointSensor*, a *BearingSensor*, and an *Add* component.

Create 3 wires from the *BearingSensor* to the *Steering* component. One wire should read: when the *BearingSensor* triggers **LeftOfRef**, set the *Steering* **Angle** to -15, which is a little to the right.

Similarly, set the second wire to read: when the *BearingSensor* triggers **RightOfRef**, set the *Steering* **Angle** to 15, a little to the left. The third wire should read: when the *BearingSensor* triggers

OnRef, set the *Steering* **Angle** to 0 to go straight ahead.

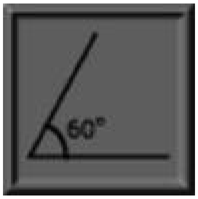
Next create a wire from the *WaypointSensor* to the *Add* which reads: when the *WaypointSensor* triggers **Change**, set the *Add*'s **Input1** to the *WaypointSensor*'s property **Bearing**.

Input2 of the *Add* can be set as a constant value by clicking on the *Add* icon and setting **Input2**'s initial value to 20.

Now create a wire from the *Add* to the *BearingSensor* which reads: when the *Add* triggers **Change** set the *BearingSensor*'s **RefBearing** to the **Output** of the *Add* component.

AngleFinder

A component that calculates the angle of a vector.



Category:

Logical

Properties:

Angle, XPosition, YPosition

Events:

Change, Set

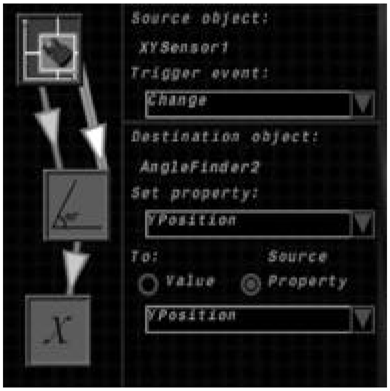
Description

Grp	*Math*
Wgt	*0*
Pts	*0*

The *AngleFinder* component can be used to calculate the angle of a vector from the origin to a point specified by **XPosition**, {YPosition}.

XPosition and **YPosition** are the two input properties. **Angle** is the output.

The *AngleFinder* component will generate a **Set** event whenever **XPosition** or **YPosition** are set. It will trigger a **Change** event when the **Angle** changes.



Source object:

XYSensor1

Trigger event:

Change

Destination object:

AngleFinder2

Set property:

YPosition

To: [ ] Value [✓ Source Property]

YPosition

Usage

Usage: To find the angle of your vehicle from the origin, use the *XYFinder* in conjunction with the *AngleFinder*. Wire the **XPosition** output of the *XYFinder* to the **XPosition** input of the *AngleFinder*. Do the same for **YPosition**.

The **Angle** output of the *AngleFinder* will be between -180 and 180.

AutoHealer

A component that slowly restores health.

Category:

Physical

Properties:

None

Events:

None

Grp	*Defense*
Wgt	*115*
Pts	*70*

Description

The *AutoHealer* component will automatically regenerate health on your vehicle. For the *AutoHealer* to trigger the vehicle must come to a stop. Once stopped, a timer will begin and health points will slowly regenerate.

Regeneration will continue until
a) The vehicle reaches maximum level
b) The vehicle is damaged
c) The vehicle begins moving

When the timer begins, the rate at which the vehicle regenerates is 1 health point per second. That rate will increase by 1 every 2 seconds. This component does not need to be triggered and has no events or properties.

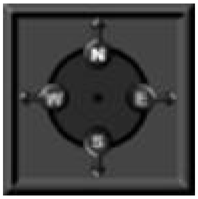
Usage

Place an *AutoHealer* component anywhere on your chassis. Every time you are stopped in the scenario, your health will begin to regenerate. Unfortunatly you must be a sitting duck while this is occuring and your enemies are much more likely to find you!

Note: The vehicle is considered stopped when its speed is less than 0.03 m/s and is spinning no more than 8 degrees/s.

BearingSensor

A sensor that will tell you if you are left of, right of, or directly heading for a reference bearing.



Category:

Physical

Properties:

DeltaBearing, FuzzyAngle, RefBearing, TrueBearing

Events:

Change, LeftOfRef, OnRef, RightOfRef

Grp	*Nav/Comm*
Wgt	*10*
Pts	*30*

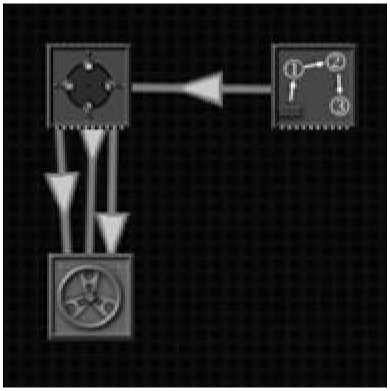
Description

The *BearingSensor* will translate a bearing into navigation directions such as left of bearing, right of bearing or directly on course. You supply the **RefBearing** through another component (such as the *WaypointSensor*) or by setting a value in the property box.

BearingSensor will trigger events: **LeftOfRef**, in which case you may want to steer right, **RightOfRef**, in which case you may want to steer left, or **OnRef**, in which case you can steer straight ahead.

Since it is impossible to get exactly the desired bearing, a **FuzzyAngle** property is used to say how much slop you are willing to tolerate. A setting of 5 degrees means you will allow your vehicle to be off from the reference bearing by as much as 5 degrees more and 5 degrees less.

Note that +90 degrees is LEFT, and -90 is RIGHT.



Usage

Use the *BearingSensor*, *Steering* and *WaypointSensor* to steer your vehicle towards a waypoint.

Connect three wires from the *BearingSensor* to *Steering*. One wire should read: when the *BearingSensor* triggers **LeftOfRef** set the *Steering* property **Angle** to the value -15 degrees (to the right).

The second wire should read: when the *BearingSensor* triggers **RightOfRef** set the *Steering* **Angle** to the value +15 degrees (to the left).

The third wire should read: when the *BearingSensor* triggers **OnRef** set the *Steering* **Angle** to 0 degrees (straight ahead).

Click on the *BearingSensor* if you would like to set the **RefBearing** directly, or create a wire from another component, such as the *WaypointSensor*, to the *BearingSensor*. This wire will allow you to set the output bearing of the *WaypointSensor* to the **RefBearing** of the *BearingSensor*.

See the Usage notes for the *WaypointSensor* for more information on using that component.

Brake

A component that slows your vehicle down.

Category:

Physical

Properties:

Pressure

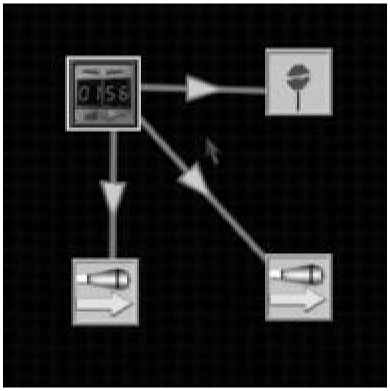
Events:

None

Grp .. Movement
Wgt 20
Pts............. 10

Description

The *Brake* causes your vehicle to slow down by applying friction at the brake's location. The **Pressure** determines how much friction is applied. To turn the brake off, set the **Pressure** to 0. This component is especially useful for stopping hover vehicles in the quickest time possible.



Usage

Place a *Brake* on the front left square of a hover chassis. Place a *Thruster* on the rear right and one on the rear left of the vehicle.

In the wiring screen, add a *Timer*.

Click on the *Thruster* to set its initial **Thrust** to 100%. Connect a wire from the *Timer* component to the *Brake* that says, when the *Timer* triggers **Tick** set the *Brake*'s **Pressure** to 100. Similarly, create a wire from the *Timer* to each *Thruster* turning them off when the *Timer* ticks.

Set the *Timer*'s initial **TickTime** to 4 seconds.

The vehicle should go at full thrust for 4 seconds, then you should see the vehicle come to a screeching halt and the *Thrusters* turn off.

Broadcast

A component used to reproduce activate events.

Category:
Logical

Properties:
Trigger

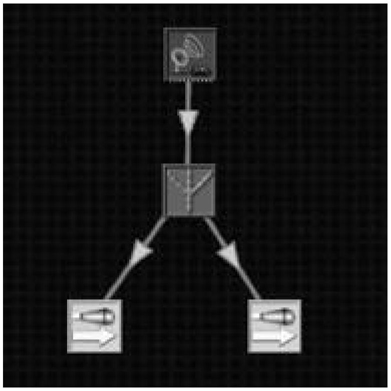
Events:
Set

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

The *Broadcast* component is used to consolidate wires when you would like one action to trigger many other events. You can label this component with the a name that will help remind you of its function.

There is only one event, **Set**, which triggers when the **Trigger** property is set.



Usage

One use of the *Broadcast* component is to help organize wires in a navigation system. Let's say that you have a hovercraft with two *Thruster*s, one on the right side of your vehicle and one on the left. Drag a *Broadcast* component on to your wiring screen and label it 'forward'.

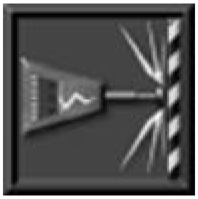
Create two wires from the *Broadcast* icon to each of the *Thruster*s. These wires should read: when the *Broadcast* triggers **Set**, set the *Thruster*'s **Thrust** to 100%.

Now connect another component, a *MediumRadar* for instance, to the 'forward' *Broadcast*. When the *MediumRadar* **TurnsOn**, activate the *Broadcast* **Trigger**. Now when the radar turns on it triggers both *Thruster*s to turn on.

Similarly, add another *Broadcast* component to the screen and label it 'turnRight'. Create a wire to the left *Thruster* which will turn it on to +100%, and create a wire to the right *Thruster* which will turn it on -100%. Now when you want to steer your hovercraft to the right, you connect a wire to this 'turnRight' *Broadcast* icon.

BumpSensor

A sensor that detects a collision between your vehicle and another object.



Category:

Physical

Properties:

FilterPlug

Events:

Bump, PlugIn

Grp	*Sensors*
Wgt	*30*
Pts	*10*

Description

The *BumpSensor* will trigger a **Bump** event whenever your vehicle collides with something in the room. The **Bump** event can then be used to activate other components, such as reverse thrust or make a noise.

The *BumpSensor* is not dependent on its location. You can place it anywhere on your vehicle and it will detect collisions at any location on the vehicle's body.

Usage

Use the *BumpSensor* along with *Speaker* to make a sound when you hit something.

Start with a HoverS (small hovercraft chassis). Click on the ***Component*** button and drag two *Thruster*s, a *BumpSensor*, and a *Speaker* onto your vehicle.

Click on the *Thrusters* and set the **Thrust** to 100%. Create a wire from the *BumpSensor* to the *Speaker*. This wire should read: when the *BumpSensor* triggers **Bump**, set the *Speaker*'s **PlaySound** value to 1 (or pick a different sound number).

When you hit ***Go***, the hovercraft will take off, and when it hits something you should hear the desired sound.

Compare

A component that will compare two numeric values. This component can be set to trigger on **GreaterThan**, **LessThan**, or **EqualTo**.

Category:

Logical

Properties:

Calc, Input1, Input2

Events:

EqualTo, GreaterThan, LessThan

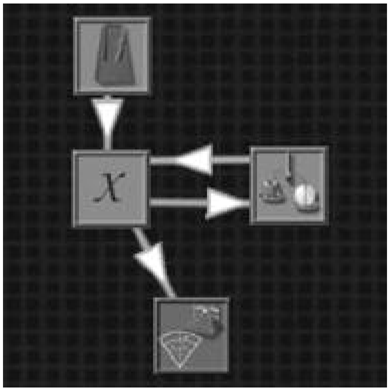
Description

Grp	*Logic*
Wgt	*0*
Pts	*0*

The *Compare* component will compare its two inputs, **Input1** and **Input2**, and tell you if **Input1** is **GreaterThan**, **LessThan**, or **EqualTo Input2**.

If the **Calc** property is true (the default), then *Compare* will trigger one of these three events each time you set a new value for either **Input1** or **Input2**.

But if you set the **Calc** property to false, then *Compare* will trigger one of its events only when you explicitly set **Calc** again (usually to false, if you want to leave it in manual-recalculate mode).



Usage

Use the *Compare* component to scan a radar from -45 degrees to +45 degrees. Increment a *Variable* which sets the radar angle, and when it gets to +45, then set it back to -45.

Start with any chassis and add a *MediumRadar*. Then go to the ***Wiring*** screen and add a *LoopTimer*, a *Variable*, and a *Compare* component.

First create the radar angle stepper by using a *LoopTimer* and *Variable*. Add a wire from the *LoopTimer* to the *Variable* that reads: when the *LoopTimer* triggers **Tick**, set the *Variable*'s **IncrementBy** property to a value of +5.

Click on the *LoopTimer* and set its **TickTime** to 0.5. Now, every half second, the *Variable* will increment by 5 degrees. The next step is to connect this to the angle of the radar so it will rotate.

Add a wire from the *Variable* to the *MediumRadar* that reads: when the *Variable* triggers **Change**, set the *MediumRadar*'s **Angle** property to the *Variable*'s **Output** property.

Next set up the *Compare* component to watch the output of the *Variable*. One input of the *Compare* will be the angle of the *MediumRadar* and the second input will be a constant value of +45 degrees.

The first wire is from the *Variable* to the *Compare*, and it reads: when the *Variable* triggers **Change**, set the *Compare*'s **Input1** to the *Variable*'s **Output** property.

To set the constant value for **Input2**, simply click on the *Variable* and set its **Input2** property to +45.

The final wire is from the *Compare* back to the *Variable*. It reads: when the *Compare* triggers **GreaterThan** (when input1 is greater than input2), then set the *Variable*'s **Input1** property to a value of -45.

Try it out!

Compass

A component that tells you the vehicle's orientation in the world.

Category:

Physical

Properties:

Bearing, RefBearing

Events:

Change

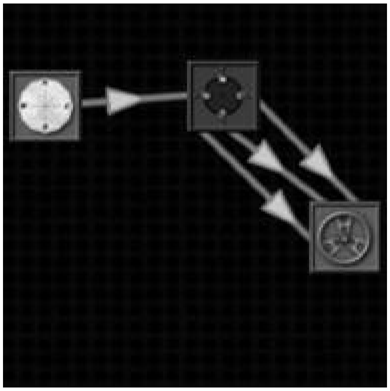
Grp.Nav/Comm
Wgt 10
Pts............ 30

Description

The *Compass* allows you to navigate in a particular direction in the world.

Set its **RefBearing** to the desired direction (where 0 is North, 90 is West, 180 is South and -90 is East).

The *Compass* gives you the vehicle-relative **Bearing** to that world direction. Every time the **Bearing** changes, *Compass* triggers the **Change** event.



Usage

Usage: If you want your vehicle to head due West, place a *Compass* and a *BearingSensor* on the vehicle.

In the wiring screen, click on the *Compass*, and set its **RefBearing** to 90 degrees for West.

Create a wire from the *Compass* to the *BearingSensor* that says, when the *Compass* triggers **Change**, set the *BearingSensor*'s **RefBearing** to the **Bearing** property of the *Compass*.

Now you can use the *BearingSensor* as usual to navigate your vehicle in that direction. (See help for *BearingSensor* if needed).

CopLight

A cop car light bar with a siren.

Category:

Physical

Properties:

Play

Events:

None

Grp........Extras
Wgt 0
Pts.............. 0

Description

This component is basically decorative, but it has a **Play** property and will sound a siren sound if you activate it. It does not generate any events.

Usage

When you bump into something, play the cop sound.

Add a *BumpSensor* and *CopLight* to your vehicle and create a wire from the *BumpSensor* to the *CopLight*. This wire should read: when the *BumpSensor* triggers **Bump**, activate **Play** on the *CopLight*.

Deadweight

A large piece of lead used to add extra weight to your vehicle.

Category:
Physical

Properties:
None

Events:
None

Description

Grp	*Movement*
Wgt	*100*
Pts	*0*

The *Deadweight* component simply adds weight to your vehicle. There are no properties to set and it will not trigger any events.

Usage

This component adds 100 grams to the weight of your vehicle. You will probably notice the difference most on a hovercraft.

Adding weight to a hovercraft will make it harder to push around, or give it more momentum when shoving other hovercrafts.

DebugMessage

Allows you to display a message string while the scenario is running

Category:

Logical

Properties:

Color, Message

Events:

None

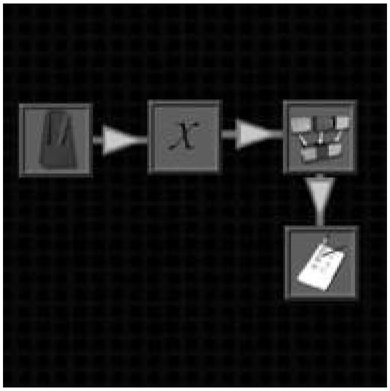
Grp. Debugging
Wgt 0
Pts.............. 0

Description

The *DebugMessage* component allows you to display an informative message on the screen while the scenario is running. It will display whatever message you specify, up to a certain length limit. Note that if you copy a numeric property to the **Message**, it will automatically be converted to a string. See the *Splice* component to put multiple *DebugMessages* together.

You can also set the **Color** in which the message should be displayed.

IMPORTANT: You can turn all debug messages on or off by pressing F5.



Usage

Show the elapsed time in the scenario in seconds.

Use a *LoopTimer*, *Variable*, *DebugMessage*, and *Splice* components. Create a wire from the *LoopTimer* to the *Variable* that reads: when the *LoopTimer* **Ticks**, set the *Variable*'s **IncrementBy** to 1.

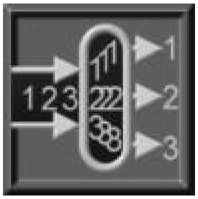
Click on the *LoopTimer* to set its **TickTime** to 1 second. Click on the *Splice* component and set its **Part1** property to "Time: " and set its **Part3** property to "seconds."

Create a wire from the *Variable* to the *Splice* component. It should read: when the *Variable* **Changes**, set the *Splice*'s **Part2** to the *Variable*'s **Output**.

Finally, create a wire from the *Splice* to the *DebugMessage* component. This should read: when the *Splice* **Changes**, set the *DebugMessage*'s **Message** to the *Splice*'s **Message**.

Demux

Triggers one of several events, depending on input value



Category:

Logical

Properties:

Input, Maximum, Remainder

Events:

AboveRange, Set0, Set1, Set2, Set3, Set4, Set5, Set6, Set7, Set8

Description

Grp ....Logic
Wgt 0
Pts 0

Demux accepts an integer **Input** and uses it to trigger one of several events, based on its value. If **Input** is 1, *Demux* triggers **Set1**; if 2, it triggers **Set2**, and so forth up to **Set8**.

You can think of *Demux* as having either a range of 1 - 8, or a range of 0 - 7, whichever makes most sense to you. The key is that **Set0** and **Set8** are two different names for the SAME event. So if you send the value 0 to *Demux*, it will trigger both **Set0** and **Set8**. If you send the value 8 to *Demux*, it will trigger both **Set0** and **Set8**. So you can think of it as supporting either 1 - 8 or 0 - 7 ... but you can't think of it as supporing 0 - 8.

Note that you can cascade *Demux* components, using the **Maximum** property, the **Remainder** property and the **AboveRange** event. **Remainder** is always set to (**Input** - 8).

Usage

Suppose you want the first *Demux* to handle the values 1 through 8, with the second *Demux* handling values 9 through 12.

To do that, connect the first *Demux* to the second *Demux* with a wire triggered by the first *Demux*'s **AboveRange** event, and make that wire copy the first *Demux*'s **Remainder** to the second *Demux*'s **Input**. Then (on the Wiring Screen) set the first *Demux*'s **Maximum** to 8.

Now, if you send the value 9 to the first *Demux*, it will recognize that this is greater than its **Maximum** and trigger **AboveRange** rather than triggering any of its **Set1** through **Set8** events. The wire fired by **AboveRange** will copy **Remainder** -- in this case, (9 - 8) or 1 -- to the second *Demux*'s **Input**, causing the second *Demux* to trigger its **Set1** event.

Of course, you can cascade an arbitrary number of *Demux* components this way.

You can also cascade them using a 0 - 7 rule, where the first *Demux* handles values 0 through 7, the second one handles values 8 through 15, and so forth. In that case, wire them together the same way, but set each one's **Maximum** to 7 rather than to 8. That way, if the first *Demux* sees the value 8, it will trigger **AboveRange** rather than any of its **Setx** events, **Remainder** will be set to 0, and the second *Demux* will trigger its **Set0** event.

If you set **Maximum** to 0, then *Demux* will assume that it is not being cascaded. In that case, it will divide its **Input** by 8 and use the remainder to trigger one of its **Setx** events. (If you send it the value 8, the remainder of (**Input** / 8) is 0, which will trigger **Set8** as well as **Set0**.)

Maximum values other than 0, 7 and 8 are accepted, but we regard them as uninteresting. Use at your own risk.

Divide

A component that divides Input1 by Input2 and will give you the Quotient and Remainder.

Grp	*Math*
Wgt	*0*
Pts	*0*

Category:

Logical

Properties:

Calc, Input1, Input2, Output, Quotient, Remainder

Events:

Change, Set

Description

You can use this component either to divide two numbers or to do modulo (clock) arithmetic.

The **Change** event is triggered whenever the output changes. The **Set** event is triggered whenever either input is set.

Quotient is the result of dividing **Input1** by **Input2**, using integer math. **Remainder** is the remainder.

So if **Input1** is 11, and **Input2** is 2, the **Quotient** will be 5 and the **Remainder** will be 1.

Output is the result of dividing **Input1** by **Input2**, using floating-point math. In the example shown above, **Output** will be 5.5.

Note that *Divide* uses the **Calc** mechanism: if **Calc** is left 'true', each time you set either **Input1** or **Input2** you might get **Set** or **Change** events. If you set **Calc** to 'false', then *Divide* only recomputes **Output** (and possibly triggers **Set** or **Change**) when you set **Calc**.

Usage

Divide can be used to do things like find an average. For example, to average two values, connect them to the two inputs of an *Add* component. The **Output** of the *Add* goes to **Input1** of a *Divide*, and its **Input2** is set to 2. The **Quotient** will be the average of the two sensors.

Divide is also good for keeping a number within a specified range. You can "normalize" an angle to the 0-360 range by dividing by 360 and using the **Remainder**.

See the *Subtract* component for an example that also uses *Divide*.

You can also use *Divide* to "scale" some other component's input. For instance, suppose that on some event you want to set a *Variable* to the value 5.6. It's easy enough to draw a wire to set the *Variable*'s **Input** -- but how do you coax the user interface to let you set the value 5.6?

It's harder than it might appear. One solution is to draw a wire that sets a *Divide*'s **Input1** to 56, where **Input2** has already been set to 10. Then, when *Divide* triggers **Set** (or **Change**, depending on the situation) copy its **Output** to the *Variable*'s **Input**.

Filter

A filter that is used to discriminate between friend, foe, projectile and other.

Category:

Logical

Properties:

EnemyVehicle, FilterSocket, Other, Projectile, State, Teammate

Events:

Change, TurnOff, TurnOn

Description

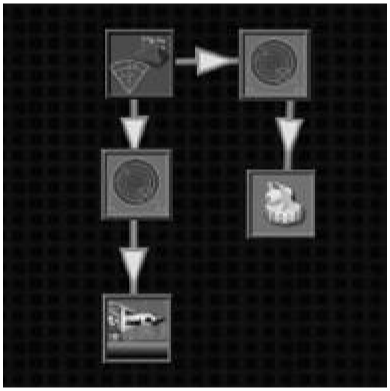
Grp	*Logic*
Wgt	*0*
Pts	*0*

The *Filter* component is used to differentiate what a sensor has detected. Sensors that have a **FilterPlug** property and a **PlugIn** event allow you to attach a filter to them. Then the filter will trigger events only for properties that you are interested in.

The properties you can filter are **EnemyVehicle**, **Teammate**, **Projectile** and **Other**. If you set one of these properties to ON (1), then the *Filter* will trigger the **TurnOn** event if an object of that kind is detected by the sensor component.

Other events of the filter include **TurnOff** which will fire when the sensor stops seeing the object, and **Change** which will fire whenever the sensor's state changes, turning either on or off.

Another property of the filter is **State** which will tell you which state the filter is currently in: ON (1) or OFF (0).



Usage

Use two *Filter*s with your radar to fire a rocket at an enemy, and a laser at a projectile.

Drop a *MediumRadar*, two logical *Filters*, a *Laser* and a *RocketLauncher* on a vehicle. Create a wire from the radar to the first *Filter*. The default wiring properties don't need to be changed: when the *MediumRadar* triggers **PlugIn**, set the *Filter*'s **FilterSocket** to the *MediumRadar*'s **FilterPlug**.

This event only happens once at the start of the scenario and serves to connect the radar to the filter. You can connect multiple filters to the same radar.

Create a wire from the radar to the second *Filter* with the default wire settings.

Next, click on the first *Filter* and set the desired filtering for **EnemyVehicle**, **Projectile**, **Teammate**, or **Other**. For this example, set the **EnemyVehicle** to 1 and the others to 0. Click on the second *Filter* and set the **Projectile** to 1 and the others to 0.

Create a wire from the first *Filter* to your *RocketLauncher*. When the *Filter* triggers **TurnOn**, activate the **Fire** property of the weapon. It should now fire only at the enemy and nothing else.

Lastly, create a wire from the second *Filter* to your *Laser*. When the *Filter* triggers **TurnOn**, activate the **Fire** property of the *Filter*. The *Filter* will now only fire at projectiles and not other vehicles.

Filter_IFF

A filter (Identify Friend or Foe) that is used to discriminate between friend, foe, and projectile.

Category:

Physical

Properties:

EnemyVehicle, FilterSocket, Other, Projectile, State, Teammate

Events:

Change, TurnOff, TurnOn

Grp..... Sensors
Wgt 10
Pts............ 40

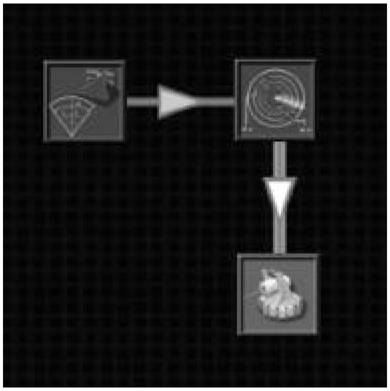
Description

The *Filter_IFF* component is used to differentiate what a sensor has detected. Sensors that have a **FilterPlug** property and a **PlugIn** event allow you to attach a filter to them. Then the filter will trigger events only for properties that you are interested in.

The properties you can filter are **EnemyVehicle**, **Teammate**, **Projectile** and **Other**. If you set one of these properties to ON (1), then the *Filter_IFF* will trigger the **TurnOn** event if an object of that kind is detected by the sensor component.

Other events of the filter include **TurnOff** which will fire when the sensor stops seeing the object, and **Change** which will fire whenever the sensor's state changes, turning either on or off.

Another property of the filter is **State** which will tell you which state the filter is currently in: ON (1) or OFF (0).



Usage

Use a *Filter_IFF* to avoid shooting your own teammates. Drop a *MediumRadar* and a *Filter_IFF* on a vehicle. Create a wire from the radar to the filter. The default wiring properties don't need to be changed: when the *MediumRadar* triggers **PlugIn**, set the *Filter_IFF*'s **FilterSocket** to the *MediumRadar*'s **FilterPlug**. This event only happens once at the start of the scenario and serves to connect the filter to the radar. (You can't connect more than one radar to a filter.)

Next, click on the *Filter_IFF* and set the desired filtering for **EnemyVehicle**, **Projectile**, and **Teammate**. For this example, set the **EnemyVehicle** to 1 and the others to 0.

Now create a wire from the *Filter_IFF* to your weapon. When the *Filter_IFF* triggers **TurnOn**, activate the **Fire** property of the weapon. It should now fire only at the enemy and not your teammate or other projectiles.

Fireworks

A component that shoots off a brief fireworks display when activated.

Category:

Physical

Properties:

FireColor

Events:

None

Grp .... Extras
Wgt 0
Pts 0

Description

The *Fireworks* component will send a small burst of colored fireworks into the air when you set its **FireColor** property. The **FireColor** property can be set to any of the standard mode colors.

Usage

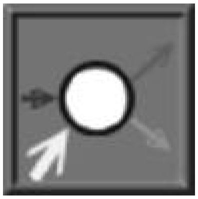
Set off blue fireworks every 5 seconds. To do this, add a *Fireworks* component and a *LoopTimer*.

Click on the *LoopTimer* component and set its default **TickTime** to 5 seconds. Add a wire from the *LoopTimer* to the *Fireworks* which reads: when the *LoopTimer* triggers **Tick**, set the *Fireworks* **FireColor** property to blue.

Note that since the fireworks colors match the mode colors, you can also try wiring a *ModeSwitch* to the *Fireworks*. Whenever you change modes, you'll fire off a burst showing your new mode color. It's a great debugging tool!

FSMMaster

Initial state of (and controller for) a Finite State Machine



Category:

Logical

Properties:

Mode

Events:

None

Grp .... Logic
Wgt 0
Pts............... 0

Description

A state machine allows you to define different states and the transitions between the states. For instance with the Hot Potato challenge, you may define a state that seeks the opponent, a state to move away from the opponent, and a state to back away from the wall when you bump into it. The *FSM* components let you use modes and colored wires to define the transitions.

The *FSMMaster* defines the first or initial state, and the *FSMState* components are used for the other states. Link all of the states together in the order you desire if you want the states to progress in order. Every time you set the **Mode** on the *FSMMaster* it will transition to the next state. In this example you would use all gray wires and set the **Mode** to gray.

Usage

For a more complicated state machine where there are multiple transitions out of the states, use colored wires and modes (see the *ModeSwitch* for more info on modes). For instance your robot may be in state 1, waiting for sensor inputs. If it gets a bump trigger it should go to state 2, shown with a red wire. If it gets a radar ping, it should go to state 3, shown with a green wire.

To cause the state machine to make a transition, the bump sensor would be connected to the *FSMMaster*'s **Mode** property, setting the mode to red on bump. The *FSMMaster* combines the behaviors of *ModeSwitch* and *FSMState*: when you set **Mode**, the *FSMMaster* does indeed set the vehicle's wire mode as a side effect. Then it causes the currently-active *FSMState* component to trigger **TurnOff**, as above. So if State 1 is active and the *BumpSensor* triggers, state 1's **TurnOff** event will fire the red wire to State 2's **Trigger**. But if the Radar triggers and sets the *FSMMaster*'s **Mode** to green instead, State 1's **TurnOff** will fire the green wire to State 3's **Trigger**.

Whenever the state machine transitions to a particular *FSMState*, that component sets its **State** to 1 and triggers its **TurnOn** and **Change** events. When the machine transitions away to another *FSMState*, the old *FSMState* sets its **State** to 0 and triggers its **TurnOff** and **Change** events.

By convention, you connect the **TurnOff** event on one *FSMState* component to the **Trigger** property of another to define a transition.

Because *FSMMaster* is the initial state of the machine, its **TurnOn** is triggered at vehicle startup.

You can, if need be, forcibly reset the state machine to some particular state by directly activating that *FSMState*'s **Trigger**. For instance, you can reset the machine to its initial state by activating *FSMMaster*'s **Trigger**. In that case, the *FSMState* component that was previously active will set its **State** to 0 and trigger **Change**. But because there are typically wire arcs connected to **TurnOff** that would activate other *FSMState*s, in the forcible-reset case the previously-active *FSMState* does not trigger its **TurnOff** event.

If you set *FSMMaster*'s **Mode** to a particular mode color, but there is no wire of the corresponding color connecting the currently-active *FSMState*'s **TurnOff** event to some other *FSMState*'s **Trigger** property, then the currently-active *FSMState* will trigger its **TurnOff** event -- but it will remain active: it will neither set its **State** to 0 nor trigger its **Change** event.

If you create a wire to a *Broadcast* component and then back to itself, the *FSMState* triggers its **TurnOn**, but does not change its **State** or trigger **Change**.

So for a given *FSMState*:

- you get **TurnOn** whenever you| activate its **Trigger**| - if some other *FSMState* was| previously active, you get **Change**| to **State** = 1| - as long as it's active, you get| {TurnOff} whenever you set| *FSMMaster*'s **Mode**| - you get **Change** to **State** = 0| only when some other *FSMState*| triggers **TurnOn**.

The Finite State Machine components give the vehicle designer a more powerful decision-making mechanism than using modes alone. With *ModeSwitch*, each vehicle mode in effect replaces the previous mode, and the vehicle itself does not track any history of the sequence of modes that have been set. But a state machine allows the vehicle to respond to particular combinations of events, or particular sequences, in distinct ways.

A classic Finite State Machine has a "final" state, and when it reaches that state, computation ceases. Clearly our *FSMState* state machine should continue running as long as the vehicle itself is operational; but you can make the vehicle respond to any interesting state of the machine by drawing a wire from that *FSMState*'s **TurnOn** event.

Trigger a different color fireworks for each of three states of a FSM. When the vehicle hits a wall, the *BumpSensor* will force the state back to state 3. For this example we will use a *LoopTimer* to walk through the states in order. We don't need to define modes or colored wires. See the diagram on the right.

Start with a hovercraft and two *Thrusters*. Add a *Fireworks* component and a *BumpSensor*.

In the Wiring screen add a *LoopTimer*, *FSMMaster*, and two *FSMStates*.

Create a wire from the *FSMMaster* to the first *FSMState*; from the first *FSMState* to the second one; and from the second one back to the master. The default wiring uses the {TurnOff} event of the first state to {Trigger} the next state.

Next create a wire from each of the state components to the *Fireworks* component. These wires should use the **TurnOn** of the state to set the **FireColor** property to a different color (red, green, and blue, for instance).

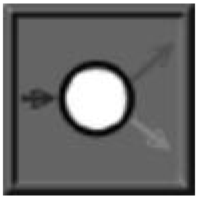
Create a wire from the *BumpSensor* to one of the *FSMState* components to activate its **Trigger** when the *BumpSensor* senses **Bump**. So this color fireworks will go off when the vehicle hits the wall representing the case where you want to be in a particular state whenever there is a collision.

Add a wire from the *LoopTimer* to the *FSMMaster* to force the state machine to move onto the next state every 2 seconds. This wire reads, when *LoopTimer* triggers **Tick**, set the *FSMMaster*'s **Mode** to gray. Click on the *LoopTimer* to set its default **TickTime** to 2 seconds.

Also, set the *Thrusters* to 50% or so. While the vehicle is making its way across the room, it will display fireworks of a different color for each state it enters. When it hits the wall it will go to the state you defined and then continue from there at the next ticktime.

FSMState

One state of a Finite State Machine



Category:

Logical

Properties:

State, Trigger

Events:

Change, TurnOff, TurnOn

Description

See *FSMMaster* for information on the *FSMState*.

Grp	*Logic*
Wgt	*0*
Pts	*0*

Usage

HealthPack

A component that restores your vehicle to maximum HP.

Category:

Physical

Properties:

Trigger

Events:

None

Grp	*Defense*
Wgt	*85*
Pts	*45*

Description

The *HealthPack* restores your vehicle's health to its maximum when **Trigger** is recieved. This component can only be used once each time the vehicle is spawned.

Usage

Use the *HealthPack* component in conjunction with the *HealthSensor*. Wire the *HealthSensor* to the *HealthPack* using the *HealthSensor*'s CriticalDamage event to trigger the *HealthPack*. When your robot's level of health gets too low, your robot will regenerate one time only.

HealthSensor

A sensor that detects the health remaining on your vehicle.

Category:

Physical

Properties:

Critical_pct, Healthy_pct, Output

Events:

Change, CriticalDamage, Repaired

Description

Grp	*Sensors*
Wgt	*20*
Pts	*10*

The *HealthSensor* component gives you the number of health points remaining on your vehicle.

The **Change** event is triggered whenever the vehicle is damaged or healed.

Set the **Critical_pct** property (critical percentage) to the percentage of health points that you would consider your vehicle in critical condition. The **CriticalDamage** event will fire when the health goes below **Critical_pct**.

You can also set the **Healthy_pct** property, and the **Repaired** event will fire when the health goes above **Healthy_pct**.

The **Output** property gives you the current number of the vehicle's health points.

Usage

See the *HealthPack* help for usage of the *HealthSensor* component.

Joystick

A component that will allow you to get joystick events.

Category:

Logical

Properties:

None

Events:

Change

Grp	*Debugging*
Wgt	*0*
Pts	*0*

Description

The *Joystick* component allows you to receive joystick events in the game. The position of the joystick is split and kept in two parts. The **JoystickAxisX** and **JoystickAxisY** contain the x and y coordinates, respectively.

This field is set up similarly to a Cartesian coordinate system, to the left and down are negative values and to the right and up are positive values. **JoystickAxisX** is the left or right position and **JoystickAxisY** is the up or down position. Each range goes from -100 to 100.

Usage

The *Joystick* can be used on a wheeled rover to create a simple, yet very effective, navigation system. First, drag an *Engine* and *Steering* on to your rover.

The Y-axis will be used to control the rover's speed and no fixing is needed so you can directly wire the *Joystick* to the *Engine*. When the joystick changes, set the *Engine*'s **Throttle** to the **JoystickAxisY**

The X-axis will control the *Steering*, unfortunately the **JoystickAxisX** needs a little fixing before running it into the *Steering*. Add a *Divide* to your rover and wire the *Joystick* to it. When the *Joystick*'s position changes, set **input1** to **JoystickAxisX**. Select the *Divide* and change **input2** to -10.

The negative is necessary because the *Joystick* returns values from -100 to 100 and the *Steering* goes from 30 to -30 so each value must be negated. Also, we divide by 10 to make controlling the rover easier. By dividing, each input will cause a smaller turn, which is easier to control. Try different values to find which works best for you!

JoystickButton

A component that will monitor a joystick button.

Category:

Logical

Properties:

ButtonID

Events:

Change, TurnOff, TurnOn

Grp	*Debugging*
Wgt	*0*
Pts	*0*

Description

The *JoystickButton* component is used to control action in the arena with the Joystick buttons. For each button you would like to use, you need separate *JoystickButton* component.

Set the **ButtonID** to a number between 0 and 31 to determine which button this component is monitoring. Typically ButtonID 0 is the main trigger or fire button. Button IDs are defined in the Windows interface for your joystick.

The **TurnOn**, **TurnOff**, and **Change** events will be triggered when the button is pressed and released.

Usage

To make a rover that will shoot when the primary button on you joystick is pressed, place a *RocketLauncher* and a *JoystickButton* on your rover.

Set the *JoystickButton*'s **ButtonID** property to 0. This usually corresponds to the trigger or primary joystick, but you may have to try a couple of different values. Drag a wire from the *JoystickButton* to the *RocketLauncher*. Whenever the **TurnOn** eventis triggered, have the *RocketLauncher* **Fire**.

KarmaVariable

Acts exactly like a regular variable, but the value it holds persists even after a rover has been destroyed and repawns

Category:

Logical

Properties:

IncrementBy, Input, Output, Query

Events:

Change, Querying, Set

Description

Grp	*Math*
Wgt	*0*
Pts	*0*

Like *Variable*, the *KarmaVariable* component stores a number that can be used by other components. It can be set up as a counter by setting the **IncrementBy** property. It can also be used to propagate changes between two similar components. See *Variable* for more information about its ordinary behavior.

What distinguishes *KarmaVariable* is that if the vehicle is killed in combat and subsequently resurrected, *KarmaVariable* remembers the last value it had prior to the vehicle's death.

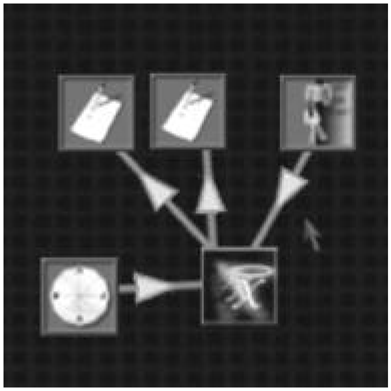
When a vehicle containing a *KarmaVariable* is first loaded into a scenario, its **Input** property is set according to the initial value specified on the Wiring Screen. Thereafter, the vehicle can use the *KarmaVariable* just as it would use a *Variable*.

But if the vehicle is killed, and then later resurrected, *KarmaVariable* behaves differently:

1. First, on resurrection, it ignores the initial value set for the **Input** property. The **Output** property retains the last value it had prior to the vehicle's death.

2. The *KarmaVariable* triggers both **Set** and **Change**.

The **Set** and **Change** Events are triggered so that any wires intended to keep downstream components in sync with the *KarmaVariable* will copy the last previous-life value to those components.



Usage

Store the compass setting of a vehicle before it dies and show that setting on the screen when the vehicle is resurrected.

Choose the Word War I scenario with a tough opponent so your dumb rover will get killed fairly easily. On your vehicle add a *KarmaVariable*, a *Compass*, two *DebugMessages*, and a *StartUp* component.

Connect the *Compass* to the *KarmaVariable* so that every **Change** event will get stored in its **Input**. Connect the *KarmaVariable* to the first *DebugMessage* and have it send its **Output** to the *DebugMessage* whenever the variable gets **set** (basically all the time).

Connect the *KarmaVariable* to the second *DebugMessage* and have it print out the **Output** only when it is **Queried**. Finally, add a wire from the *StartUp* component to the *KarmaVariable* that triggers **query** on **respawn**. So the second message will only update when the vehicle respawns and it should contain the last known compass setting of the vehicle.

KeySensor

A component that will allow you to use the Keyboard to trigger events in the world. It is not a legal component in most scenarios, but a good debugging tool.

Category:

Logical

Properties:

KeyLast, KeyList

Events:

Key1, Key2, Key3, Key4, Key5

Grp.	*Debugging*
Wgt	*0*
Pts	*0*

Description

The *KeySensor* was created to help in debugging or testing out various other components. It is not available in most scenarios, but you can use it in the ***TestingGround*** scenarios.

The *KeySensor* lets you define up to 5 keys that you can use in the scenario to fire a weapon or steer your vehicle.

Set the **KeyList** property to the string of 5 keys you would like to use. The default setting is "esdfc". Then connect wires from the *KeySensor* to other components.

The keys defined in the **KeyList** get mapped in order to events called **Key1**, **Key2**, **Key3**, **Key4**, and **Key5**. You can only use the standard printable keys on the keyboard.

Usage

The *KeySensor* is a great tool for helping you figure out how other components work, or in debugging something you have created. Use a *KeySensor* to play sound through the *Speaker* component.

Choose any type of Chassis and go into the ***Component*** screen. Drag a *Speaker* component onto your vehicle. Then go into the ***Wiring*** screen and drag a *KeySensor* onto the wiring screen.

Connect a wire from the *KeySensor* to the *Speaker*. This wire should read: when the *KeySensor* triggers **Key1**, set the *Speaker's* **PlaySound** property to 1.

Add a second wire from the *KeySensor* to the *Speaker*. This wire should read: When the *KeySensor* triggers **Key2** set the *Speaker's* **SoundPlay** property to 2.

Now when you hit ***GO***, your vehicle should play sounds 1 and 2 when you hit the "e" and "s" keys on your keyboard. You can redefine the keys you want to use by clicking on the *KeySensor* and changing the **KeyList**.

LargeEngine

An engine that can be used with a wheeled or treaded chassis.

Category:

Engine

Properties:

Throttle

Events:

None

Grp .. Movement
Wgt 300
Pts............. 30

Description

A *LargeEngine* component is used with wheeled or treaded vehicles for movement.

All three engines, small, medium and large, have a **Throttle** property which determines the engine speed. The range is -100% to +100%. A negative value will reverse the direction of your vehicle.

The *LargeEngine* is recommended for the large chassis. This is the heaviest engine.

Usage

For a wheeled vehicle, just add an engine and set its **Throttle**.

For a treaded vehicle, add an *Engine* as well as a *TreadControl*. The *Engine* and *TreadControl* work together to move the tank, but they don't need a wire between them.

Set the engine to 100%, then use the *TreadControl* to determine direction and speed (see *TreadControl*) of your vehicle.

Laser

A laser gun with a good range and a moderate repeat rate. Damage decreases with range.

Category:
Weapon

Properties:
Fire

Events:
None

Grp	*Weapons*
Wgt	*75*
Pts	*25*

Description

The *Laser* is a weapon which sends out a high powered laser pulse. Its property, **Fire**, is an activate property which will fire a pulse. It takes about 2.5 seconds to recharge.

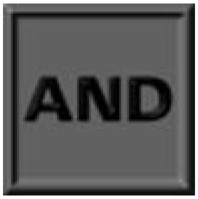
Usage

The *Laser* is an instant-hit weapon with a fairly long range, so it's good to use if you have an accurate system for tracking your opponent.

Refire rate: 2.5 seconds
Damage: 10 hit points maximum, varies with distance
Range: 10 meters
Speed: instant hit
Pivots: none

LogicalAND

A component that will tell you when two logical inputs are both true.



Category:

Logical

Properties:

InputA, InputB, State

Events:

Change, TurnOff, TurnOn

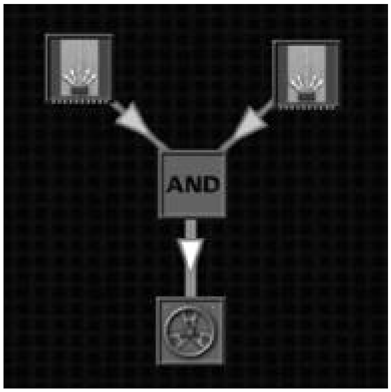
Description

Grp	*Logic*
Wgt	*0*
Pts	*0*

The *LogicalAND* has two inputs and one output. The output, **State**, will be 1 (true) when both **InputA** and **InputB** are 1. It will be 0 (false) if either input is 0.

There are three possible events that can be triggered from this component. **Change** will trigger whenever the output **State** changes, either from 1 to 0 or from 0 to 1. The **TurnOff** event will trigger when the **State** goes from 1 to 0, and the **TurnOn** event will trigger when the **State** goes from 0 to 1.

LogicalAND



Usage

Use the *LogicalAND* in conjuction with components which have a true or false event to ensure that two events have happened before affecting a third component. For instance, when two *TrackSensor*s are on the track, you can force the vehicle to steer straight ahead.

Choose a scenario with a track, such as ***Tutorial Track Sensors***, or ***Figure Eight Race***. Go into the ***Vehicle*** selection screen and choose a wheeled vehicle. In the ***Components*** selection screen, add two *TrackSensor*s, an *Engine*, and a *Steering* component. Go into the ***Wiring*** screen and add a *LogicalAND*.

Wire one *TrackSensor* to the *LogicalAND*. This wire should read: when the *TrackSensor* **Changes**, set the *LogicalAND*'s **InputA** equal to the *TrackSensor*'s property **State**. This means **InputA** will change whenever the *TrackSensor*'s output state changes.

Add a wire from the second *TrackSensor* to the *LogicalAND* to read: when the *TrackSensor* **Changes**, set the *LogicalAND*'s **InputB** equal to the *TrackSensor*'s property **State**. This track sensor is going to change **InputB** whenever it goes on and off the track.

Finally, add a wire from the *LogicalAND* to the *Steering* component. This wire should read: when the *LogicalAND* **TurnsOn** (which happens when both track sensors are on the track), set the *Steering* value to 0 (go straight ahead).

Please see the *TrackSensor* component help for more information on using these sensors.

LogicalNAND

A component that will tell you when either of its two inputs is false.

Grp .... Logic
Wgt 0
Pts.............. 0

Category:

Logical

Properties:

InputA, InputB, State

Events:

Change, TurnOff, TurnOn

Description

The *LogicalNAND* has two inputs and one output. The output, **State**, will be 1 (true) when either **InputA** or **InputB** are 0. It will be 0 (false) if both inputs are 1.

There are three possible events that can be triggered from this component. **Change** will trigger whenever the output **State** changes, either from 1 to 0 or from 0 to 1. The **TurnOff** event will trigger when the **State** goes from 1 to 0, and the **TurnOn** event will trigger when the **State** goes from 0 to 1.

Usage

This component is equivalent to wiring a *LogicalAND* component to a *LogicalNOT* component.

Truth Table:

A	B	NAND	NOR	XOR	XNOR
1	1	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	0
0	0	1	1	0	1

LogicalNOR

A component that will tell you when two logical inputs are both false.

Category:

Logical

Properties:

InputA, InputB, State

Events:

Change, TurnOff, TurnOn

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

The *LogicalNOR* has two inputs and one output. The output, **State**, will be 1 (true) when neither **InputA** nor **InputB** are 1. It will be 0 (false) if either input is 1.

There are three possible events that can be triggered from this component. **Change** will trigger whenever the output **State** changes, either from 1 to 0 or from 0 to 1. The **TurnOff** event will trigger when the **State** goes from 1 to 0, and the **TurnOn** event will trigger when the **State** goes from 0 to 1.

Usage

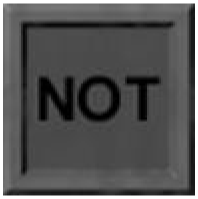
This component is equivalent to wiring a *LogicalOR* component to a *LogicalNOT* component.

Truth Table:

A	B	NAND	NOR	XOR	XNOR
1	1	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	0
0	0	1	1	0	1

LogicalNOT

A component that will give you the opposite of the input state.



Category:

Logical

Properties:

InputA, State

Events:

Change, TurnOff, TurnOn

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

The *LogicalNOT* component will generate an output **State** which is the opposite of its input setting, **InputA**. The **State** will be 1 (true) when the input is 0 (false).

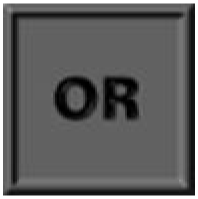
There are three possible events that can be triggered from this component. **Change** will trigger whenever the **State** changes, either from 1 to 0 or from 0 to 1. The **TurnOff** event will trigger when the **State** goes from 1 to 0, and the **TurnOn** event will trigger when the **State** goes from 0 to 1.

Usage

Use the *LogicalNOT* to invert the true or false signal from other components which give a logical output, such as the *LogicalAND*, *LogicalOR*, and components which have a **State** output property.

LogicalOR

A component that will tell you when either of its two inputs is true.



Category:

Logical

Properties:

InputA, InputB, State

Events:

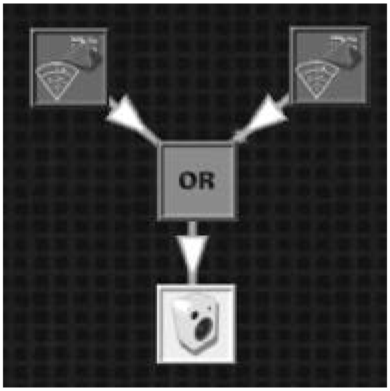
Change, TurnOff, TurnOn

Description

Grp	*Logic*
Wgt	*0*
Pts	*0*

The *LogicalOR* component has two inputs and one output **State**. The **State** will be 1 (true) if either of the inputs, **InputA** or **InputB**, is 1. The **State** will be 0 (false) only if both of the inputs are 0.

There are three possible events that can be triggered by this component. **Change** will trigger whenever the **State** changes from 1 to 0 or 0 to 1. **TurnOff** will trigger only when the **State** changes from 1 to 0, and **TurnOn** will trigger when the **State** changes from 0 to 1.



Usage

Use the *LogicalOR* to trigger an event when either of two conditions have been met. For instance, play a sound if either of your two radars sees the enemy vehicle.

Add two *MediumRadars* onto your vehicle, and a *Speaker*. Go to the ***Wiring*** screen and add a *LogicalOR* to your workbench.

Click on your left most radar and set its **Angle** property towards the left. Set the right most radar's **Angle** towards the right.

Create a wire from the left most *MediumRadar* to the *LogicalOr* which reads: when the *MediumRadar* triggers **Change**, set the *LogicalOR*'s **InputA** to the *MediumRadar*'s **State** property. This means the input to the OR will change with the output state of the radar.

Add a wire from the right most *MediumRadar* to the *LogicalOr* which reads: when the *MediumRadar* triggers **Change**, set the *LogicalOR*'s **InputB** to the *MediumRadar*'s **State** property. Now **InputB** follows the state of the right most radar.

Finally, add a wire from the *LogicalOr* to the *Speaker*. When the *LogicalOR* triggers **TurnsOn**, set the *Speaker*'s **SoundPlay** value to 3.

LogicalXNOR

A component that will tell you when two logical inputs are the same.

Category:

Logical

Properties:

InputA, InputB, State

Events:

Change, TurnOff, TurnOn

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

The *LogicalXNOR* component has two inputs and one output **State**. The **State** will be 1 (true) if the inputs are the same. The **State** will be 0 (false) if the inputs are different.

There are three possible events that can be triggered by this component. **Change** will trigger whenever the **State** changes from 1 to 0 or 0 to 1. **TurnOff** will trigger only when the **State** changes from 1 to 0, and **TurnOn** will trigger when the **State** changes from 0 to 1.

Usage

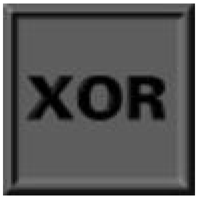
This component is equivalent to wiring a *LogicalXOR* component to a *LogicalNOT* component.

Truth Table:

A	B	NAND	NOR	XOR	XNOR
1	1	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	0
0	0	1	1	0	1

LogicalXOR

A component that will tell you when two logical inputs are different.



Category:

Logical

Properties:

InputA, InputB, State

Events:

Change, TurnOff, TurnOn

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

The *LogicalXOR* component has two inputs and one output **State**. The **State** will be 1 (true) if the inputs are different. The **State** will be 0 (false) if the inputs are the same.

There are three possible events that can be triggered by this component. **Change** will trigger whenever the **State** changes from 1 to 0 or 0 to 1. **TurnOff** will trigger only when the **State** changes from 1 to 0, and **TurnOn** will trigger when the **State** changes from 0 to 1.

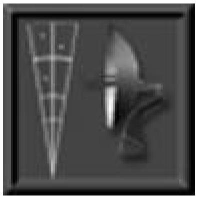
Usage

Truth Table:

A	B	NAND	NOR	XOR	XNOR
1	1	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	0
0	0	1	1	0	1

LongRangeRadar

A radar that can detect objects as far away as 15 meters with as much as 30 degrees scan width.



Category:

Physical

Properties:

Angle, FilterPlug, MaxRange, ScanWidth, State

Events:

Change, PlugIn, TurnOff, TurnOn

Description

Grp ..... Sensors
Wgt 30
Pts 30

Use the *LongRangeRadar* to detect vehicles or objects that are up to 15 meters away from your vehicle. Vary the distance you are scanning with the **MaxRange** property.

The **ScanWidth** property describes a cone-shaped area where the radar is scanning. The maximum angle is 30 degrees. The default is 10 degrees.

The **RadarAngle** property describes the direction that the radar is pointing. The default is 0 degrees, which means straight ahead. If you rotated the radar as you placed it, then 0 degrees refers to the direction that you rotated the component.

You can see all of these properties in action when you run your vehicle in a scenario. A cone-shaped area projecting from your vehicle shows what area the radar can 'see'.

The **PlugIn** event is used to connect to a *Filter_IFF* component for discriminating what object you have detected. The **FilterPlug** property is used to plug into the **FilterSocket** of the *Filter_IFF* component. See the help for *Filter_IFF* for more information.

Usage

Use one or more *LongRangeRadar* components to find other vehicles or objects in a scenario. You can wire these radars to movement or steering controls to head towards or away from a desired object.

See the *Filter_IFF* component to discriminate between friend and foe and other objects.

LoopTimer

A settable timer that automatically restarts after counting down.

Category:

Logical

Properties:

TickTime

Events:

Tick

Description

Grp ...... *Timing*
Wgt............ *0*
Pts............. *0*

The *LoopTimer* will count down from a desired number of seconds to zero. When it hits zero it will generate a **Tick** event and start again.

Use the **TickTime** property to set the desired count in seconds. A **TickTime** of zero will stop the *LoopTimer*.

Usage

The *LoopTimer* can be used to start or stop other components on a regular basis. Any time you want to fire off an event regularly, the *LoopTimer* will be a good solution.

For example, to play a sound every 3 seconds: add a *Speaker* and *LoopTimer* to your vehicle. Create a wire from the *LoopTimer* to the *Speaker* which reads: when the *LoopTimer* triggers **Tick**, set the *Speaker*'s **PlaySound** property to 2.

LoopTimer

Click on the *LoopTimer* and set the **TickTime** to 3. Click on the *Speaker* and set the **SoundGroup** to any number. Now every 3 seconds you will hear a sound play.

LootCarrier

A component that allows your vehicle to carry a flag, jewels or other 'loot'.

Category:

Physical

Properties:

Drop, LootColor, State

Events:

Change, TurnOff, TurnOn

Description

Grp	*Extras*
Wgt	*30*
Pts	*10*

Use a *LootCarrier* to carry the enemy's flag or to carry other loot depending on what the scenario has to offer.

The **LootColor** property describes what color loot your vehicle is carrying. The colors are the same ones defined by the *ModeSwitch* component.

The **State** property will be true (1) if the carrier is holding something and false (0) if it is not.

The **Drop** property is an activate property which will cause the vehicle to drop the loot.

The *LootCarrier* will trigger the **TurnOn** event if it picks something up; the **TurnOff** event if it drops or loses its loot; and the **Change** event if either of the other events is triggered.

Usage

Use the *LootCarrier* to pick up the enemy's red flag and shoot off fireworks when you get it.

Add a *LootSensor*, a *BearingSensor*, a *LootCarrier* and a *Fireworks* component to your vehicle along with appropriate movement components.

Click on the *LootSensor* component icon in the wiring screen and set the **LootColor** to red. Also connect the *LootSensor* to the *BearingSensor* to steer your vehicle towards the red loot (as described in the *LootSensor* help).

Now add a wire from the *LootCarrier* to the *Fireworks* which reads: when the *LootCarrier* triggers **TurnOn**, set the *Fireworks* **FireColor** property to 3 (or any color you would like). Or set the *Fireworks* **FireColor** to the *LootCarrier*'s **LootColor** and the fireworks that goes off will be the same color as the loot it picked up.

LootSensor

A sensor that will give you the bearing and distance to the 'loot' - jewels, enemy flag, or other good stuff.

Category:

Physical

Properties:

Bearing, Distance, LootColor

Events:

Change, NotFound

Description

Grp	*Sensors*
Wgt	*30*
Pts	*30*

The *LootSensor* is used to determine where the good stuff is. In some scenarios the 'Loot' may be the enemy's flag. In other scenarios it may be jewels or other items you might want to pick up.

Set the **LootColor** property to the color of the loot you are looking for.

The **Distance** and **Bearing** output properties can be used with other components to steer towards the loot. *LootSensor* will track loot even if it's moving (because someone else has picked it up with a *LootCarrier*).

The *LootSensor* will trigger a **Change** event whenever the **Distance** or **Bearing** to the loot changes.

It will trigger a **NotFound** event if there is currently no loot of the color you specified in the scenario. This can happen if you specify a color which isn't used for loot by the current scenario, or if you specify the color of a valid loot item but the item later disappears for one reason or another.

After triggering **NotFound**, the *LootSensor* won't trigger any more events until you set **LootColor** to the color of some existing loot, or until a loot object of the current **LootColor** appears in the arena. At that point, the *LootSensor* will start triggering **Change** events again.

Usage

Use the *LootSensor* to steer your vehicle towards the loot. Using a wheeled vehicle, add a *LootSensor*, a *Steering*, and a *BearingSensor*.

Create a wire from the *LootSensor* to the *BearingSensor* which reads: when the *LootSensor* triggers **Change**, set the *BearingSensor*'s **RefBearing** to the *LootSensor*'s **Bearing**.

Then create three wires from the *BearingSensor* to the *Steering* component to steer towards the loot.

The first wire reads: when the *BearingSensor* triggers **LeftOfRef**, set the *Steering* **Angle** to -15 (towards the right).

The next wire reads: when the *BearingSensor* triggers **RightOfRef**, set the *Steering* **Angle** to +15 (towards the left).

The third wire reads: when the *BearingSensor* triggers **OnRef**, set the *Steering* **Angle** to 0 (go straight ahead).

MachineGun

A rapid fire gun mounted on a pivoting turret.

Category:
Weapon

Properties:
Angle, Fire

Events:
None

Grp....Weapons
Wgt 100
Pts............. 15

Description

The *MachineGun* is used to inflict damage to your opponent. It can rotate 45 degrees to either side by setting its **Angle** property. When you activate its **Fire** property, it shoots off a burst of 10 bullets.

Usage

This is an instant-hit weapon; each individual bullet (there are ten per burst) does 3 hit points of damage. The gun sprays the bullets around in a cone, so the closer you are to your target, the more likely you are to have most of them hit the target.

Reload rate: 1 second
Damage: 3 hit points/bullet; 10 bullets over 1.2 seconds;
Range: 8 meters
Speed: instant hit
Pivots: +45 to -45 degrees

MediumEngine

An engine that can be used with a wheeled or treaded chassis.

Category:

Engine

Properties:

Throttle

Events:

None

Description

Grp . Movement
Wgt 200
Pts 30

A *MediumEngine* component is used with wheeled or treaded vehicles for movement.

All three engines, small, medium and large, have a **Throttle** property which determines the engine speed. The range is -100% to +100%. A negative value will reverse the direction of your vehicle.

The *MediumEngine* is recommended for the medium-sized chassis.

Usage

For a wheeled vehicle, just add an engine and set its **Throttle**.

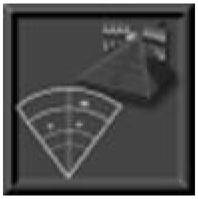
For a treaded vehicle, add an *Engine* as well as a *TreadControl*. The *Engine* and *TreadControl* work together to move the tank, but they don't need a wire between them.

MediumEngine

Set the engine to 100%, then use the *TreadControl* to determine direction and speed (see *TreadControl*) of your vehicle.

MediumRadar

A radar that can detect objects within 5 meters with as much as 90 degrees scan width.



Category:

Physical

Properties:

Angle, FilterPlug, MaxRange, ScanWidth, State

Events:

Change, PlugIn, TurnOff, TurnOn

Description

Grp	*Sensors*
Wgt	*30*
Pts	*30*

Use the *MediumRadar* to detect vehicles or objects that are up to 5 meters away from your vehicle. Vary the distance you are scanning with the **MaxRange** property.

The **ScanWidth** property describes a cone-shaped area where the radar is scanning. The maximum angle is 90 degrees. The default is 30 degrees.

The **RadarAngle** property describes the direction that the radar is pointing. The default is 0 degrees, which means straight ahead. If you rotated the radar as you placed it, then 0 degrees refers to the direction that you rotated the component.

You can see all of these properties in action when you run your vehicle in a scenario. A cone-shaped area projecting from your vehicle shows what area the radar can 'see'.

The **PlugIn** event is used to connect to a *Filter_IFF* component for discriminating what object you have detected. The **FilterPlug** property is used to plug into the **FilterSocket** of the *Filter_IFF* component. See the help for *Filter_IFF* for more information.

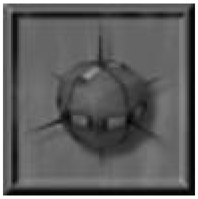
Usage

Use one or more *MediumRadar* components to find other vehicles or objects in a scenario. You can wire these radars to movement or steering controls to head towards or away from a desired object.

See the *Filter_IFF* component to discriminate between friend and foe and other objects.

MineLayer

A component that drops explosive proximity mines.



Category:
Weapon

Properties:
Fire

Events:
None

Grp	*Weapons*
Wgt	*75*
Pts	*20*

Description

The *MineLayer* drops a mine when fired; the mines explode on contact with any vehicle, so make sure you mount this thing to fire the mines away from your direction of motion!

The *MineLayer* has no events of its own. Activate its **Fire** property to drop a mine.

Usage

Use the *MediumRadar* looking behind your rover to leave a mine in the path of someone following you.

To do this add a *MediumRadar* to the back of your rover and set its **RadarAngle** to 180 (or -180) so it is looking backwards. [Alternatively, you can rotate the *MediumRadar* when you are placing it on the vehicle.]

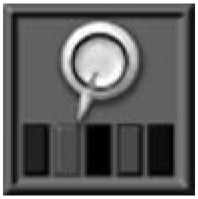
Add a *MineLayer* to the back of your rover as well and use the right mouse button to rotate it before dropping it on the grid.

In the ***Wiring*** screen, connect a wire from the *MediumRadar* to the *MineLayer* which reads: when the *MediumRadar* triggers **TurnOn**, activate the *MineLayer*'s **Fire** property.

Refire rate: 0.8 seconds
Damage: 40 hit points for direct hit
Range: 0 meters
Speed: 0 (doesn't move from where it lands)
Pivots: none

ModeSwitch

A component that will change your vehicle's 'mode' during a competition. You can define up to 7 modes, specified by the colored wires in the wiring screen.



Category:

Logical

Properties:

Mode

Events:

Change, Leave, Set

Description

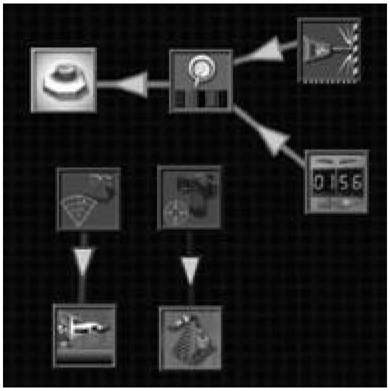
Grp ....Logic
Wgt 0
Pts 0

The *ModeSwitch* component allows you to define up to 7 different modes for your vehicle. One mode might be used for offensive behavior, another for defense.

Each mode is defined by colored wires in the wiring screen. When your vehicle is in 'red' mode, all the red wires will be active. When it changes to the 'blue' mode, all the blue wires are active.

The default wire color, grey, is ALWAYS active: grey wires continue to work in ALL other modes. If you aren't using modes, use grey wires.

Set the **Mode** property using an event from another component, such as a *BumpSensor* or *ProximityRadar*. The *ModeSwitch* will trigger a **Set** event whenever it gets set by another device. It will trigger **Change** whenever it gets set by another device and the mode actually changes. And it will trigger a **Leave** event just before a **Change** event when it is going to change modes.



Usage

The *ModeSwitch* can be used to help define distinct behaviors of a vehicle. For instance, one mode might be used for offense and another for defense. In this example we will use the red mode for offense and blue for defense.

Create a vehicle with a *RocketLauncher* and *MediumRadar* for its offense, and a *ProximityRadar* and *WeldingTorch* for defense. Also add a *RunningLight* which can be connected to the *ModeSwitch* to visually show you what mode your vehicle is in. In this example, use a *BumpSensor* to change modes.

First create a wire from the *ModeSwitch* to the *RunningLight*. The default wire properties don't need to be changed. It reads: when the *ModeSwitch* triggers **Change**, set the **Mode** (color) of the *RunningLight* to be the same as the **Mode** of the *ModeSwitch* .

Connect the *BumpSensor* to the *ModeSwitch*. This should read: when the *BumpSensor* triggers **Bumps**, set the *ModeSwitch*'s **Mode** to Red (choose the color red). Note that this wire should be grey so it can occur any time in the game.

Next connect the *MediumRadar* component to the *RocketLauncher*: when the *MediumRadar* triggers **TurnsOn**, activate the **Fire** property of the *RocketLauncher*. Once you have made this connection change the wire to red by clicking on the red box at the bottom of the property box. This wire will only fire when the vehicle is in "red" mode.

Similarly, connect the *ProximityRadar* to the *WeldingTorch* and set the wire color to Blue, so that this event only fires when the vehicle is in Blue mode.

In this example, a Timer is used to set the mode to blue after 10 seconds. Click on the Timer and set its **TickTime** to 10 seconds. Then connect it to the *ModeSwitch*: When the *Timer* triggers **Tick** set the *ModeSwitch*'s **Mode** to Blue.

After 10 seconds in the game, the mode will change to blue and the blue wires (as well as the grey) are now active. In this example the blue mode means this vehicle will now be using its *ProximityRadar* to look for objects and will fire the *WeldingTorch* if it finds anything.

Remember that the grey wires fire all the time. The colored wires fire only when your vehicle is in a specific mode. The colored tabs along the left side of the workbench will allow you to view only wires of the desired color. Click on the Red tab and you will only see Red (and grey) wires. Click on the grey tab to see all wires.

The *RunningLight* can help during a competition to let you see what mode your vehicle is in. Simply create a wire from the *ModeSwitch* to the *RunningLight*.

A final note: You cannot depend on the event of one component to both change the vehicle mode AND fire off an event of the new mode color.

For instance, if you want the **Bump** event of the *BumpSensor* to change the mode to red, AND you want the **Bump** event to fire the *RocketLauncher* while in red mode, you need to do this: connect a grey wire from the *BumpSensor* to the *ModeSwitch* to turn the mode to red when it feels a bump.

Then add a RED wire from the *ModeSwitch* to the *RocketLauncher* to fire the rocket. In other words, fire the colored event from the *ModeSwitch*, not from the *BumpSensor*.

Multiply

A component that multiplies its two inputs and generates an output equal to the product.

Category:

Logical

Properties:

Input1, Input2, Output

Events:

Change, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

Description

The *Multiply* component multiplies two inputs, **Input1** and **Input2**, and gives you the **Output** as the product of the inputs.

It will trigger a **Change** event when the output changes, and a **Set** event when either input gets a new value.

Usage

If you wanted to be able to set a random bearing at multiples of 90 degrees, set up a *Randomizer* to generate values from 0 to 3, then use a *Multiply* component to multiply the value by 90.

ProximityRadar

A radar for detecting objects within 3 meters with as much as 360 degrees scan width.

Category:

Physical

Properties:

Angle, FilterPlug, MaxRange, ScanWidth, State

Events:

Change, PlugIn, TurnOff, TurnOn

Grp	*Sensors*
Wgt	*30*
Pts	*30*

Description

Use the *ProximityRadar* to detect vehicles or objects that are a short distance from your vehicle. Vary the distance you are scanning with the **MaxRange** property.

The **ScanWidth** property describes the area where the radar can scan. The maximum angle (and default angle) is 360 degrees, which surrounds your vehicle completely. You can decrease this angle and create a cone shaped area for the radar to scan.

The **RadarAngle** property describes the direction that the radar is pointing. The default is 0 degrees, which means straight ahead. If you rotated the radar as you placed it, then 0 degrees refers to the direction that you rotated the component.

If you are using a **ScanWidth** of 360 degrees, then the **RadarAngle** doesn't matter. If you decrease the **ScanWidth**, then you can point this radar in a particular direction.

You can see all of these properties in action when you run your vehicle in a scenario. A translucent area projecting from your vehicle shows what area the radar can 'see'.

The **PlugIn** event is used to connect to a *Filter_IFF* component for discriminating what object you have detected. The **FilterPlug** property is used to plug into the **FilterSocket** of the *Filter_IFF* component. See the help for *Filter_IFF* for more information.

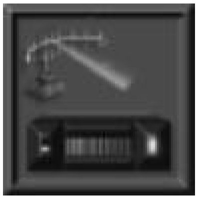
Usage

Use one or more *ProximityRadar* components to find other vehicles or objects in a scenario. You can wire these radars to movement or steering controls to head toward or away from a desired object.

See the *Filter_IFF* component to discriminate between friend and foe and other objects.

RadarDetector

Detects if your vehicle is in another radar.



Category:

Physical

Properties:

State

Events:

Change, TurnOff, TurnOn

Grp	*Defense*
Wgt	*10*
Pts	*10*

Description

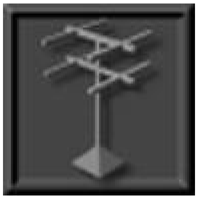
The *RadarDetector* is used to detect when your rover is in the radar field of another rover. The *RadarDetector* will trigger **TurnOff** or **TurnOn** depending upon the situation and **Change** will de triggered in either event.

Usage

Place a *RadarDetector* anywhere on your chassis. Each time you enter a radar field it will trigger a **TurnOn** event and when you leave this field it triggers **TurnOff**. This can easily be implemented with a *ModeSwitch* to have independent "search" and "evade" routines.

RadioReceiver

A component used to receive signals from a RadioTransmitter.



Category:

Physical

Properties:

IncomingNumber, Station

Events:

NumberReceived

Description

Grp	*Nav/Comm*
Wgt	*10*
Pts	*30*

The *RadioReceiver* is used to receive signals from a *RadioTransmitter*, which is mounted on a teammate's vehicle. You can't send signals to vehicles on the other team.

The signals sent are in the form of numbers. The *RadioReceiver* fires the event **NumberReceived** every time a message is received. Read the message by examining the **IncomingNumber** property.

Set the **Station** property to the same number on both the *RadioReceiver* and *RadioTransmitter* for the two vehicles to communicate. There are 50 radio stations that can be used to communicate between vehicles.

Usage

You will need both a *RadioReceiver* and a *RadioTransmitter* for either to be useful. Make sure that they are both initialized with the same station.

To pass messages, set the **SendNumber** property on the *RadioTransmitter* to the desired value. Wire up a component to the *RadioReceiver* on the **NumberReceived** event. You may also use the wire to copy the message to another component.

Bear in mind, you need not use the message itself. More often than not, simply knowing that a message has been received is enough to accomplish a given task.

See the *XYFinder* for a more detailed example using the *RadioReceiver*.

RadioTransmitter

A component used to send signals to a RadioReceiver.

Grp	*Nav/Comm*
Wgt	*10*
Pts	*30*

Category:

Physical

Properties:

SendNumber, Station

Events:

None

Description

Use the *RadioTransmitter* to pass messages to the *RadioReceiver*. Set the **Station** property to the same number on both the transmitter and receiver. You can only send messages to a vehicle on the same team.

Then set the **SendNumber** property to a number and the signal will be sent. If you only need to cause an event on the listening end of the radio, simply set **SendNumber** to 0 and ignore the value of the **IncomingNumber** property on the receiving end.

Usage

You will need both a *RadioReceiver* and a *RadioTransmitter* for either to be useful. Make sure that they are both initialized with the same station. Test your vehicles in ***Testing Ground 2*** where there is a teammate to work with.

To pass messages, set the **SendNumber** property on the transmitter to the desired value. Wire up a component to the receiver on the **NumberReceived** event. You may also use the wire to copy the message to another component.

Bear in mind, you need not use the message itself. More often than not, simply knowing that a message has been received is enough to accomplish a given task.

See the *XYFinder* component for a detailed example using the *RadioTransmitter*.

Randomizer

A component that generates a random number within a specified range.

Category:

Logical

Properties:

Maximum, Minimum, Output, Trigger

Events:

Set

Description

Grp	*Math*
Wgt	*0*
Pts	*0*

Any time you want your vehicle to behave unpredictably, use a *Randomizer* to the circuitry.

The **Maximum** and **Minimum** properties are used to establish the range of the output. For choosing random angles, the range can be -180 to +180. To choose a random sound on the *Speaker*, the range would be from 1 to 5. *Randomizer* only returns integer values.

Set the **Trigger** property (an activate property) to cause a new random number to be generated. The **Output** property holds the new number.

The **Set** event is triggered whenever the Randomizer has generated a new random number.

Note: at the end of a scenario, if you hit ***Instant Replay***, the *Randomizer* will generate the same sequence of values. If you hit ***Play Again***, new random values will be generated.

Usage

Send off random colored fireworks every time your vehicle bumps into something.

Using a hovercraft vehicle, add some *Thruster*s, a *BumpSensor*, a *Randomizer) and a Fireworks* component.

Click on the *Thruster*s and set them to full speed ahead. This way your hovercraft will speed around the room bumping into everything.

Click on the *Randomizer* and set the **Minimum** property to 0 and the **Maximum** property to 6. These corresponde to the 7 colors available in the *Fireworks*.

Create a wire from the *BumpSensor* to the *Randomizer* which reads: when the *BumpSensor* triggers **Bump**, activate the **Trigger** property of the *Randomizer*.

Next create a wire from the *Randomizer* to the *Fireworks* which reads: when the *Randomizer* triggers **Set**, set the *Fireworks*' **FireColor** to the *Randomizer's* property, **Output**.

When the vehicle bumps, a new random number is generated and sent to the *Firworks* component to set off the fireworks.

RangeTest

A component used to detect whether a value is above, below or within a certain range of values.

Category:

Logical

Properties:

Input, Maximum, Minimum, Output

Events:

AboveRange, BelowRange, Change, InRange, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

Description

Use the *RangeTest* component to check if a value is within a given range, or force it to be within that range.

This component will generate these events: **AboveRange**, **BelowRange**, and **InRange**. The **Change** event will be triggered whenever the **Output** changes. The **Set** event is triggered whenever an input is modified.

Set the **Minimum** and **Maximum** properties of this component to define the range of interest. Then set the **Input** value which is to be tested.

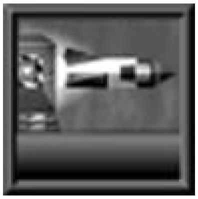
The **Output** property is the same as the input when it is within the range. If it is greater than the **Maximum**, then **Output** will be the maximum. If it is less than the **Minimum**, then **Output** will be the minimum.

Usage

See the *Speedometer* component for an example that uses the *RangeTest* component.

RocketLauncher

A component that fires a slow but deadly rocket -- watch out for the splash damage!



Category:

Weapon

Properties:

Fire, RocketSpeed

Events:

None

Description

Grp ... Weapons
Wgt 150
Pts 20

The *RocketLauncher* sends out a rocket that travels straight and explodes on impact. It takes three seconds to reload before you can fire again, so use it carefully!

Set the activate property, **Fire**, to launch a rocket.

You can control the speed at which the rockets travel -- either 1, 2, or 3 meters per second. The damage done by a rocket depends on its speed -- a rocket traveling at 1 m/sec (the default) does 50 hit points of damage. 2 m/sec does 25 damage, and 3 m/sec does 15 damage.

Usage

Use a *MediumRadar* to spot your opponent and fire the *RocketLauncher*.

From the ***Component*** selection screen, drag a *MediumRadar* onto the front of your vehicle and a

RocketLauncher to a central location on your vehicle.

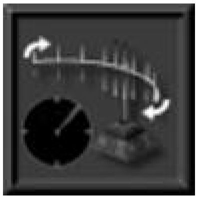
In the ***Wiring*** screen, create a wire from the *MediumRadar* to the *RocketLauncher*. This wire should read: when the *MediumRadar* triggers **TurnsOn**, activate the *RocketLauncher*'s **Fire** property.

Set the cone of the *MediumRadar* by clicking on its icon and setting default properties. The **ScanWidth** should be fairly narrow, 10-15 degrees for instance, to get a good shot.

Refire rate: 3 seconds
Damage: 15, 25, or 50 hit points for direct hit
Range: infinite
Speed: 1, 2 or 3 meters/second
Pivots: none

RotatingRadar

A radar that can detect objects within 5 meters that will automatically rotate at a fixed rate.



Category:

Physical

Properties:

Angle, FilterPlug, MaxRange, Rate, ScanWidth, State

Events:

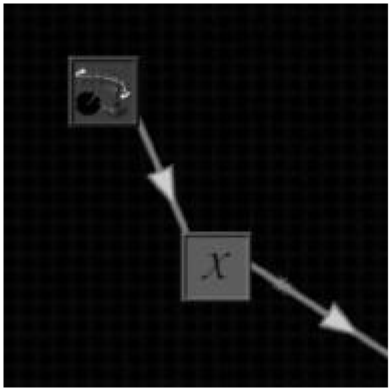
Change, PlugIn, TurnOff, TurnOn

Description

Grp	*Sensors*
Wgt	*45*
Pts	*30*

The *RotatingRadar* is similar to the *MediumRadar* with an added 'rotating' feature. (See help on *MediumRadar*.)

Set the **Rate** propery of the *RotatingRadar* to the rate at which you want the radar to spin. For example if you want the *RotatingRadar* to spin counter-clockwise at 45 degrees per second, set the **Rate** to 45. To stop the *RotatingRadar* from spinning, set the **Rate** to 0. clockwise rotation is set using negative numbers.



Usage

Use a *RotatingRadar* component to record the last known relative direction of your opponent.

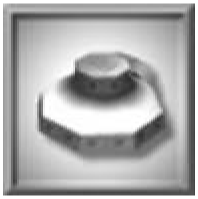
Create a wire from the *RotatingRadar* to a *Variable* component. This wire should read, when the *RotatingRadar* triggers **TurnOn**, set the *Variable*'s **Input** property to the radar's current **Angle**.

Click on the *RotatingRadar* and set its **ScanWidth** to 15 degrees. Whenever the *RotatingRadar* sweeps past the opponent, it will record the relative angle in the *Variable* component. Use this value to guide your vehicle to your opponent.

Note: You can build your own rotating radar using one of the other radars and some logical components to save weight.

RunningLight

A component that helps you find your vehicle in a crowd or helps you debug your vehicle's behavior.



Category:

Physical

Properties:

Mode

Events:

None

Description

Grp	*Extras*
Wgt	*0*
Pts	*0*

The *RunningLight* can be set to one of 7 colors which correspond to the colors used in the *ModeSwitch* component. Set its color **Mode** and it will change to the appropriate color.

Usage

When using the *ModeSwitch* component, you can add this *RunningLight* to your vehicle, connect them together and see what mode you are in by the color it displays.

Another example is to set the color of the *RunningLight* based on whether your left or right TrackSensor sees the edge of the track.

For this example, add a *RunningLight*, and two *TrackSensor*s to your vehicle. Create a wire from the left *TrackSensor* to the *RunningLight*. This wire should read: when the left *TrackSensor* triggers **TurnsOff**, set the *RunningLight*'s **Mode** property to red (choose the red colored block under the property box.

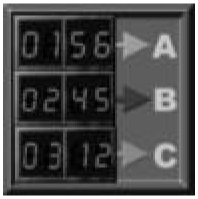
Add a wire from the right *TrackSensor* to the *RunningLight*. This one should read: when the right *TrackSensor* triggers **TurnsOff**, set the *RunningLight*'s **Mode** to blue.

Now when you run this vehicle in a scenario with a track (***Tutorial: TrackSensor***s, for instance), when the left *TrackSensor* turns off or just sees the edge of the track, the *RunningLight* will turn red.

Don't forget that you need to add the appropriate engine or thruster components to get your vehicle to move!

Sequencer

Triggers a set of events in a well-defined order



Category:

Logical

Properties:

TickTime

Events:

Set0, Set1, Set2, Set3, Set4, Set5, Set6, Set7, Set8

Grp ....Logic
Wgt 0
Pts 0

Description

The *Sequencer* component simply triggers its events in the following order:

Set0 Set1 Set2 Set3 Set4 Set5 Set6 Set7 Set8

If you set **TickTime** to a nonzero time, you get this behavior:

Immediately trigger **Set0**
Wait **TickTime** seconds
Trigger **Set1**
Wait **TickTime** more seconds
Trigger **Set2**
...
Wait **TickTime** more seconds
Trigger **Set8**

Usage

This can be used to implement a simple state machine, such as "Set Blue mode, back up and turn for two seconds, turn the other way and go forward for two more seconds, then revert to previous mode."

But if you set **TickTime** to 0, you eliminate all the waits from the above: *Sequencer* simply triggers all its events immediately, in the stated order. This can be used when it's important that some wire must fire before some other wire.

Note that you can use *Sequencer* to build a state machine even when you want different time delays between steps. For example, if you want to do something for one second, then something else for three seconds, then a final thing for one more second, you can connect wires to **Set0**, **Set1**, **Set4** and **Set5**, and set **TickTime** to 1. *Sequencer* will trigger **Set2** one second after **Set1**, and **Set3** one second after **Set2**, but you can ignore that.

You can cascade *Sequencer* components to build a sequence of more than eight steps (or involving more complex patterns of time delays). Simply connect the first *Sequencer*'s **Set8** event to copy **TickTime** to the second *Sequencer*. Of course, you can connect any number of *Sequencer*s this way.

You can also construct a looping sequence, in which a *Sequencer* triggers some set of its events over and over again. Suppose you want to step through **Set0**, **Set1**, **Set2**, **Set3**, and then back to **Set0** again, at two-second intervals. Just connect **Set4** to a *Broadcast*, and connect the *Broadcast* back to set **TickTime** to 2 again. (Two seconds after **Set3** triggers, **Set4** will trigger, which will immediately trigger **Set0**, and so forth.)

You can use a *Startup* component to start a loop like this when the vehicle first wakes up.

Naturally, you can construct a looping sequence using cascaded *Sequencer*s as well. If you're using more than one *Sequencer* component in your loop, you don't even need a *Broadcast* component -- you can just make any one of the last *Sequencer*'s events copy its **TickTime** back to the first *Sequencer*!

SmallEngine

An engine that can be used with a wheeled or treaded chassis

Category:

Engine

Properties:

Throttle

Events:

None

Grp .. Movement
Wgt 100
Pts............ 30

Description

A *SmallEngine* component is used with wheeled or treaded vehicles for movement.

All three engines, small, medium and large, have a **Throttle** property which determines the engine speed. The range is -100% to +100%. A negative value will reverse the direction of your vehicle.

The *SmallEngine* is recommended for the small chassis. This is the lightest engine.

Usage

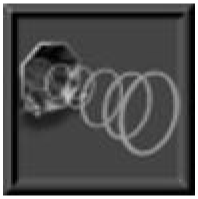
For a wheeled vehicle, just add an engine and set its **Throttle**.

For a treaded vehicle, add an *Engine* as well as a *TreadControl*. The *Engine* and *TreadControl* work together to move the tank, but they don't need a wire between them.

Set the engine to 100%, then use the *TreadControl* to determine direction and speed (see *TreadControl*) of your vehicle.

Sonar

A component that will give you the distance to the nearest object in front of it. You can use a Filter_IFF to filter out unwanted objects.



Category:

Physical

Properties:

Angle, Distance, FilterPlug, Fire, MaxRange

Events:

NoPing, Ping, PlugIn

Grp	*Sensors*
Wgt	*30*
Pts	*30*

Description

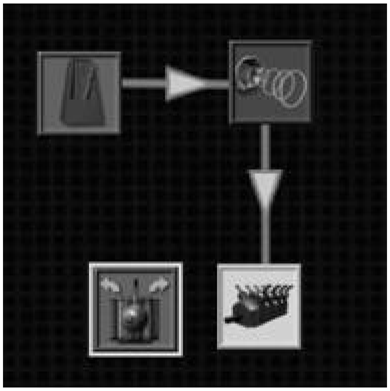
The *Sonar* component fires a blip straight ahead of it. If it receives a return echo, it will trigger a **ping** event.

The **Angle** property refers to the direction this component is facing, and the **MaxRange** is how far out the blip will fire. Set these values and then set the activate property, **Fire**, once or on a regular basis with a LoopTimer.

The output property, **Distance**, is the distance to the nearest object.

Sonar will trigger a **Ping** event if it sees something directly in front of it. It will trigger **NoPing** if it doesn't see anything within its **MaxRange**.

The **PlugIn** event is used to connect to a *Filter_IFF* component for discriminating what object you have pinged. The **FilterPlug** property is used to plug into the **FilterSocket** of the *Filter_IFF* component. See the help for *Filter_IFF* for more information.



Usage

As an example, use *Sonar* to stay a desired distance away from the wall. Use a TreadL (large treaded vehicle), *LargeEngine*, *TreadControl*, *Sonar*, and *LoopTimer* components.

Create a wire from the *LoopTimer* to the *Sonar*: when the *LoopTimer* triggers **tick**, activate the **Fire** property of the *Sonar*. Click on the *LoopTimer* and set the **Ticktime** to about 0.2 seconds. Every 0.2 seconds, the *Sonar* will check to see if anything is in front of it.

Click on the engine component and set its **Throttle** to 100%, set the **LeftTread** and **RightTread** of the *TreadControl* to 100% as well. Then add a wire from the *Sonar* to the engine to tell it to stop the engine when the *Sonar* sees something. This wire should read: when *Sonar* triggers **Ping**, set the engine's **Throttle** to 0.

Click on the *Sonar* component to set its **MaxRange** to 2 meters. When you hit ***GO*** this should result in a tank that moves forward until it is within 2 meters of an object and then it will stop.

Speaker

A component that plays a number of different sounds. Use for taunting an opponent or for debugging your vehicle.

Category:
Physical

Properties:
PlaySound, SoundGroup

Events:
None

Description

Grp	*Extras*
Wgt	*0*
Pts	*0*

The *Speaker* has 15 different sounds it can play. It doesn't generate any events.

Set the **SoundGroup** property to define the set of sounds to play. There are three different sets, each with five sounds.

Set the **PlaySound** property to play one of the 5 sounds within a group.

The different sound groups are:

SoundGroup 1 - The "Effects" set:
(1) Bonk
(2) Horn
(3) Slide Up
(4) Slide Down
(5) Break Glass

SoundGroup 2 - The "Numbers" set:
(1) "One!"
(2) "Two!"
(3) "Three!"
(4) "Four!"
(5) "Five!"

SoundGroup 3 - The "Taunt" set:
(1) "Whatever"
(2) "Nah nah"
(3) Laughter
(4) "Ouch"
(5) "Yahoo!"

Usage

Play a noise whenever your vehicle bumps into something.

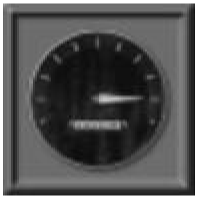
After choosing a chassis, go into the ***Component*** selection screen and drag a *Speaker* and *BumpSensor* onto your vehicle. Also add appropriate movement components.

Create a wire from the *BumpSensor* to the *Speaker*. This wire should read: when the *BumpSensor* triggers **Bump**, set the *Speaker*'s **PlaySound** property to a value of 1. This wire will cause the *Speaker* to play sound number 1 when the vehicle bumps into something.

Click on the *Speaker* icon to change the **SoundGroup** as desired.

Speedometer

A component that tells you how fast you are moving



Category:

Physical

Properties:

CurrentSpeed, Fuzziness

Events:

Change

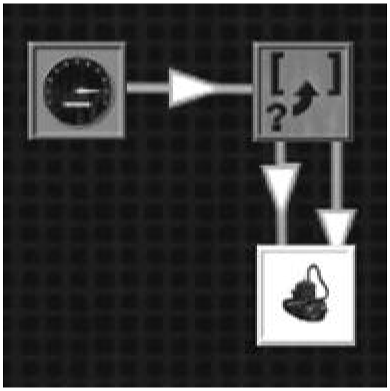
Description

Grp	*Sensors*
Wgt	*30*
Pts	*30*

The *Speedometer* will give you the forward speed, **CurrentSpeed**, of your vehicle. It will trigger **Change** whenever the vehicle's speed changes by the amount specified in the **Fuzziness** property.

The *Speedometer* is sensitive to the direction it is mounted on your vehicle. If you mount it in the default direction, straight ahead, then the **CurrentSpeed** property gives you forward (positive values) and backward (negative values) speed.

If you mount it at a 90 degree angle, it will give you left and right speed. If you mount it 180 degrees, then forward speeds, relative to your vehicle, will be negative.



Usage

Regulate the speed of your wheeled vehicle. Start with a medium wheeled chassis, WheelM. Go to the Component selection screen and add a *MediumEngine*, a *Speedometer*, and a *RangeTest* component.

Create a wire from the *Speedometer* to the *RangeTest*. This wire should read: when the *Speedometer* triggers **Change**, set the *RangeTest* **Input** property to the **CurrentSpeed** property of the *Speedometer*.

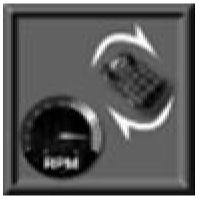
Now click on the *RangeTest* icon and set the desired range that you would like your vehicle's speed to stay in. Set the **Minimum** and **Maximum** properties to 1.8 and 2.1.

Create a wire from the *RangeTest* component to the *MediumEngine* which reads: when the *RangeTest* triggers **AboveRange**, set the *MediumEngine*'s **Throttle** to 0 (turn off the engine). Then create a second wire between these two components which reads: when the *RangeTest* triggers **BelowRange**, set the *MediumEngine*'s **Throttle** to 80 (turn on the engine).

This will turn on and off the throttle to regulate the speed to remain within the range you specified. It turns on and off so quickly that it results in a smooth motion for your vehicle.

SpinOMeter

A component that tells you how fast you are spinning



Category:
Physical

Properties:
CurrentSpeed, Fuzziness

Events:
Change

Grp	*Sensors*
Wgt	*30*
Pts	*30*

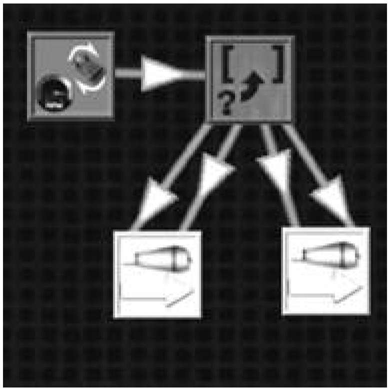
Description

The *SpinOmeter* is used to determine how fast your vehicle is spinning. This is especially good for hovercrafts, and can be used to help counter too much spin.

The **Change** event is triggered when the rotational speed changes by more than the **Fuzziness** property.

The output property of this component is the **CurrentSpeed**.

Note: Turning clockwise is negative rotational speed and turning counter-clockwise is a positive rotational speed.



Usage

Use the *SpinOmeter* to counteract any spin that your hovercraft picks up from bouncing off walls or other objects.

Start with a hovercraft chassis and add a *SpinOmeter*, two *Thrusters*, and a *RangeTest* component.

Click on the *RangeTest* component and set the desired **Maximum** and **Minimum** rotational speeds. For this example, set the minimum to -5 and the maximum to +5.

Create a wire from the *SpinOmeter* to the *RangeTest*. It should read: when the *SpinOmeter* triggers **Change**, set the *RangeTest*'s **Input** property to the *SpinOmeter*'s **CurrentSpeed** property. With this wire, your *RangeTest* component is testing the rotational speed of your vehicle.

If the hovercraft is spinning at +5 degrees/second, it is going in a counter-clockwise direction. So we want to counter-act that with a clockwise thrust, which means turn the left thruster on forward and the right thruster reverse.

To do this create a wire from the *RangeTest* to the LEFT *Thruster* which reads: when the *RangeTest* triggers **AboveRange**, set the left *Thruster*'s **Thrust** to +50. Create a wire from the *RangeTest* to the RIGHT *Thruster* to read: when the *RangeTest* triggers **AboveRange**, set the right *Thruster*'s **Thrust** to -50.

Similarly, when the hovercraft is spinning -5 degrees/sec, we need to increase the right thruster and reverse the left thruster.

Create another wire from the *RangeTest* to the LEFT *Thruster* which says: when the *RangeTest* triggers **BelowRange**, set the left *Thruster*'s **Thrust** to -50. A second a wire from the *RangeTest* to the RIGHT *Thruster* reads: when the *RangeTest* triggers **BelowRange**, set the right *Thruster*'s **Thrust** to +50.

SpinThruster

A thruster that is rotatable, a bit heavier and more costly than the simple Thruster.

Category:

Physical

Properties:

Angle, Thrust

Events:

None

Description

Grp . Movement
Wgt 120
Pts 30

The *SpinThruster* acts very much like a standard *Thruster* with the added property of **Angle**, which lets you set and change the angle of thrust during a competition.

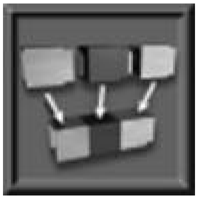
The **Thrust** property is a percentage of total power which ranges from -100% to 100%.

Usage

Wire a *SpinThruster* directly to a component that has an output **Bearing** property, such as a *WaypointSensor*.

Splice

Combines text and/or numbers into a single message



Category:

Logical

Properties:

Calc, Message, Part1, Part2, Part3, Part4

Events:

Change, Set

Grp. Debugging
Wgt 0
Pts............... 0

Description

The *Splice* component can be used to assemble multiple values into a single message. For example, if you set **Part1** to "The value is " and **Part3** to "!", and then use a wire to copy the **Output** of a *Variable* to **Part2**, when the *Variable* happens to contain the value 17, *Splice* will produce the **Message** "The value is 17!"

Splice triggers the **Set** event when it is set, and the **Change** event whenever its value changes.

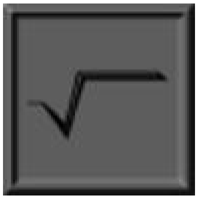
Note that *Splice* uses the **Calc** mechanism: if **Calc** is left 'true', each time you set any of **Part1**, **Part2**, **Part3** or **Part4** you might get **Set** or **Change** events. If you set **Calc** to 'false', then *Splice* only recomputes **Message** (and possibly triggers **Set** or **Change**) when you set **Calc**.

Usage

See the *DebugMessage* component for an example of *Splice*.

SquareRoot

A component to compute the square root of an input.



Category:

Logical

Properties:

Input, Output

Events:

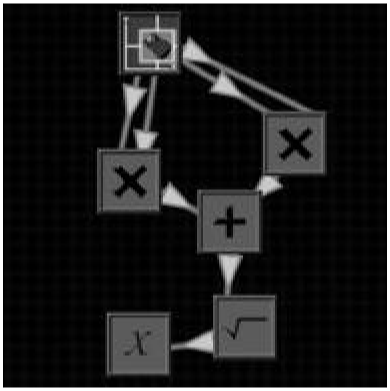
Change, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

Description

This component gives you the square root of **Input** and stores the result in **Output**.

The *SquareRoot* component will generate a **Set** event whenever **Input** is set. It will trigger a **Change** event when **Output** changes.



Usage

Usage: To find the distance from your vehicle to the origin in world coordinates, use the *SquareRoot* component along with two *Multiply* components, an *Add*, and an *XYSensor*.

Create a wire from the *XYSensor* to the *Multiply*, that reads, when the *XYSensor* triggers **Change**, set the *Multiply*'s **Input1** to the *XYSensor*'s **Xposition**. Add a second wire from the *XYSensor* to the *Multiply* that reads when the *XYSensor* triggers **Change**, set the *Multiply*'s **Input2** to the *XYSensor*'s **Xposition**. Now the output will be the Xposition squared.

With the second *Multiply*, create two more wires from the *XYSensor* to square the YPosition.

Next create a wire from each of the *Multiply* components to the two inputs of the *Add* component to add these numbers.

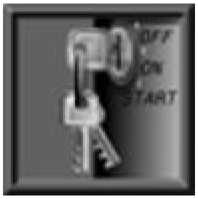
Finally, create a wire from the *Add* to the *SquareRoot* that reads, when the *Add* triggers **Change**, set the *SquareRoot*'s **Input** to the **Output** of the *Add*.

The **Output** of the *SquareRoot* will be the distance of your vehicle from the origin.

You can wire this to the *DebugMessage* component to print out the value on the competition screen.

Startup

A component that gives a single tick at the beginning of the scenario.



Category:

Logical

Properties:

None

Events:

Tick

Grp	*Timing*
Wgt	*0*
Pts	*0*

Description

The *Startup* component is used to kick things off in a given vehicle upon starting the scenario or when the vehicle respawns after a death. It triggers the following Events:

- **Tick** is triggered once each time the vehicle wakes up, at the start of the scenario and also every time the vehicle is resurrected after having been killed.

- **Start** is triggered just once at the start of the scenario. It is not triggered again, even if the vehicle is killed and subsequently resurrected.

- **Respawn** is not triggered initially -- it is triggered only when the vehicle is resurrected after having been killed. If the vehicle is killed and resurrected more than once, it is triggered once each time the vehicle is resurrected.

Usage

For many components there are two ways you can get them going at the beginning of a scenario. You can give them initial values, or you can wire them up to the *StartUp* component.

The *Speaker* is one exception. You cannot set an initial value for a *Speaker* property the *StartUp* component.

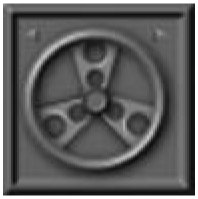
Add a *Speaker* to your vehicle. Go to the Wiring screen and add a *StartUp* component.

Create a wire from the *StartUp* component to the *Speaker*. This wire should read: When the *StartUp* triggers **Tick**, set the *Speaker*'s **PlaySound** property to 3 (or any number that you would like).

When the scenario starts, you will hear this sound. See the *KarmaVariable* for an example on using the *StartUp* component during respawning.

Steering

A component that allows you to steer a wheeled vehicle.



Category:

Physical

Properties:

Angle

Events:

None

Grp . Movement	
Wgt	*30*
Pts	*10*

Description

The *Steering* component is used with a wheeled chassis to change direction. It can turn as much as 30 degrees in either direction.

Set the **SteeringAngle** property to the desired steering angle.

Usage

Use *Steering* whenever you start with a wheeled vehicle: WheelS, WheelM, WheelL. It won't do anything on a hover or treaded vehicle.

As an example, when your radar spots a vehicle on the left, steer towards it.

Start with a wheeled chassis (WheelS, for instance). Go into the ***Component*** selection screen and drag a *MediumEngine*, a *Steering* component, and a *MediumRadar* onto your vehicle.

Click on the *MediumRadar* and set its **RadarAngle** to face to the left. Then add a wire from the *MediumRadar* to the *Steering* which reads: when the *MediumRadar* triggers **TurnsOn**, set the *Steering* **Angle** to a value of 15, which is to the left.

Now when the radar spots something, your vehicle will steer towards it.

Subtract

A component that subtracts input2 from input1 and generates an output equal to the difference.

Grp	*Math*
Wgt	*0*
Pts	*0*

Category:

Logical

Properties:

Input1, Input2, Output

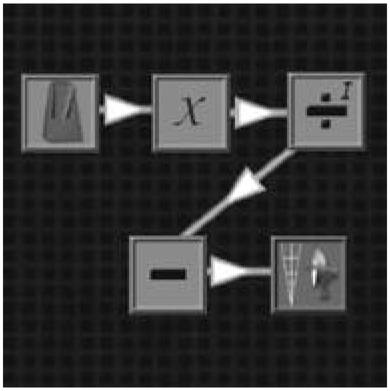
Events:

Change, Set

Description

This component subtracts **Input2** from **Input1**, and gives you the difference as **Output**.

The *Subtract* component will generate a **Set** event whenever **Input1** or **Input2** changes. It will trigger a **Change** event when the output changes.



Usage

Sweep a *LongRangeRadar* from -45 to +45 degrees. For this example we create a modulo or 'clock' arithmetic circuit. The basic idea is that we increment a counter, divide by 90 and use only the remainder. The remainder for the first 90 counts is 0-89. The remainder for 90-179 is also 0-89. So you can continue to increment the counter and you will keep getting the values 0-89 as the remainder of the divide.

In this example, we then subtract 45 from the remainder so we get the values -45 to 45 over and over again. This output is the **Angle** input for the *LongRangeRadar*.

Drag a *LongRangeRadar* component onto your vehicle from the ***Components*** selection screen. Go into the ***Wiring*** screen and add a *LoopTimer*, a *Variable*, a *Divide*, and a *Subtract*.

The *LoopTimer* sets up the step speed for sweeping the *Sonar*. Click on it and set the **TickTime** to 0.2 seconds.

Create a wire from the *LoopTimer* to the *Variable*. This wire should read: when the *LoopTimer* triggers **Tick**, set the *Variable*'s **IncrementBy** property to 5 degrees. So every 0.2 seconds, the variable increments by 5 degrees.

Next create a wire from the *Variable* to the *Divide* which reads: when the *Variable* triggers **Change**, set the *Divide*'s **Input1** to the *Variable*'s **Output**.

Click on the *Divide* icon and set the **Input2** to a constant value of 90 degrees. So you always divide **Input1** by 90.

Now create a wire from the *Divide* to the *Subtract* which reads: when the *Divide* triggers **Change**, set the *Subtract*'s **Input1** property to the *Divide*'s **Remainder**.

Click on the *Subtract* icon and set the **Input2** property to 45. Always subtract 45.

The last wire goes from the *Subtract* component to the *LongRangeRadar* and reads: when the *Subtract* triggers **Change**, set the *LongRangeRadar*'s **Angle** to the *Subtract*'s **Output**.

Try it out!

Switch

A component that reproduces events if it is enabled.

Category:

Logical

Properties:

Enabled, Toggle, Trigger

Events:

Change, Set, TurnOff, TurnOn

Grp	*Logic*
Wgt	*0*
Pts	*0*

Description

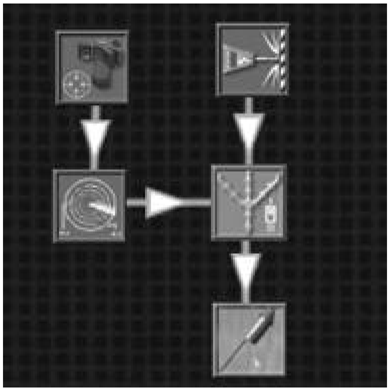
The *Switch* works like a Broadcast component in that it can rebroadcast a trigger to many other components. The difference is that *Switch* has an **Enabled** property. If **Enabled** is true (1), then setting the **Trigger** property will result in triggering the **Set** event.

If **Enabled** is false (0), then the **Set** event will not trigger.

The **Change** event triggers when **Enabled** changes from true to false or false to true.

The **TurnOn** and **TurnOff** events fire when **Enabled** is turned on and turned off, respectively.

You can **Toggle** the *Switch*, which makes **Enabled** invert its state. If **Enabled** is true, when you set **Toggle**, it will become false. If **Enabled** is false, when you set **Toggle**, it will become true.



Usage

Use a filtered *ProximityRadar* to determine if you are near another vehicle when you detect a bump. If so, shoot off the fireworks. If you are not near another vehicle, don't shoot off the fireworks.

Add *ProximityRadar*, *Filter_IFF*, *BumpSensor*, and *Fireworks* to your vehicle from the ***Component*** selection screen.

Go into the ***Wiring*** screen and add a *Switch* .

Create a wire from the *ProximityRadar* to the *Filter_IFF* to allow only enemy vehicle information to come from the filter. (See the *Filter_IFF* help for more details).

Create a wire from the *Filter_IFF* to the {Switch}, which reads: when the *Filter_IFF* triggers **TurnOn**, set the *Switch*'s **Enabled** property to true (1). Add a second wire between the same two components that reads: when the *Filter_IFF* triggers **TurnOff**, set the *Switch*'s **Enabled** property to false (0).

Next create a wire from the *BumpSensor* to the *Switch* which reads: when the *BumpSensor* triggers **Bump**, activate the *Switch*'s **Trigger** property.

Finally, add a wire from the *Switch* to the *Fireworks* that reads: when the *Switch* triggers **Set**, set the **FireColor** property to 4 (or any color you like).

Now the fireworks will go off only when there is a collision and the enemy vehicle is in close range.

TaxiLight

A Taxicab light bar for mounting on a vehicle. It plays a sound when triggered.

Category:

Physical

Properties:

Play

Events:

None

Description

Grp	*Extras*
Wgt	*0*
Pts	*0*

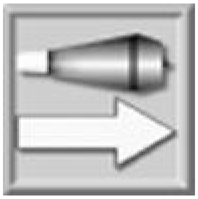
This component is basically decorative, but you can set the activate property, **Play** to hear a taxi whistle. It does not generate any events.

Usage

When you bump into something, play the taxi sound. Add a *BumpSensor* and a *TaxiLight* to your vehicle. Create a wire from the *BumpSensor* to the *TaxiLight*. This wire should read: when the *BumpSensor* triggers **Bump**, activate **Play** on the *TaxiLight*.

Thruster

A component that applies a force in the desired direction.



Category:

Physical

Properties:

Thrust

Events:

None

Grp	*Movement*
Wgt	*75*
Pts	*10*

Description

The *Thruster* is used most often with hovercraft vehicles for their primary form of motion. When you place a *Thruster* on a vehicle in its default direction, facing forward, it will propel the vehicle forward.

This thruster fires forward and backward only. When you place this *Thruster* on your vehicle you can rotate it in 90 degree increments. No rotation is allowed through wiring.

Thrust is a percentage of power that you set for the *Thruster* and it ranges from -100% to 100%.

Usage

Turn on and off *Thruster*s as desired to steer a hovercraft toward an object or waypoint.

As an example, get your hovercraft to turn in place. Start with a hover chassis and add two *Thruster*s to the left and right of the vehicle.

Go to the Wiring screen and click on one of the *Thrusters*. Set its **Thrust** to 100%. Click on the other one and set its **Thrust** to -100%.

Hit ***GO***, and watch it turn.

Timer

A one-shot settable timer which can be paused and restarted.

Category:

Logical

Properties:

PauseTimer, RestartTimer, TickTime

Events:

Tick

Grp.......Timing
Wgt 0
Pts.............. 0

Description

The *Timer* component will count down from a desired **TickTime** to zero and then trigger the **Tick** event.

The *Timer* will automatically start counting down as soon as you set the **TickTime** property. A **TickTime** of 0 will cause it to stop. Setting **PauseTimer** will also stop the timer from counting down. **RestartTimer** will start it again.

Usage

The *Timer* component can be used in conjunction with other components to start or stop an event in the future.

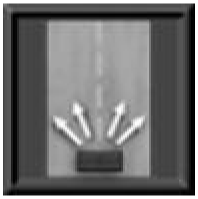
As an example, turn off a *Thruster* after 2 seconds. Add *Thruster*s and a *Timer* to your vehicle. The *Timer* component is found in the ***Wiring*** screen.

Connect a wire from the *Timer* to a *Thruster*. This wire should read: when the *Timer* triggers **Ticks**, set the *Thruster*'s **Thrust** to 0%.

Now click on the *Timer* to set its **TickTime** property to 2.

TrackSensor

A component that detects whether the sensor sees the track.



Category:

Physical

Properties:

Angle, Range, State

Events:

Change, TurnOff, TurnOn

Grp	*Sensors*
Wgt	*20*
Pts	*30*

Description

The *TrackSensor* is placed at a particular position on the vehicle, and is given a **Range** and an **Angle** from that point. It inspects the ground at that location to see if the track is there.

If it started on the track and it no longer sees the track, it triggers a **TurnOff** event. If it started off the track and now it sees the track, it triggers a **TurnOn** event. The **Change** event is triggered along with either of these conditions.

The **Range** of the TrackSensor defines the distance from its mounting point to where it will be inspecting the ground. This is measured in meters.

The **Angle** of the sensor defines its direction relative to the vehicle's direction.

The **State** is the current output state of the sensor, on or off.

Usage

Use the *TrackSensor* in conjunction with steering controls to keep a vehicle on track or away from a track object.

For example, to follow a track, use a *TrackSensor* and *Steering* component on a wheeled chassis.

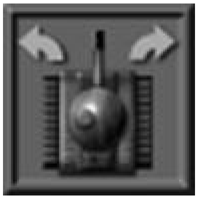
Place the *TrackSensor* on the front right of the vehicle. Click on the icon and set its **Angle** property to -15 (to the right), and its **Range** to 1 meter.

Create a wire from the *TrackSensor* to the *Steering* component. This wire should read: when the *TrackSensor* triggers **TurnsOff**, set the *Steering* component's **Angle** to 15 degrees (turn left).

Now when the vehicle sees the edge of the track on the right side, it steers left.

TreadControl

A component that controls power to the left and right treads of a treaded vehicle.



Category:
Physical

Properties:
LeftTread, RightTread

Events:
None

Grp .. Movement
Wgt 50
Pts............. 10

Description

The *TreadControl* component is used to steer a treaded vehicle. It is a transmission that controls the amount of engine power that goes to each of the treads.

Set the **LeftTread** property to add power to the left side of the tank, from -100 to 100%. Similarly set the **RightTread** property to add power to the right side of the tank.

Without an engine, or with an engine set to zero throttle, the *TreadControl* does nothing.

Usage

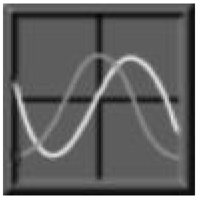
Use the *TreadControl* component in conjunction with one of the engines to move and steer a treaded vehicle.

Start with a Treaded chassis (TreadS, TreadM, or TreadL). Go to the ***Component*** selection screen and drag a *MediumEngine*, and a *TreadControl* component onto your chassis.

To move forward, click on the engine and set its **Throttle** to 100%. Then click on the *TreadControl* and set its **LeftTread** to 100%, and its **RightTread** to 100%.

Try it out in the scenario. To turn left, set the **RightTread** higher than the left.

Trigonometry



A component that calculates Sine, Cosine and Tangent.

Category:
Logical

Properties:
Cosine, Angle, Sine, Tangent

Events:
Change, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

Description

The *Trigonometry* component can be used to calculate the sine, cosine or the tangent of a given angle.

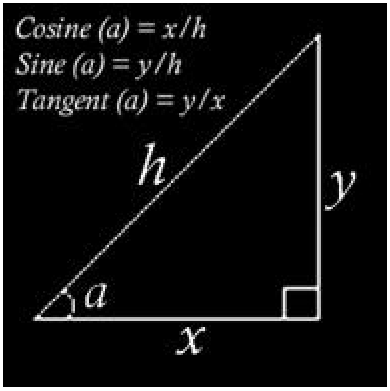
Angle (in degrees) is the only input and there are three possible outputs. Given the right triangle shown:

Cosine gives you the ratio of the length of the adjacent side, X, to the hypotenuse H. This value will always be between -1 and 1.

Sine gives you the ratio of the length of the opposite side, Y, to the hypotenuse H. This value will always be between -1 and 1.

Tangent gives you the ratio of the length of the opposite side Y to the adjacent side X.

There are two events. **Change** will trigger when the value of **Angle** changes. **Set** event is triggered when the **Angle** property is updated even if it hasn't changed.



Variable

A component to store a numeric variable. Can also be used as a counter.

Category:

Logical

Properties:

IncrementBy, Input, Output, Query

Events:

Change, Querying, Set

Grp	*Math*
Wgt	*0*
Pts	*0*

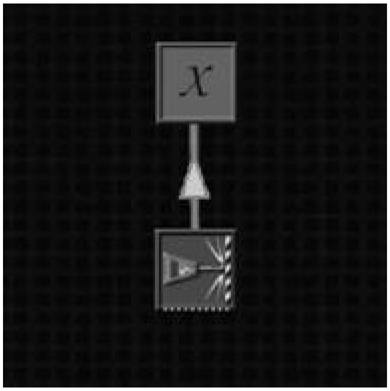
Description

The *Variable* component stores a number that can be used by other components. It can be set up as a counter by setting the **IncrementBy** property. It can also be used to propagate changes between two similar components. Examples for these uses are shown below.

The **Change** event is triggered whenever the *Variable*'s **Input** changes. **Set** event is triggered when the **Input** is updated even if it hasn't changed.

The **Query** property and **Querying** event are used together to force the variable to emit its number at a later time.

The **Output** property gives you the current value of the *Variable*.



Usage

The *Variable* can be used in a number of different ways. It can simply store a number which might be used as a constant in math operations. It can be set to increment automatically as a counter. It can store a variable, such as the current mode, and be able to reset the ModeSwitch to that mode at a later time. It can also be used to propagate a value between two components.

The *Variable* as a Counter

If you would like to count the number of times you have bumped into something, or the number of times your radar has **TurnedOn**, you can use the *Variable* as a counter.

Add a *BumpSensor* and *Variable* to your vehicle. Create a wire from the *BumpSensor* to the *Variable* which reads: when the *BumpSensor* triggers

Bump, set the *Variable*'s property **IncrementBy** to the value 1.

Variable as *ModeSwitch* Reverter
For some situations it may be helpful to be able to revert back to the previous mode. One way to do this is to wire the *ModeSwitch* to the *Variable*: when the *ModeSwitch* triggers **Leave**, set the *Variable*'s **Input** to the *ModeSwitch*'s **Mode**.

Just before the *ModeSwitch* changes to a new mode it will trigger it's **Leave** property and with this connection to the *Variable*, it will save its current mode.

Then create a wire from the *Variable* to the *ModeSwitch* which says: when the *Variable* triggers **Querying**, set the *ModeSwitch*'s **Mode** to the *Variable*'s **Output**.

Then create a wire from another component to the *Variable* which sets its **Query** property. When that event fires, the *ModeSwitch* will revert back to its previous mode.

Variable Used to Propagate Properties

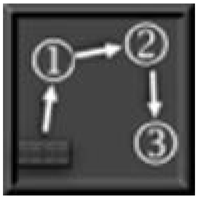
The Variable component can be used to propagate values between two similar components. This is important for components which don't create any events of their own, like *Thruster*s.

If you would like a number of components to get the same property values, you can use a *Variable* component to store the value and then send it out to as many components as you need.

For example, if you want two *Thruster*s to change to reverse upon bumping into something, then you could create a wire from the *BumpSensor* to the *Variable*. This wire would read: when the *BumpSensor* triggers **Bump**, set the *Variable*'s **Input** to a value of -100.

Then create two wires from the *Variable* to each of the *Thruster*s. These wires would both read: when the *Variable* triggers **Set**, set the *Thruster*'s **Thrust** to the *Variable*'s **Output**.

WaypointSensor



A sensor that gives you the distance and bearing to the next waypoint.

Category:

Physical

Properties:

Bearing, CurrentWaypoint, Distance, WaypointList

Events:

Change, HitWaypoint, NotFound

Grp.Nav/Comm	
Wgt	*10*
Pts............	*30*

Description

The *WaypointSensor* is used to navigate to preset waypoints in the scenario. If you click on a scenario (such as ***Capture the Flag*** in Battles), then press ***F1***, you will get a top down view of the room with the waypoints marked. You can also see them on the floor of the room when the scenario is running. There can be as many as 26 waypoints in a scenario, labeled with letters from A-Z. You can track these waypoints in any order by naming the waypoints in the **WaypointList**.

For example, if you want your vehicle to go from waypoint G to waypoint R, S, and T, returning to G between each, you might want to set the **WaypointList** to RGSGTG.

The *WaypointSensor* keeps track of your list of waypoints as numbers from 1 to N, where N is the length of the **WaypointList**. The

CurrentWaypoint resets to 1 when you change the **WaypointList**.

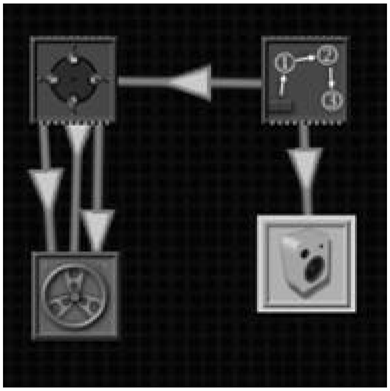
When the vehicle reaches the current waypoint, it fires a **HitWaypoint** event and increments to the next waypoint in the list.

The *WaypointSensor* triggers the **Change** event whenever your vehicle's bearing or distance to the waypoint has changed. The **Bearing** and **Distance** properties of the *WaypointSensor* give you the relative orientation and distance of the current waypoint from your current position. You can feed this **Bearing** information to the **RefBearing** of a *BearingSensor* to control the direction of your vehicle.

The *WaypointSensor* rotates to show the direction to the specified waypoint.

If there is no such waypoint, the *WaypointSensor* triggers the **NotFound** event so that your vehicle can detect the condition. Once the *WaypointSensor* has triggered **NotFound**, it will not trigger any other events until you reset it to look for a waypoint that does exist, by changing either **CurrentWaypoint** or **WaypointList**.

WaypointSensor



Usage

Use the *WaypointSensor* along with a *BearingSensor* to set the direction of your vehicle to the next Waypoint. You may want to go through the ***Tutorial: Waypoint Sensor***, chapter 3, for more details on using this component.

Start with a wheeled chassis and add an *Engine* and *Steering* component, as well as a *WaypointSensor* and *BearingSensor*. In this example we also have a *Speaker* component.

Add a wire from the *WaypointSensor* to the *BearingSensor*. The default wire selection will do what you want: when the *WaypointSensor* triggers **Change**, set the **RefBearing** of the *BearingSensor* to the *WaypointSensor*'s **Bearing**.

Click on the *WaypointSensor* to set the **WaypointList**.

Read the usage notes for the *BearingSensor* for more information on wiring the *BearingSensor* to your steering system.

In this example, the *BearingSensor* is connected to a *Steering* component with three wires. One wire reads: when the *BearingSensor* triggers **LeftOfRef**, set the *Steering* wheel's **Angle** to a value of -15 degrees (steer to the right).

The second wire reads: when the *BearingSensor* triggers **RightOfRef**, set the *Steering* wheel's **Angle** to a value of +15 degrees (steer to the left).

The third wire reads: when the *BearingSensor* triggers **OnRef**, set the *Steering* wheel's **Angle** to a value of 0 degrees (steer straight ahead).

The *WaypointSensor* has also been connected to a *Speaker*. When the *WaypointSensor* triggers **HitWaypoint** then it plays a sound.

Now make sure you give your engine a default throttle value so that your vehicle will move, and see how it does!

WeldingTorch

A weapon used to inflict close-range damage on your opponents.

Category:
Weapon

Properties:
Fire

Events:
None

Grp....Weapons
Wgt 100
Pts............ 10

Description

The *WeldingTorch* blows a hot flame that can be used to cook your opponents. It has a range of about one vehicle length. It stays on for about 2 seconds.

Set the **Fire** property to turn it on.

Usage

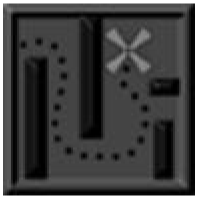
Use a *ProximityRadar* to determine when another vehicle is close enough to use the torch.

Connect a wire from the *ProximityRadar* to the *WeldingTorch*. This wire should read: when the *ProximityRadar* triggers **TurnsOn**, activate the *WeldingTorch*'s **Fire** property.

Refire rate: 2 seconds
Damage: 20 hit points/second
Range: 1 meter
Speed: instant hit
Pivots: none

XYFinder

A component that will report the bearing and distance to a specified XY position.



Category:

Physical

Properties:

Bearing, Distance, XPosition, YPosition

Events:

Change, HitWaypoint

Grp.Nav/Comm
Wgt 10
Pts............ 30

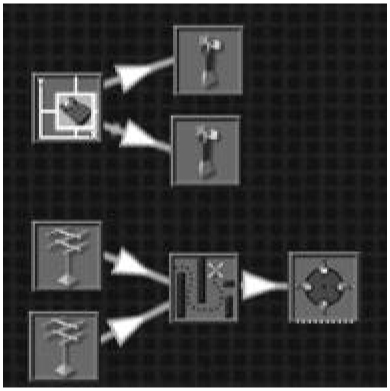
Description

The *XYFinder* will give you the distance and bearing to a given XY position.

Set the **XPosition** and **YPosition** properties through another component. Then read the output properties **Bearing** and **Distance**, to get its bearing relative to your vehicle and its distance from your vehicle.

The **Change** event will trigger every time the XPosition or YPosition changes.

XYFinder will trigger **HitWaypoint** when it reaches the specified (**XPosition**, **YPosition**). After that, you must change either or both of **XPosition** and **YPosition** before it will trigger **HitWaypoint** again.



Usage

In a team scenario or ***Testing Ground 2***, have one of your vehicles radio its X and Y position to you, then the second vehicle can follow the first.

Vehicle 1 has two *RadioTransmitters* and an *XYSensor*.

Create a wire from the *XYSensor* to one of the *RadioTransmitters* which reads: when the *XYSensor* triggers **Change**, set the *RadioTransmitter*'s **SendNumber** to the *XYSensor*'s **XPosition**.

Create a second wire from the *XYSensor* to the second *RadioTransmitter* which will send the **YPosition**.

Click on the *RadioTransmitters* icons to set the **Station** properties. Set the Xposition transmitter station to the same number as the Xposition receiver station on vehicle 2. Similarly, set the Yposition transmitter and the YPosition receiver to the same station numbers.

Vehicle 2 has two *RadioReceivers*, an *XYFinder* and a *BearingSensor*. In the wiring diagram all of these components are shown on one vehicle. In your testing, put the transmitters and receivers on two vehicles from the same team.

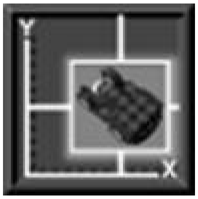
Create a wire from the *RadioReceiver* with the XPosition station number to the *XYFinder*, which reads: when the *RadioReceiver* triggers **NumberReceived**, set the *XYFinder*'s **XPosition** property to the *RadioReceiver*'s **IncomingNumber**.

Similarly, create a wire from the Yposition *RadioReceiver* to the *XYFinder* to set the **IncomingNumber** to the **YPosition**.

Finally, create a wire from the *XYFinder* to the *BearingSensor* which reads: when the *XYFinder* triggers **Change**, set the *BearingSensor*'s **RefBearing** to the *XYFinder*'s **Bearing**. Then use the trigger events of the *BearingSensor* to steer your vehicle.

XYSensor

A sensor used to determine your position in the world.



Category:

Physical

Properties:

XPosition, YPosition

Events:

Change

Description

Grp	*Nav/Comm*
Wgt	*10*
Pts	*30*

The *XYSensor* is used to get your position (X and Y) in the world. While this information is not very useful by itself it can be used in conjuction with other components.

The **Change** event will fire every time your vehicle's position changes and the **XPosition** and **YPosition** properties will be set accordingly.

Usage

Use an *XYSensor* to get your teammates to play "follow the leader"! Transmit the X and Y locations across a *RadioTransmitter* and use a *XYFinder* to get a **Bearing** and **Distance** from the location.

See the *XYFinder* for a detailed example.

Chapter 7

Frequently Asked Questions

Q. My vehicle doesn't do anything! What did I do wrong?

A. First, check that you have an engine (or thruster, if referring to a hover vehicle). Now make sure that the throttle is set to something other than zero. You probably want a value of 30 or more, but just to be sure, set it all the way to 100. If you've selected a treaded chassis, you will also need to have a TreadControl and to set both the left and right treads greater than 0. Your vehicle should now move.

Q. My rover is moving, but it isn't doing what the sensors are telling it to do. Why?

A. Are you using colored wires? If so, set them back to gray. Colored wires are only used if you have a mode switch and you have set the mode switch to that color. Next, check the sensor event that triggers a vehicle action. Many sensors default to the turn on event, but if you are using a logical gate, it would be better to trigger on the change event. Also keep in mind that a sensor that turns on will only generate a single event, even if it remains on for many seconds. Only after it has turned off, will it generate another turn on event.

Q. It seems like the radars don't always work. Sometimes they pass over the target without turning on.

A. Radars do not update on every frame. The fastest radar (long range) only updates every other frame. The slowest (short range) updates only every fifth frame. Try slowing down your scanning and the radar should work much better. MindRover runs at 20 frames per second.

Q. How do I set small values on the sliders?
A. Right-click to the left or right of the slider (not on the slider itself).

Q. I created a vehicle, but now I can’t find it! Where did it go?
A. Vehicles are automatically saved in the holobox at the top of the screen. The holobox has multiple lines of rovers, so you will need to scroll the holobox down to find yours. You can also use the dropdown box (just to the right of the holobox) to jump to the line. Your vehicle will be in the category that it was last saved in (vehicles are saved after you make a component or wiring change).
Also, if you created the vehicle in a different scenario than the one that you are trying to run, there’s a possibility that the current scenario might not allow your vehicle. Some scenarios don’t allow some components or certain chassis so any vehicle either with one of those components or made on that chassis will not show up in the holobox.
If you still cannot find your vehicle, run the Testing Ground tutorial. This scenario allows all components and chassis so your vehicle will show up. Now you can edit the vehicle and remove any illegal components so it can be used in other scenarios. The most common illegal component is the KeySensor. You can only use the KeySensor in the Testing Ground scenarios.

Q. How do I make a scenario run the same way every time?
A. The random number generator uses a "seed". You can set the seed manually. Instead of clicking on the Run button, enter the command “startscenario 99999” in the console window (you can replace the 99999 with any number of your choice).

Q.What is the advantage of using the IFF filter instead of the logical filter?
A. There isn’t an advantage. However, a contest may limit the components you can use, so it is possible that the logical filter is not allowed while the IFF filter is allowed. --for example, a contest

that only allows components available in the demo version of MindRover.

Q. I see other players creating rovers with puck and mice chassis. How do I do that?

A. To enable the extra chassis types, follow these steps *(these instructions assume that your MindRover is installed in the c:\games\MindRover directory, and that you log into MindRover as MyName)*.

1) Edit the file c:\games\MindRover\Users\MyName.usr, using an ordinary text editor (e.g. Notepad).

2) Near the bottom of the file you'll see two sets of lines. Each of the last few lines will begin with something like "scenX," where X is a number from 1 into the 20s.

3) Above the these lines will be a line containing only a decimal integer, something like 34. Edit that line and increment that number by one.

4) After the integer, add a new line under with the "chasn" command followed by the chassis to be added, like this:

```
chas1 vehicles/PuckChassis
```

That should give you the puck chassis. You will have to scroll the holobox down to the CogniToy bar in order to see it.

5) If you download the HydroBattles add-on, you can repeat the process for MouseChassisR, MouseChassisG, and MouseChassisB and just replace n with the next number instead of 1 in step 4.

Q. What about the UFO chassis?

A. You automatically get it when you beat all the installed scenarios (including any add-on scenarios you might have installed).

Q. Is there any limit to how fast my rover can go?

A. Yes, all rovers are limited to a maximum speed of 5 m/s. For most rovers, that speed is achievable given enough time, but of course a larger engine (or more thrusters) will bring your rover up to speed much more quickly!

Q. Will multiple engines work together?

A. No. Only the engine with the highest setting (factoring in the engine size) will have any effect. However, adding thrusters will provide a small increase in power (barely perceptible for a wheeled, negligible for a treaded). Conversely, putting an engine on a hover is truly just adding dead weight.

Q. How big are the grid squares where I place components?

A. Each square is 0.05 meters x 0.05 meters. Note that some squares are not adjacent to the remainder and may have differing spacing. If you know the basics of ICE and are interested in knowing exactly where each component is placed, you can do so by creating a test vehicle (place a 1x1 component in each space you want to examine) and look at the generated .ice file in your vehicles/ username directory. The .ice file will contain the exact location of each component in the `sys.placePhysicalComponent(…)` function call.

You can learn more about programming in ICE at: www.cognitoy.com/twiki