First-Class Sprites in Snap*!*

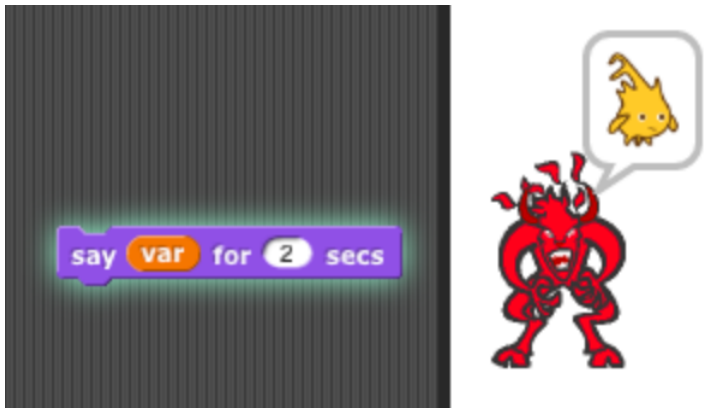
Jens Mönig
May 2 2016

Snap's standard microworld - derived from MIT Scratch - revolves around controlling cartoonish 2D actors called "sprites" living in a presentation area called "the stage". Sprites are largely autonomous objects acting on global events with hardly any links to other sprites except by name. Snap! version 4.0.7 promotes sprites to first-class members of the programming language. This opens up new frontiers for sprite (collision) detection and nesting.

The "Sensing" category of Snap's blocks palette features a new reporter revealing attributes and relationships, including sprites:

These can be shown in result-bubbles (see above), assigned to variables (and monitored in stage watchers),

passed as inputs into blocks (and also returned from them),



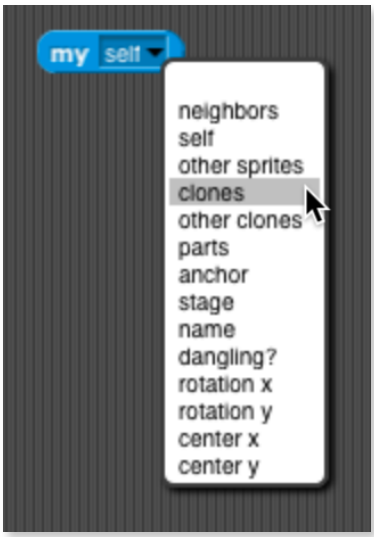
and organized in lists and tables. In other words, sprites are now first-class members of Snap*!*.

CONTENTS

In a Nutshell

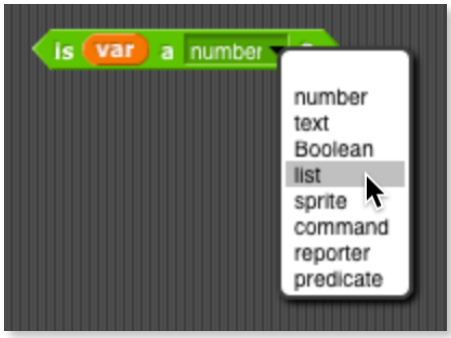
1) A new 

reporter returns attributes and relationships:



Depending on the selection in the drop-down menu the block reports either a sprite, a list of sprites, or other data types, e.g. a number representing “center x”, a text denoting the “name”, a Boolean for the “dangling?” switch.

2) A new type “sprite” is available in the type-testing predicate reporter



3) Sprite references can be used in every primitive input slot offering a drop-down menu for sprite names. You can also just keep using names.

4) For collision detection on a list of sprites instead of testing for each sprite individually you can also pass in a sprite list to find out whether ***any*** sprite in that list is touched.

5) the 

command can be used to set an attribute by passing its reified reporter into the first input slot. Since that slot rejects dropped reporters (in order to not confuse novices), passing the ring can be accomplished with CALL, RUN or LAUNCH:

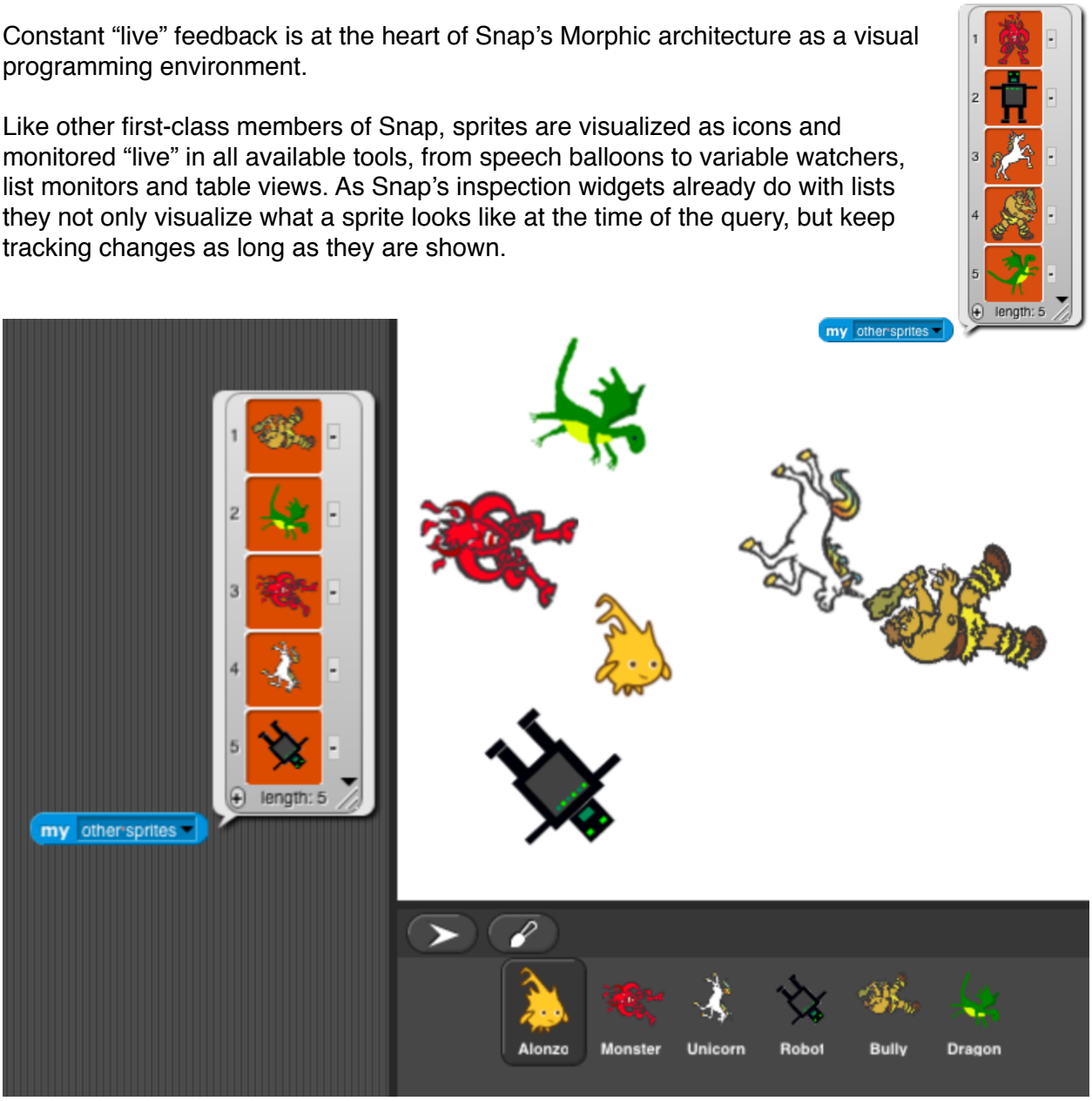
6) References to sprites are transient. Unlike references to lists but same as cloned sprites and captured continuations they do not survive saving and reopening a project. Instead the project must make sure to initialize them at runtime.

7) References to deleted sprites and deleted clones of sprites become “corpses”. Deleted sprites can still be passed as inputs into most primitives, e.g. to let another sprite “go” to them, but no script can bring them back onstage.

Liveness

Constant “live” feedback is at the heart of Snap’s Morphic architecture as a visual programming environment.

Like other first-class members of Snap, sprites are visualized as icons and monitored “live” in all available tools, from speech balloons to variable watchers, list monitors and table views. As Snap’s inspection widgets already do with lists they not only visualize what a sprite looks like at the time of the query, but keep tracking changes as long as they are shown.



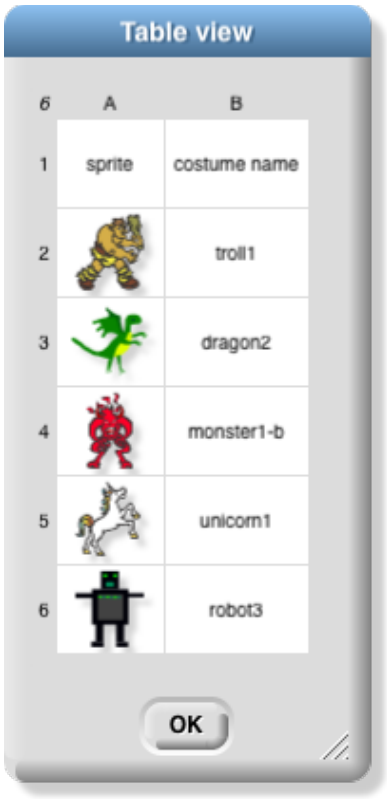
Thus, when a sprite rotates or switches to another costume, all widgets that observe that sprite likewise immediately reflect such changes.

Computing with Sprites

Sprites can now be used just like any other data in Snap. Since many existing blocks already accept a sprite name as input they can now also accept a reference to an actual sprite itself.

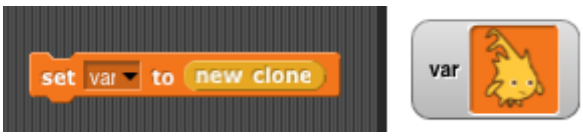
Here is an example of using MAP to create a two-column table showing the current costume name for each other sprite:

And to the right is the resulting table view:

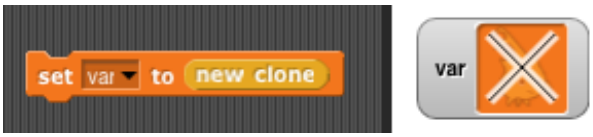


Identifying Individual Clones

Since sprites and clones can now be enumerated it is now possible to identify and track individual clones programmatically. It's also easy to build your own "make a clone" reporter:



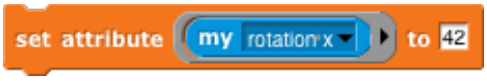
Just remember that pressing the red stop-sign button always automatically deletes all existing clones, "corpsifying" all references to them:



Manipulating Sprites Programmatically

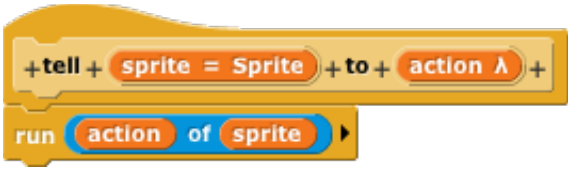
Using the new overloading semantics of the SET block you can now build your own “set attribute” command:

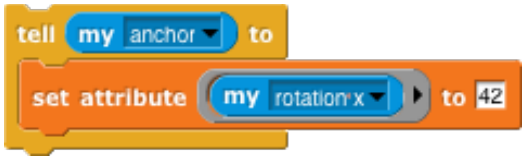
You can then use this custom block to programmatically alter (some of) the sprite’s attributes, such as the x-coordinate of its rotation center:



The rotation center is the point around which sprites turn. It is also the coordinate that’s used when the distance to a sprite is measured, regardless whether it lies within the shown costume or outside of it. Sprites store a separate rotation center for each costume. If you change a sprite’s rotation center it get associated with the sprite’s current costume. This way you can create several costumes for different rotation centers, for example to simulate a hinge or moving parts.

The existing official tools library already features TELL and ASK blocks that let you programmatically control a sprite using scripts:





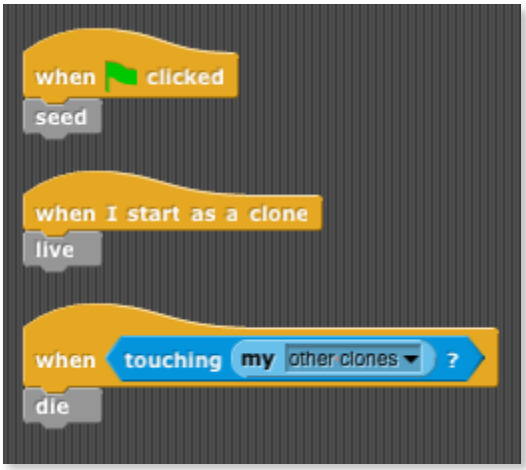
Because the definition body of these blocks use the built-in “OF” primitive that accepts a sprite name in its right input slot, these blocks can now also be used to direct a script to another sprite, or to another individual clone.

Using the same mechanism you can enumerate ALL the sprites onstage, including the one that’s running the script:

Fun Ideas to Explore

A Simple Population Simulation

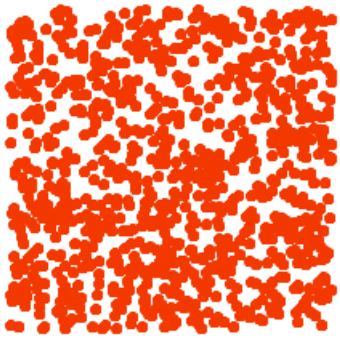
Here's the outline of a very simple simulation of a "living" Snap-turtle population. The simulation gets seeded with a single clone. Clones reproduce asexually every once in a while, but as soon as one clone bumps into another both die:



http://snap.berkeley.edu/run#cloud:Username=jens&ProjectName=population

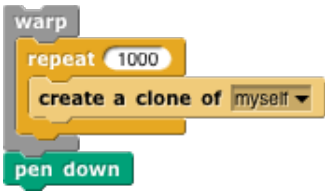
Notice that collision detection is turned into an event using the generic "WHEN" hat block unique to Snap that takes an arbitrary predicate as input. Also notice that a list of sprites is directly given to the TOUCHING predicate, instead of having to explicitly enumerate the clones.

Bug Drawings



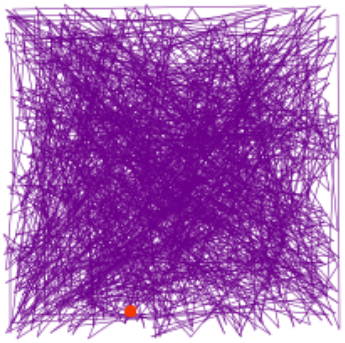
Clone a sprite a number of times distributing the clones randomly across the stage, then let the sprite "sweep" all clones while drawing an interesting pattern:

Here's how to setup the "clone-field" with a thousand dots fairly quickly:



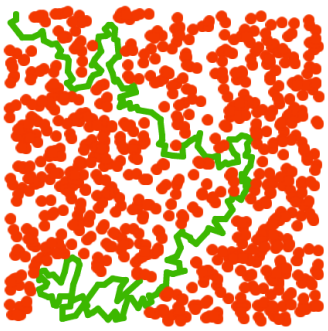
Random Enumeration

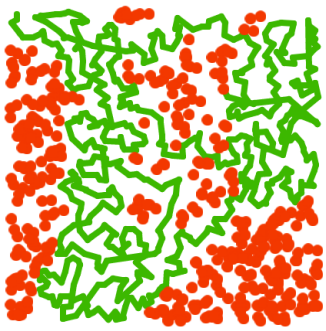
Using the FOR EACH block from the official tools library (built in Snap itself) you can let the sprite travel to each clone while leaving a pen trail. Of course, since the clones were placed randomly to begin with, the resulting pattern is also just random criss-crossing paths, often spanning what seem like unreasonably long distances for a bug to crawl:

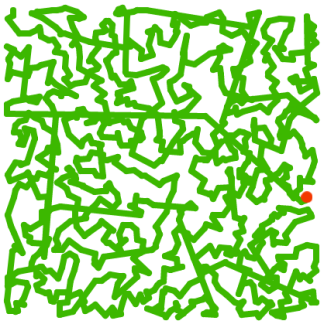


From Nearest to Nearest

A more interesting and efficient path can be accomplished by always collecting the nearest clone until none are left:





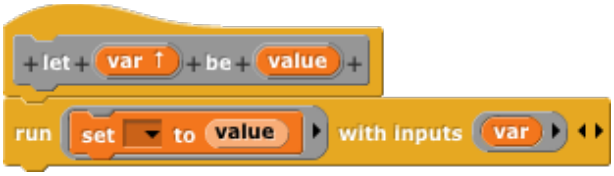


This path can be expressed with a fairly simple script.

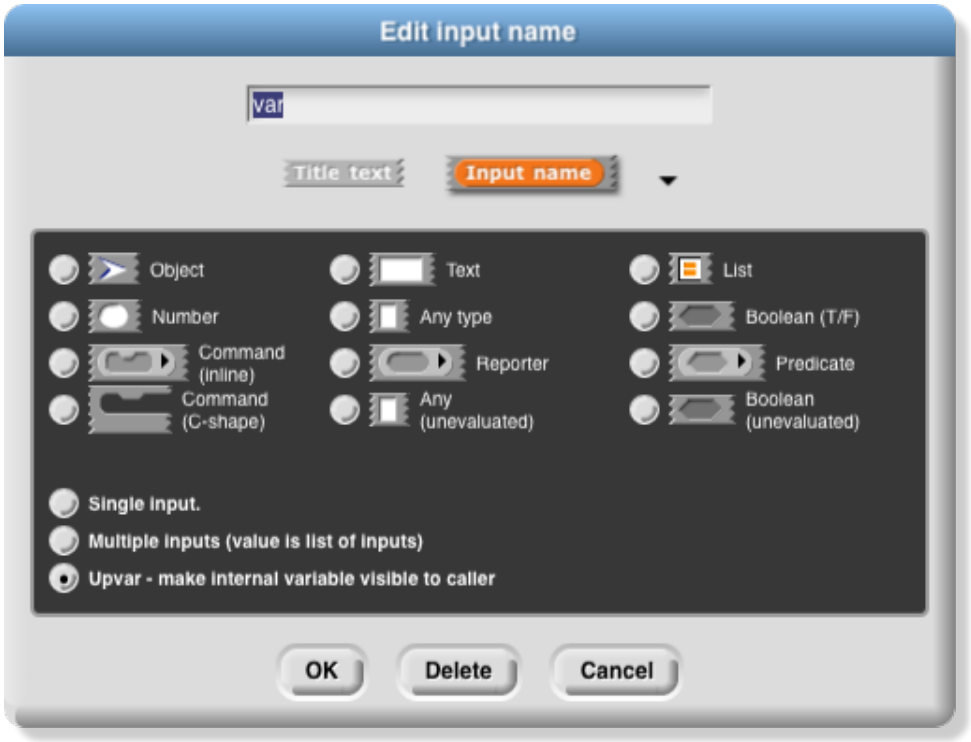
The EMPTY? predicate comes with the standard tools library.

is semantically the same as:

It can easily be defined using an “upvar”:

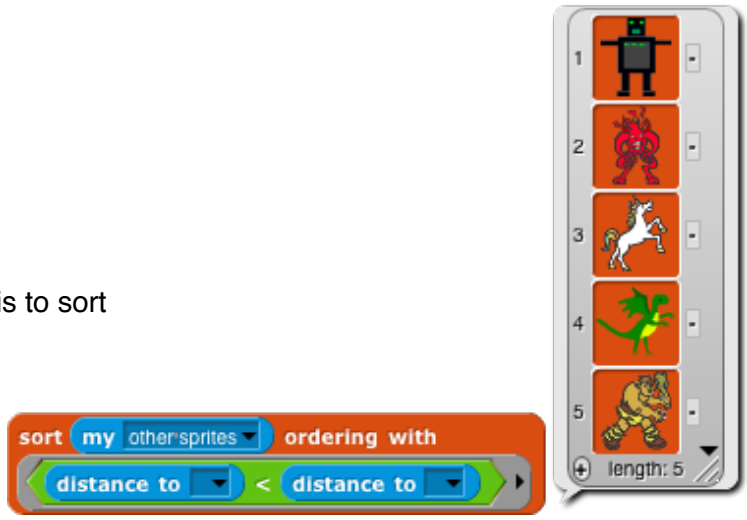


To make “var” an “upvar” click on the blob in the block definition inside Snap’s block editor, go to the long form input dialog, and select the radio button labelled “Upvar”:

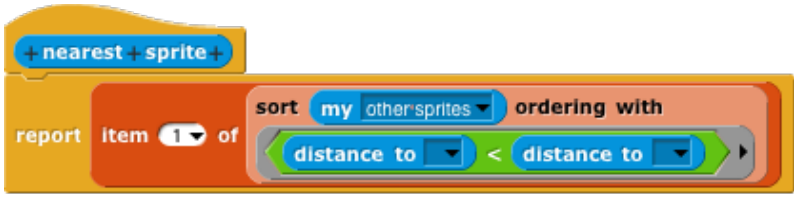


As a result the “var” blob in the custom block definition header will be adorned by an up-arrow icon.

One way to find the nearest sprite is to sort all sprites by their distance…

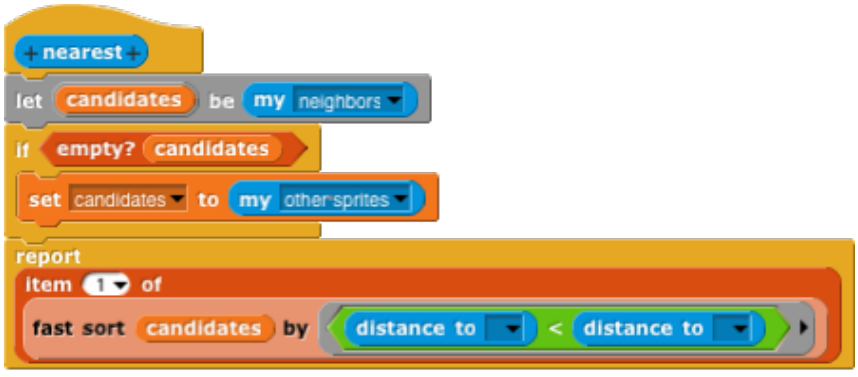


… and then to simply pick the first one and return it:



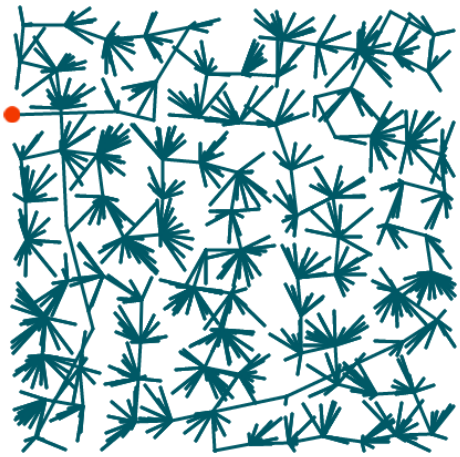
The SORT reporter is in the standard “List utilities” library. It works fine but is somewhat slow for the purpose of constantly sorting up to a thousand sprites at each step. I would therefore recommend the following quasi-primitive that uses JavaScript for sorting a Snap list, and is very fast:

A further optimization can be accomplished by only looking at the “neighbor” sprites which are nearby, and only sort the whole clone population if no neighbors are left:



From Neighborhood to Neighborhood

The approach - first collecting all the neighbors before moving on - also lends itself to explore alternative interesting patterns like this one:



Notice that in this script the “collector” sprite keeps returning to the first “node” for each set of neighbors, even though that “node” is clearly also a neighbor itself and therefore gets deleted first. This is an example of how a “corpsified” sprite (or a deleted clone) is still accessible and useful, even though it cannot be placed onstage again, and it can no longer be “told” anything.

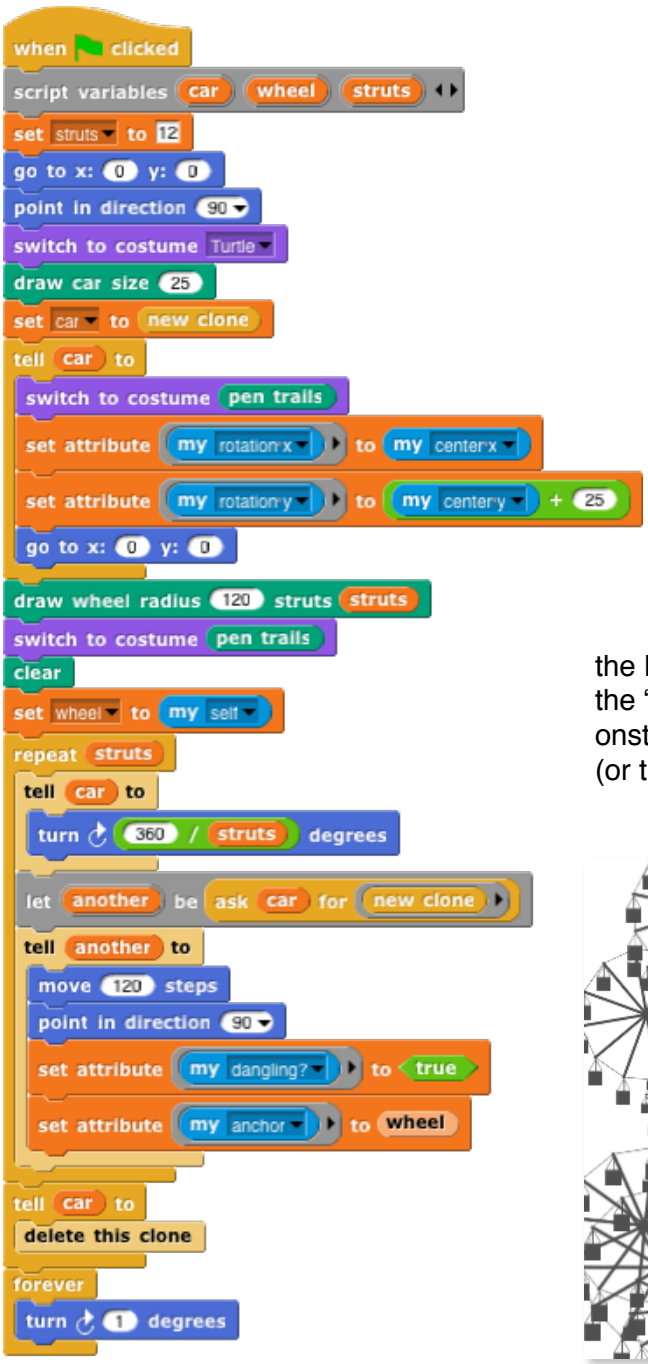
Here’s the link to a complete project:

http://snap.berkeley.edu/run#present:Username=jens&ProjectName=Woodworm

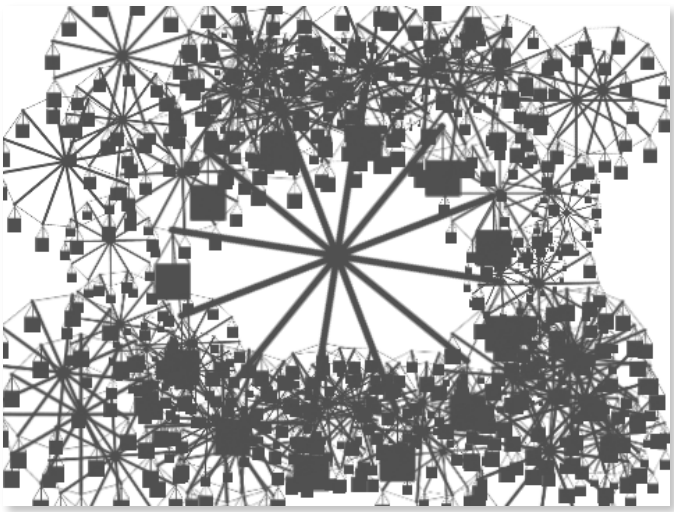
Moving Parts

A unique feature of Snap is being able to construct composite machine simulations out of nested parts, and being able to specify both the rotation center and the turning behavior of each part. New in v4.0.7 is the ability to do this programmatically. Now you can even “bootstrap” a machine from just any plain turtle, by drawing costumes, cloning the parts, assembling them to a whole, and then even cloning the composite including all its parts as often as you wish:

http://snap.berkeley.edu/run#present:Username=jens&ProjectName=Ferris%20Wheel%202016



the PEN TRAILS reporter can be imported from the “pen trails” library. It grabs whatever is drawn onstage and turns it into a costume that a sprite (or the stage) can wear.



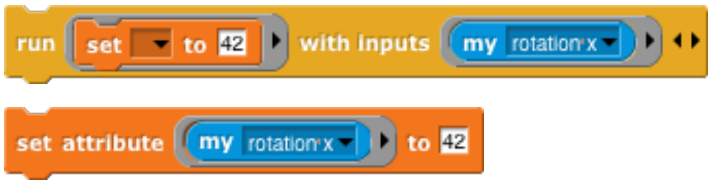
Rotation Center

As explained above the rotation center is the point around which sprites turn. Each costume of a sprite has its own rotation center. You can edit the rotation center of a costume by selecting the “cross-hairs” tool in the paint editor. By default the rotation center is the geometric middle of the costume’s bounding box. You can move it to somewhere else in the paint editor by dragging the cross-hairs marker around the canvas:

The absolute rotation center on the stage is always the same as the “position” coordinates of a sprite, and therefore redundant as a reporter (function):

it is also used to determine the distance of the sprite to other objects, such as the mouse-pointer or other sprites.

The reason the rotation center coordinates are available in the MY reporter is for them to be programmatically settable as attributes:



Notice that using SET on rotation center coordinate attributes references them to an absolute position on the stage at the time when the block is executed. This way you can set the rotation center using the mouse or relative to another sprite.

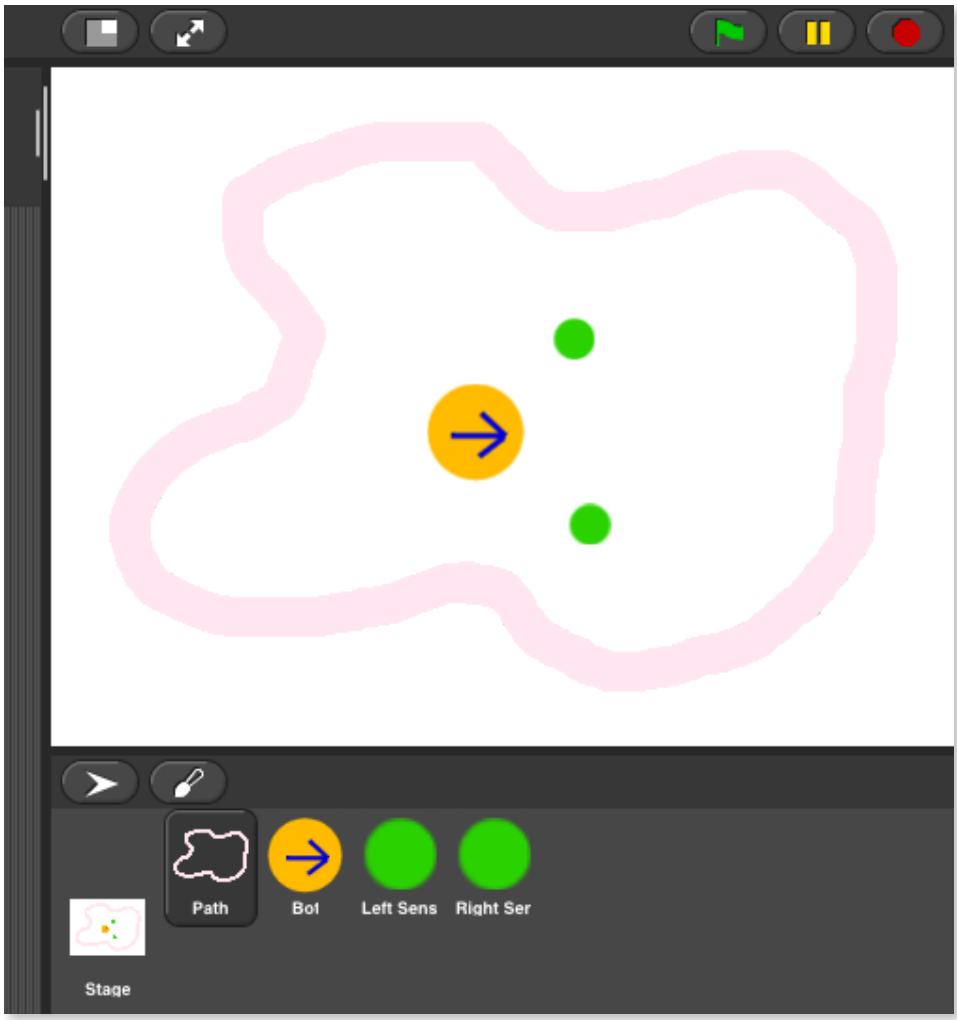
Since the rotation center and the sprite's position are identical you need another "fixed" coordinate that always stays the same, relative to the sprite's position, scale and rotation. These are the sprite's "center" coordinates, which are always the exact geometric middle of the sprite on the stage.

Using the "center" coordinates you can both reset a sprite's rotation center, and also place it relatively, as the "Ferris Wheel 2016" example above does for the "car" clones:

Nesting Sprites

Similar to assembling scripts by sticking together individual blocks Snap lets you assemble complex simulations by sticking together sprites. In Order to better understand what's going on the Ferris Wheel example above I'll recap how sprite nesting works using just the GUI with a simple, yet archetypical "line-following robot" example:

This stage contains all the ingredients we need for a line-following robot simulation: a Path (notice the path is not drown onto the stage but a separate sprite), a "Bot" sprite represented by a large orange dot with an arrow pointing in the direction given to the sprite, and two smaller dot-sprites named "Left Sensor" and "Right Sensor". One was created simply by duplication the other.

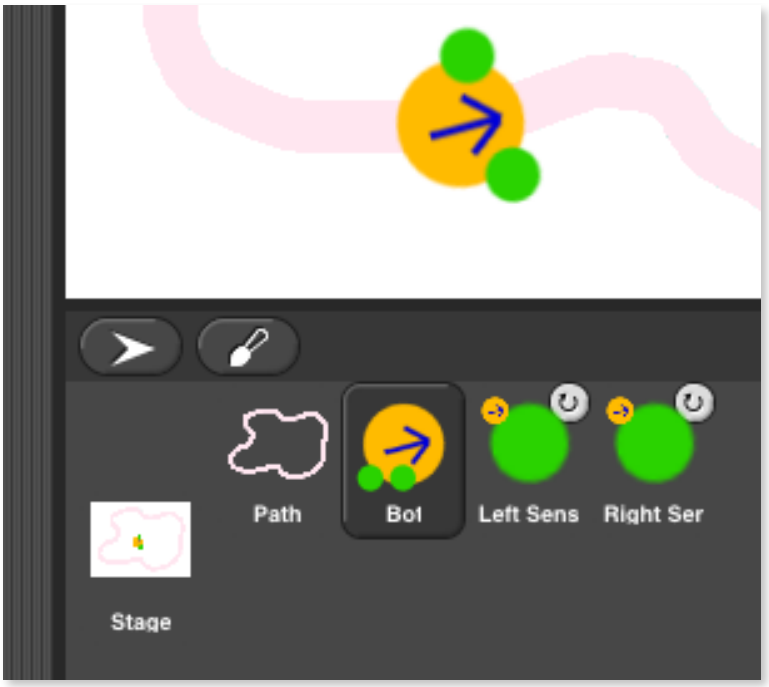


A sprite that's attached to another sprite is called a ***part***, a sprite to which one or more parts are attached is called the ***anchor*** of its parts. Parts themselves can be anchors for one or more parts of their own.

In order to attach a part to an anchor first **position it on the stage** where it should be. In the example to the left the sprite named “Left Sensor” has already been placed to the left-hand side of the “Bot” sprite. Then use the mouse to ***pick up the part’s icon from the corral*** underneath the stage and drag the icon across the stage over the anchor-sprite, to which you want to attach it. When the mouse holding the part-icon enters the designated anchor the anchor displays a ***yellow attach-target feedback halo***. Now you can drop the icon onto the anchor sprite. The icon will slide back to the corral, but the part has been attached and is now “part of” the anchor sprite. As such it now moves, rotates, scales and senses with and for the anchor.

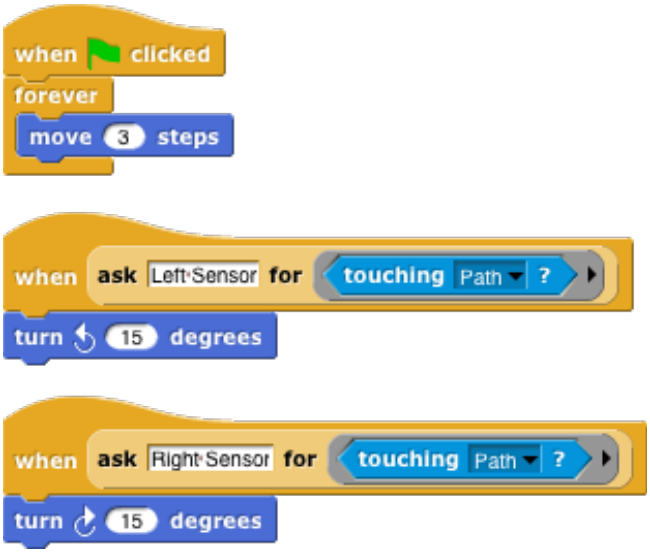
You can use the sprites’ context menus to adjust the positions and to again detach parts from their anchors.

After attaching both sensors the icons in the corral now show schematic symbols of their attached parts (at the bottom), or of their anchor (top-left). Parts furthermore have a little button that let you toggle their rotation dependency from “rotate-with-owner” (default) to “rotate independently” (also called “dangling”):



Now all that’s left to do is write the program logic that lets the robot follow the Path autonomously. Must be hard, right? After all, a line-following robot is the “hello world” of every robotics club.

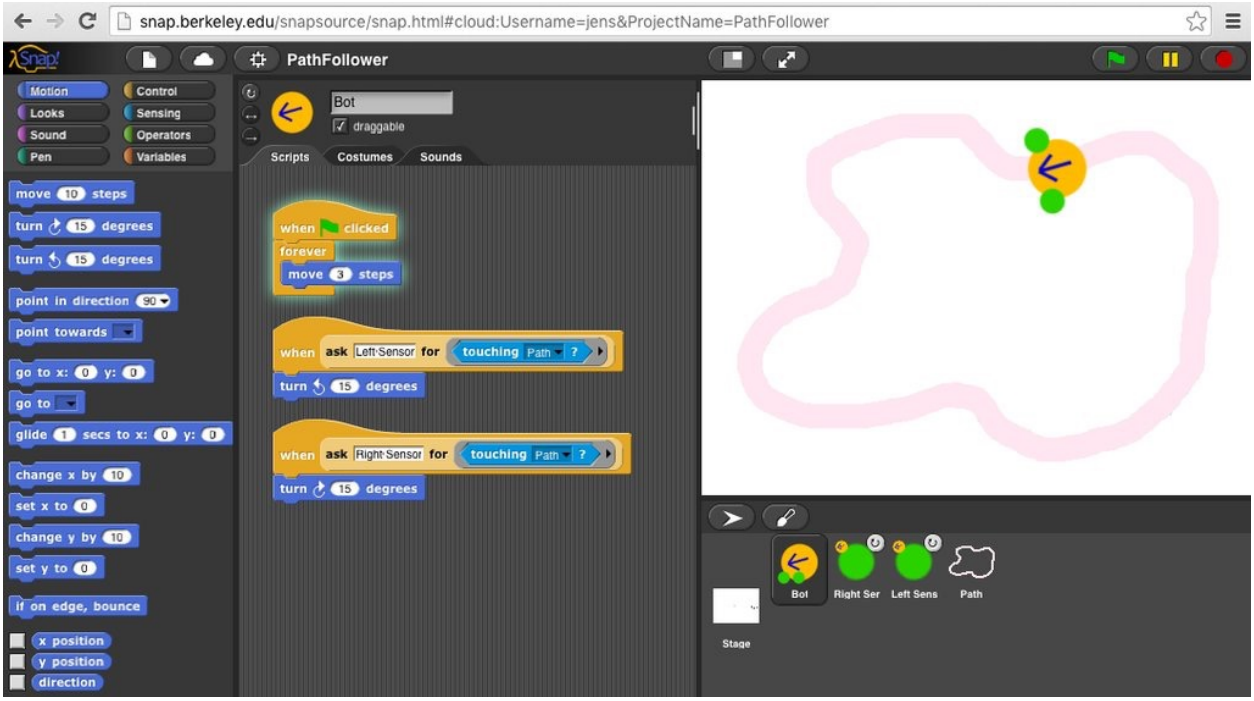
Well, here's the good news: It only takes three tiny scripts in the "Bot" sprite's scripting pane:



You can find the ASK reporter in the standard tools library. It has been written in Snap itself, so you can open it in the block editor and see how it's done (no magic, same as TELL above).

That's it, you're done:

http://snap.berkeley.edu/run#cloud:Username=jens&ProjectName=PathFollower



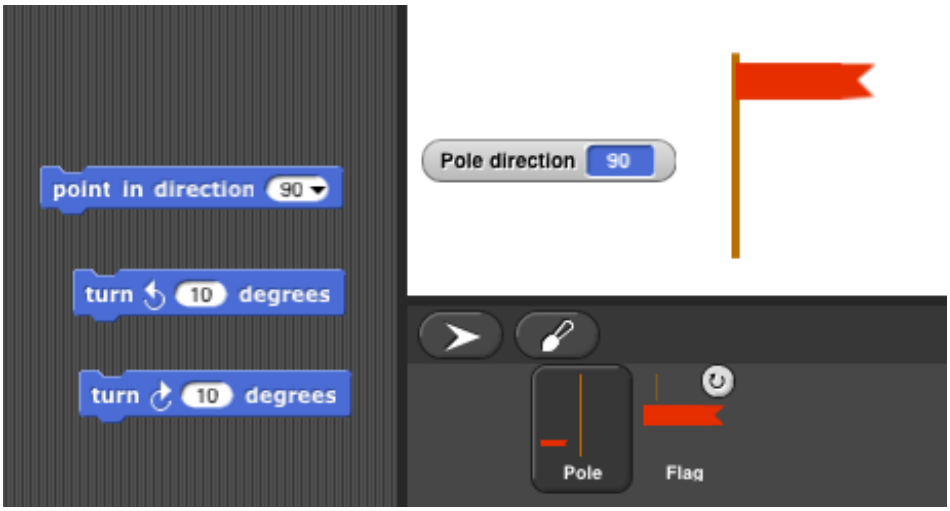
Nested sprites have been a feature of Snap*!* since 2009 when it was still named BYOB (v2). What's new in v4.0.7 is that you can now also use blocks to dynamically attach and detach sprites to each other, and that this also lets you nest clones that don't have icons in the corral:

Likewise you can detach a part by setting its anchor attribute to nothing (or something that is not a sprite).

Turning Dependencies

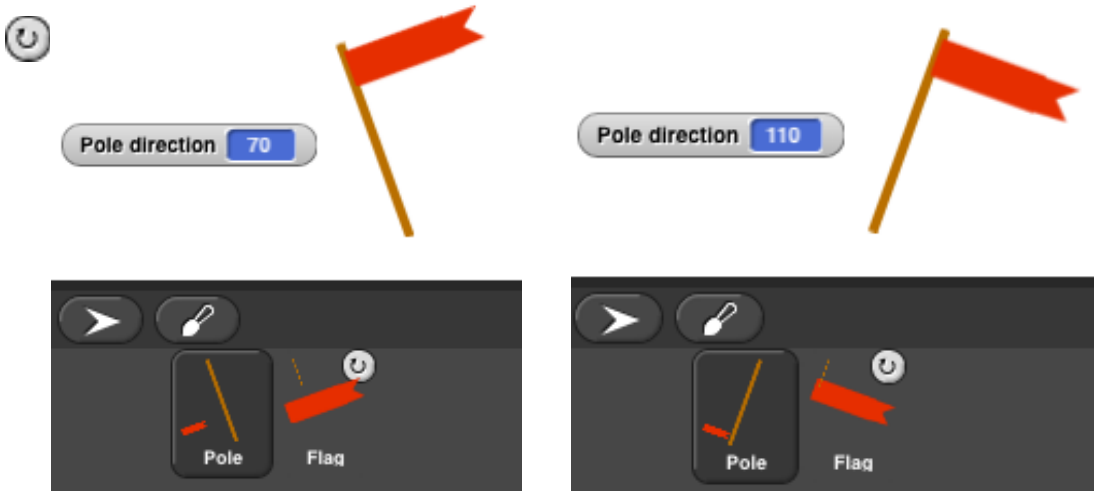
Sprites that are attached to ("part of") another sprite ("anchor") move, scale and rotate and sense along with the anchor. They can, of course, always have their own scripts that move, scale and rotate, but, when part of another sprite those actions will be relative to the anchor. In the case of rotation it can be very useful to let parts rotate independently. This lets you better simulate machines and "hinges".

Take this example of a nested sprite consisting of a flag pole and a flag:

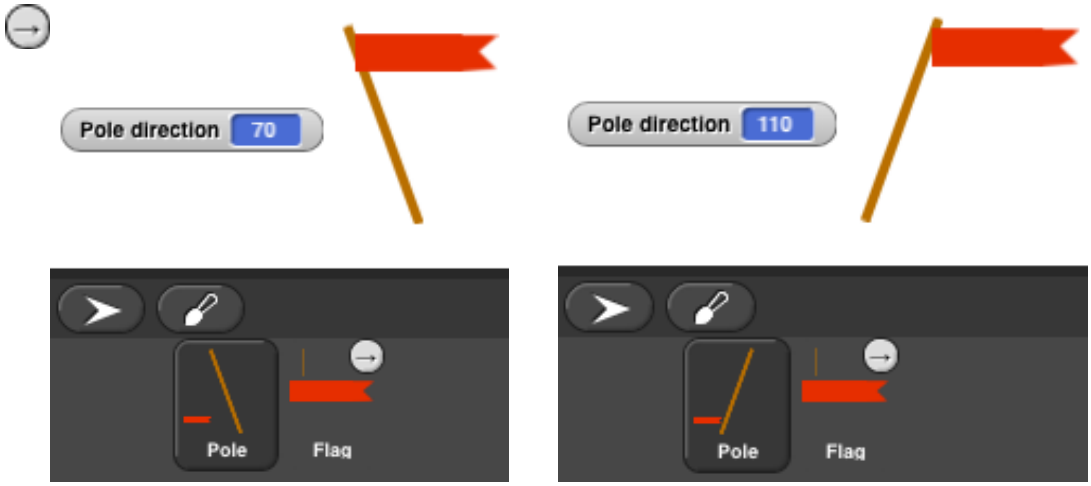


By default, parts rotate with their anchors.

Rotating with the anchor is indicated by the circular icon on the little toggle button on the Flag sprite's corral icon:



clicking on the rotation-dependency button toggles the turning-dependency to "turn individually", which is indicated by a straight arrow icon on the button:



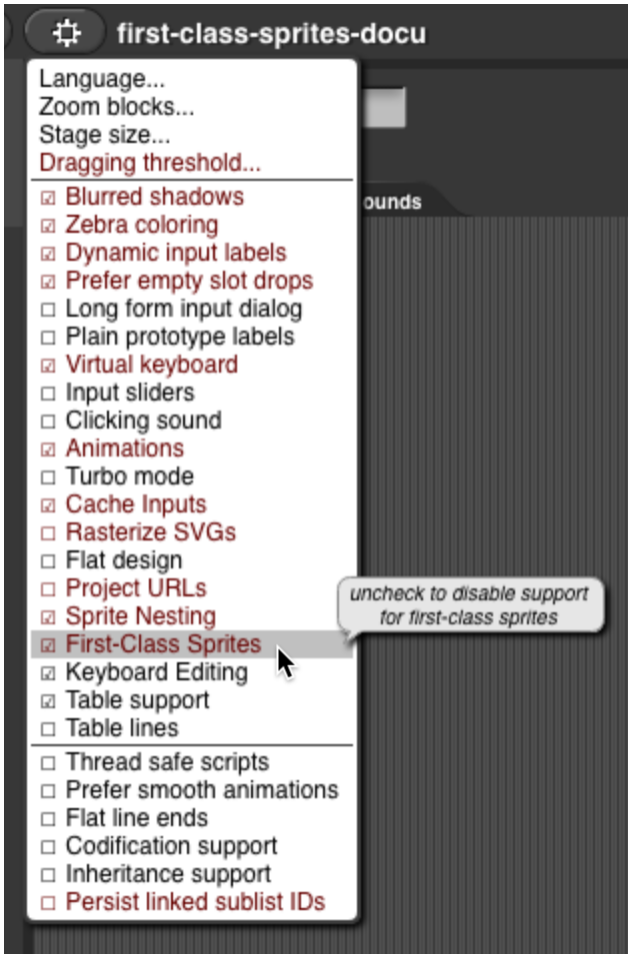
As with nesting itself, this feature has been a part of Snap since the days of BYOB v2. New in v4.0.7 is that you can now both query and set the rotation dependency using the "dangling?" option of the MY reporter and the SET block:

This way machine parts can be modeled easier, as well as the "dangling" cars of the turning Ferris Wheel above.

Disabling Support for First-Class Sprites

Some variants and forks of Snap replace its default 2D cartoon “sprite” stage metaphor with their own microworld, e.g. 3D generative art environments (Beetleblocks, CSnap), data base queries (DBSnap, DataSnap, JMU Bags, SQL-Snap), interactive graphs (Edgy) and particle systems (Cellular, Scribble) or robot drone control (ARDrone) and embroidery (Stitchcode) etc. While these extensions might wish to stay current regarding Snap’s code base they also might prefer not to have the “sprite” type in their language.

First-class sprite support can be disabled by shift-clicking on the gears button and un-selecting the according preference. Snap forks might want to do this in their startup code.



Enjoy!
-Jens