SnakePit

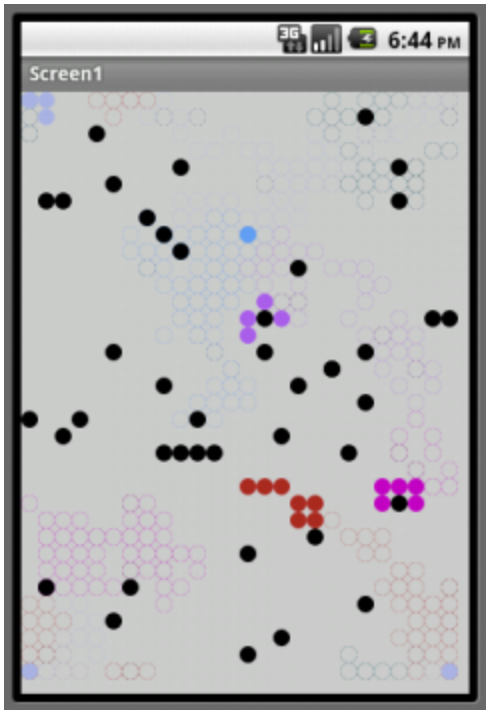
January 2018

Purpose

This project demonstrates how to use lists to run a multiple Snake simulation on a Canvas, without using any Sprites or Balls. Instead, it draws Circles on grid squares (cells) for snake eggs, and hatches a few of them into colored snakes that move randomly and grow in length as they eat free standing snake eggs. When a snake bites another snake, the tail breaks off and reverts to individual snake eggs.

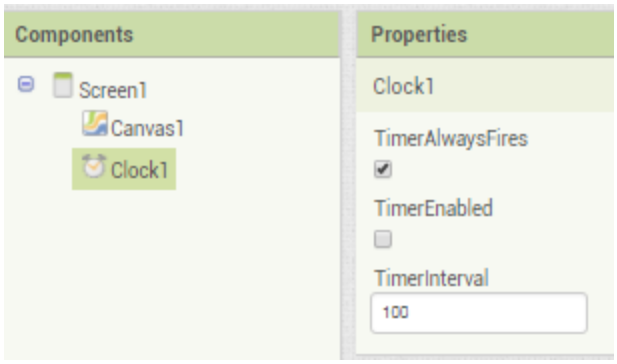
This is an advanced project, and employs doubly linked lists.

Capture



The black dots are snake eggs. The colored dots are snakes, fairly tightly coiled because of the random motion automatic movement routine. The brown snake has eaten the most eggs so far, and the light blue snake has eaten no eggs yet. The colored bubbles are artifacts of the drawing and erasing of the colored dots that make up the snakes, as the snakes move. (I rather like them, so I have left them as is.)

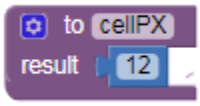
Designer Clock1



The Designer is as simple as can be, with just the Canvas and a Clock.
The Clock is disabled, to allow clean startup, and hard wired for 100 milliseconds per tick.
Feel free to run the clock faster to experiment.

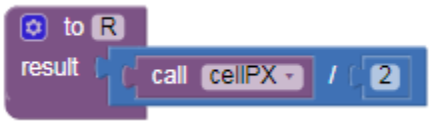
Graphics

cellPX



This function returns the number of pixels in the side of each grid cell.

R



R is the radius of the circles used to draw the snake bits. Anything other than half the cell size leaves debris on the canvas or makes the snakes look too loose.

Cell storage

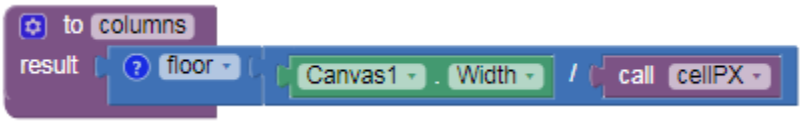
Cells are identified by row and column number, 1 based. To combine both the row and column number into a single value, we assume no more than 999 rows or columns, and express a (row, column) as a single number, rrrccc, using multiplication by 1000. To go back and forth between cell identifiers and rows and columns, we do modulo math.

We don't actually keep a 1,000 by 1,000 array. We just keep the locations of the small fixed set of eggs that we start with, that serve as links in the chains that are our snakes.

row

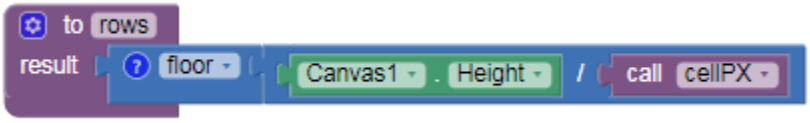
Given a cell ID, return the row number of that cell.

columns



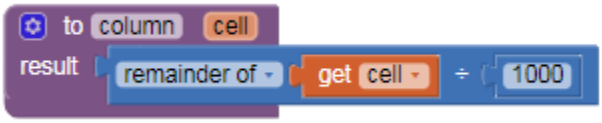
Return how many columns will fit onto the Canvas based on its Width and the number of pixels per cell.

rows



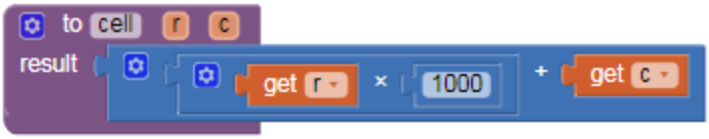
Return how many rows will fit onto the Canvas based on its Height and the number of pixels per cell.

column



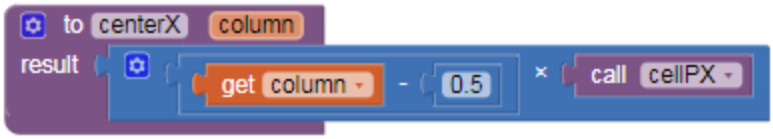
Extract the column number from a cell ID.

cell



Build a cell ID from a row number and column number.

centerX

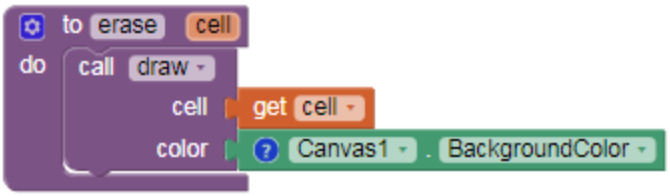


Return the x value (in pixels) for the center of a circle at a given column number.

centerY

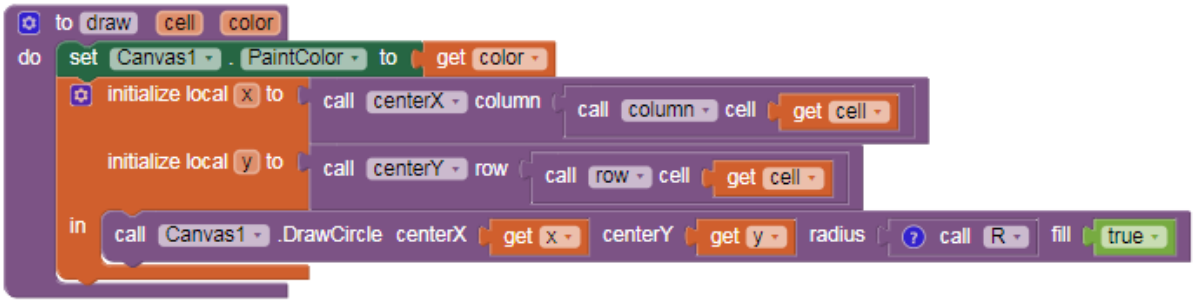
Return the y value (in pixels) for the center of a circle at a given row number.

erase



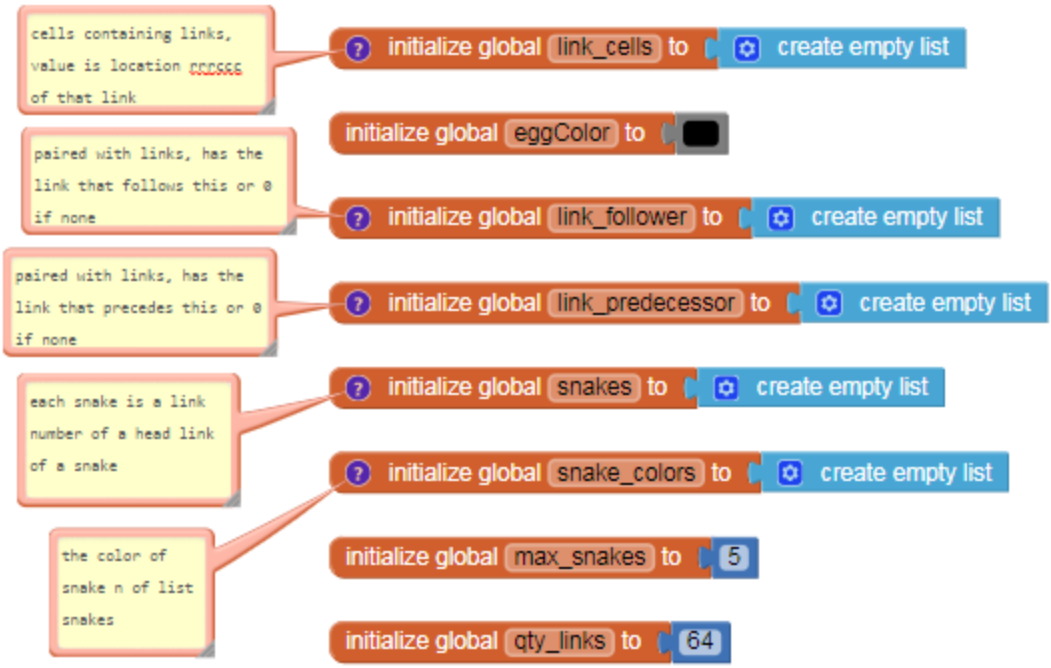
Erase the circle at a given rrrccc cell. This technique leaves a bubble on the Canvas. I could have changed this to draw a fat line to fill the entire rectangular cell, to eliminate the bubble, but I like the bubble trail left by the snake.

draw



Several layers of conversion take the row and column numbers from the cell ID, then extract the centerX and centerY graphic coordinates for the center of that cell. Then we draw a circle of radius R and the given color at that location.

global_snake_storage

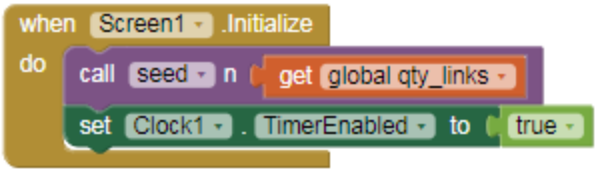


Link storage

Snakes are chains of links in a doubly linked list of constant size, once the board has been seeded with eggs. The constant **qty_links** controls how many links (rrrccc items in list **link_cells**) will be produced, and that number will remain unchanged after the seeding operation. Because that number remains unchanged, we can make parallel lists of predecessor and follower link numbers (indices into **link_cells**). An egg at position 23014 (row 23, column 14) whose link is at index 9 in list **link_cells** will start out with 0 at index 9 of lists **link_follower** and **link_predecessor**, since it is a chain of length 1.

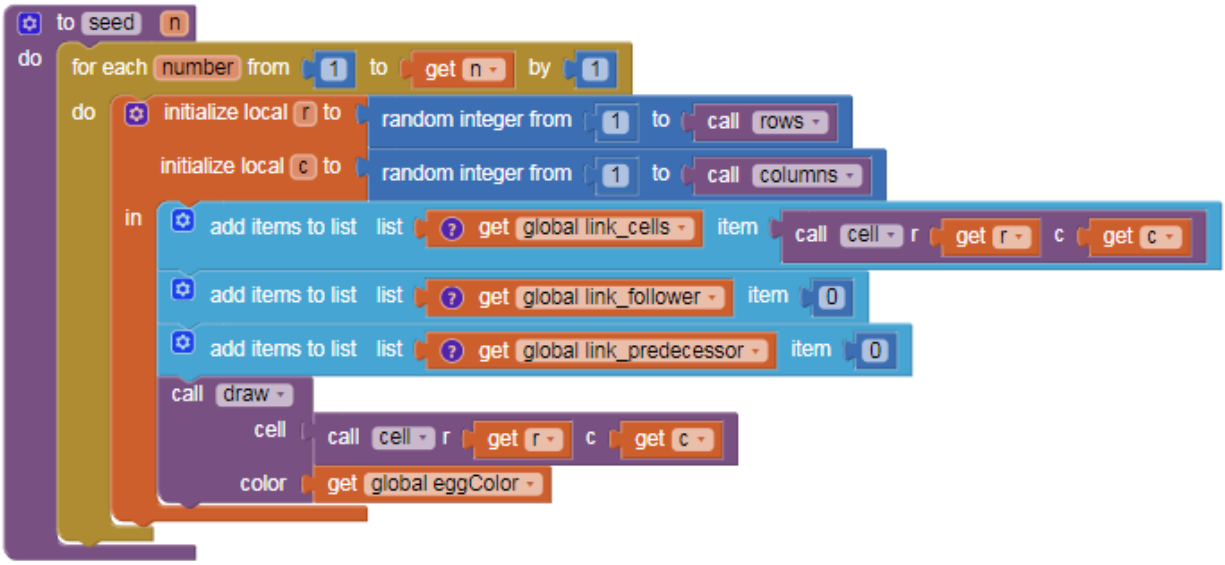
When we hatch an egg, we add its starting link number to list **snakes.** We limit the number of snakes for ecological reasons, based on constant **max_snakes**. List **snakes** has a matching list **snake_colors**, holding the color of snake **i** in list **snakes**. As snakes are born and die, we have to keep those lists in parallel.
A third list **snake_directions** might be used to keep persistent snake directions, to allow faster movement than the current Brownian Motion random model.

Screen1_Initialize



At app startup, we seed the lists of links and their structures, and paint them as eggs. We then start the Clock Timer to get some snakes hatched and moving.

seed



To seed the board, we generate the required number of links, pointing to random row and column locations, and draw an egg at each location. Being eggs, they have no followers and predecessors for the chains they would be part of if they were parts of snakes.

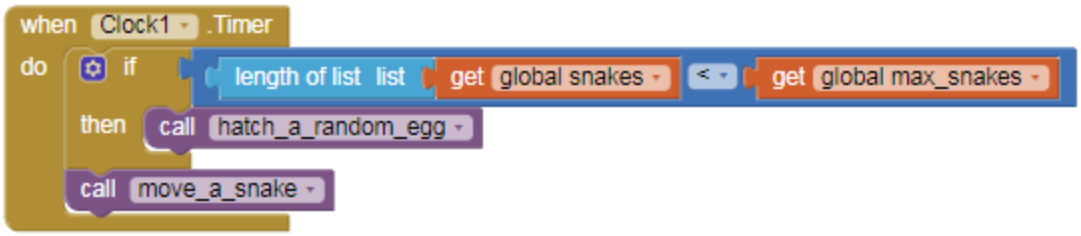
(Bug alert: I did not check for random duplicates here.)

next

To concisely navigate chains of links, we provide next and prev functions accepting a link (index into link_cells).

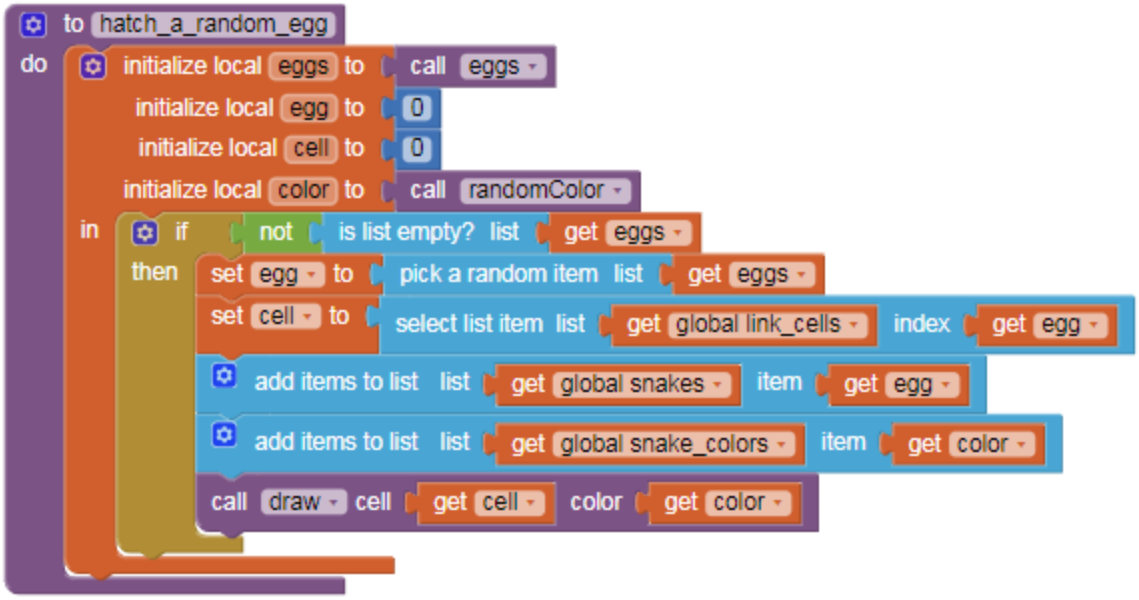
prev

Clock1_Timer



The Clock ticks rapidly, and does just a little bit at each tick, hatching one snake if there aren't enough snakes, and regardless, moving one snake.

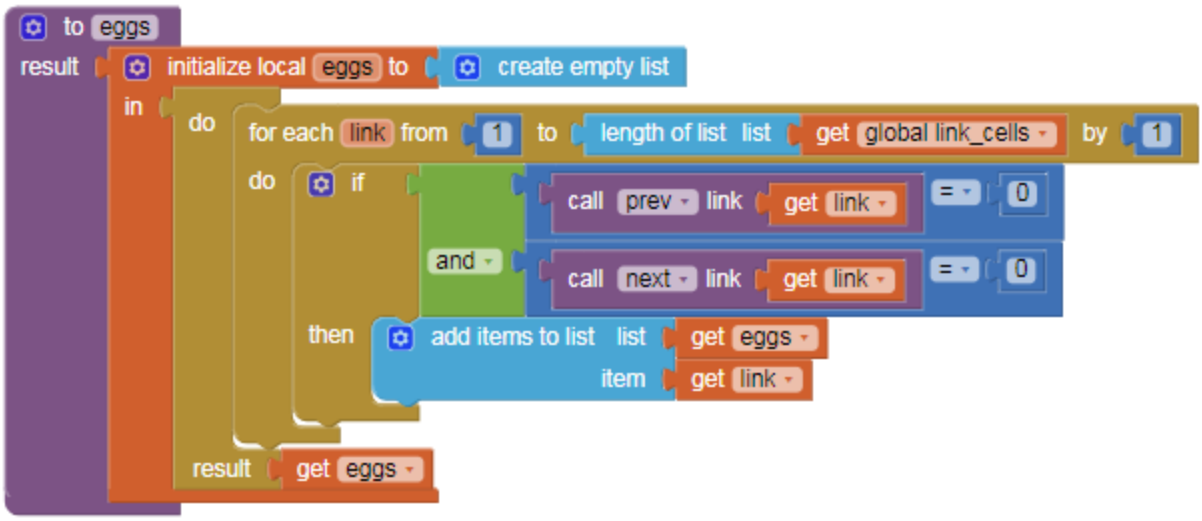
hatch_a_random_egg



The first step in hatching a random egg is to go through the lists of links, and collect the links that have no followers or predecessors, i.e. eggs. It's possible that there are none left, because they are parts of snakes now. In that case, we do nothing. If there is at least one egg on our list of eggs, we pick a random egg from that list. Since the returned egg is from a list of link indices in list link_cells, we need to look up the cell (rrrccc) for that link to draw it with a random color.

We record the head link of the new snake in global list snakes, and its color in the matching list snake_colors.

eggs



Eggs have no previous or next links.

randomColor

move_a_snake

To move a random snake, we pick a random snake from list snakes and move it.

move

To move a snake, we first need to look up the head link of that snake by its snake index, using our snake function. We then extract the rrrccc location of that link, and generate a list of its neighbors' rrrccc values. For the simplest possible implementation, we settle for picking a random neighbor rrrccc to try, using routine try_target.

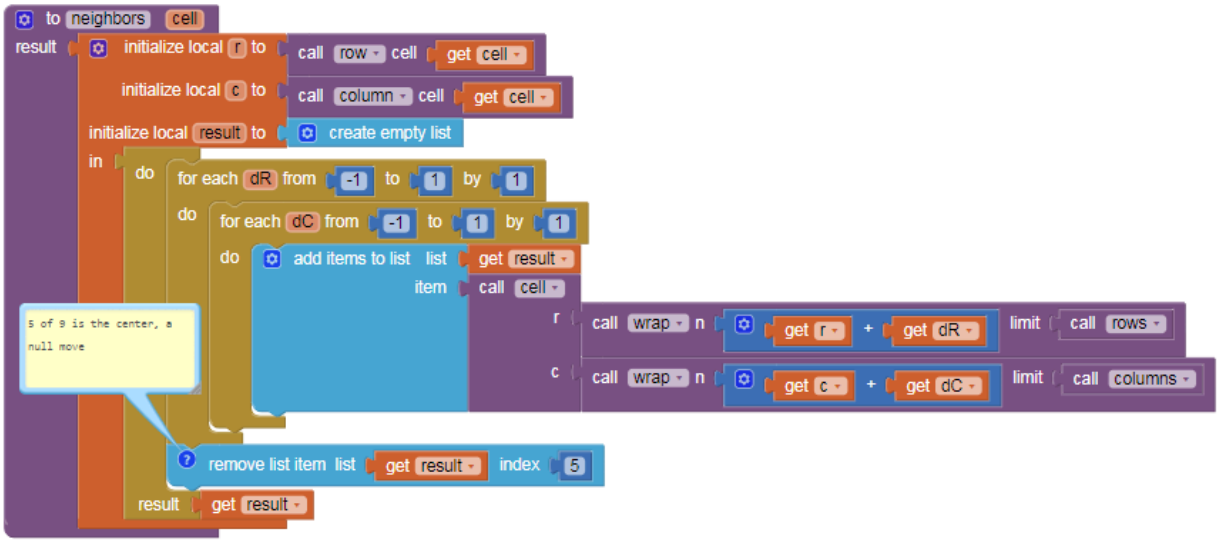
snake

This is a simple lookup in global list snakes.

location

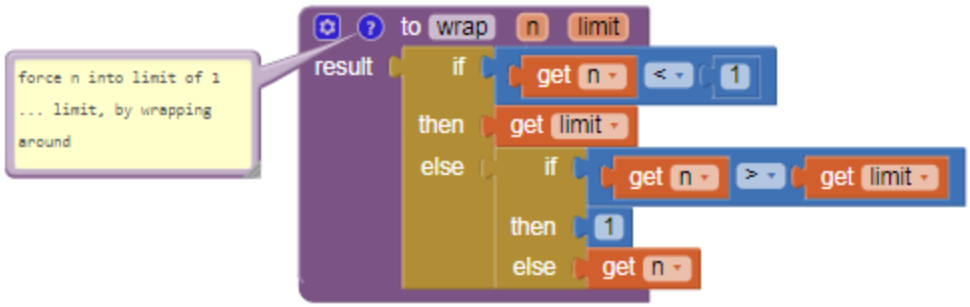
Links are indices into the list of rrrccc cells, global list link_cells.

neighbors



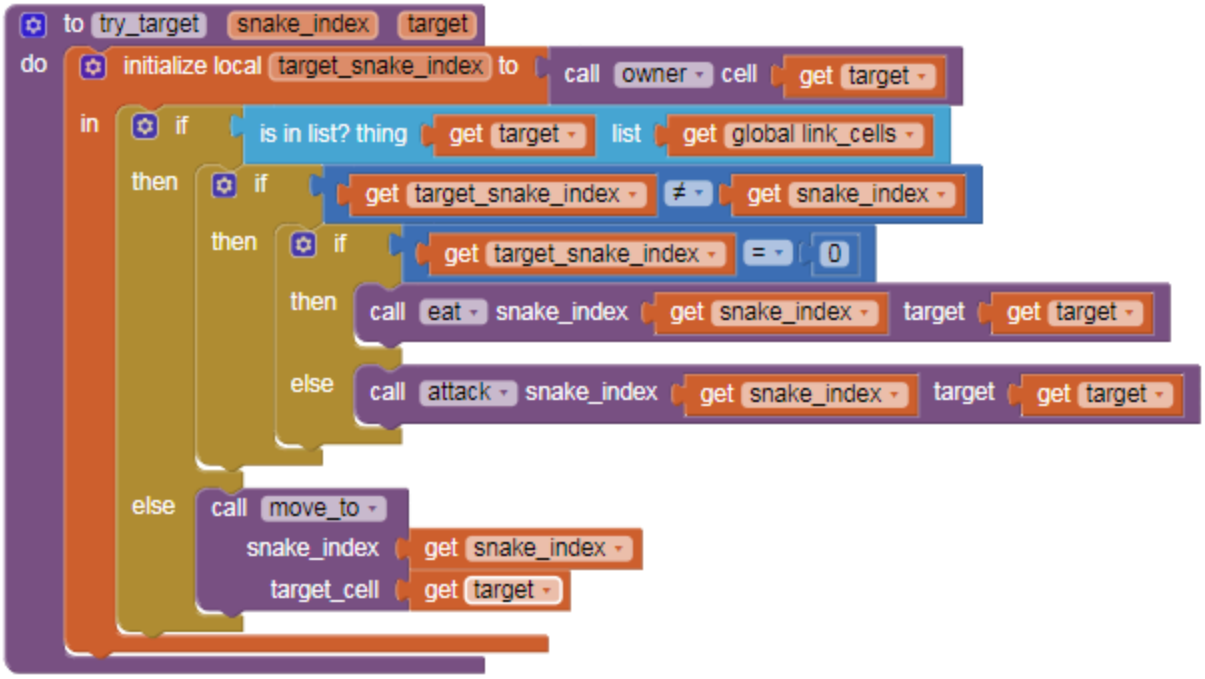
The cell ID is broken down into its row and column numbers, and its 8 surrounding cells are calculated, wrapped around the edges of the board, and reassembled into rrrccc cell IDs and returned.

wrap



We deal with the edge of the board by using doubly cylindrical geometry.

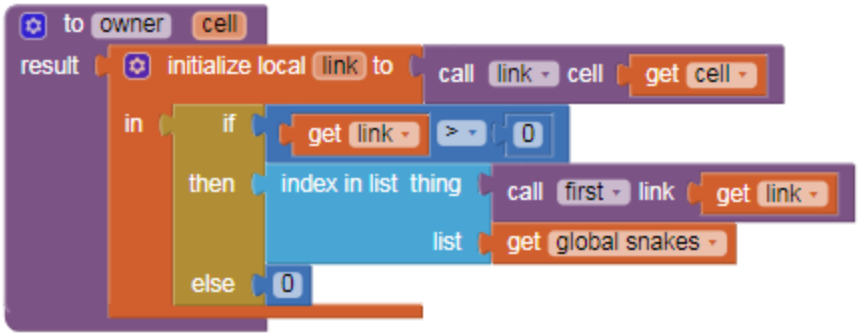
try_target



There are three possibilities when we want to move a snake onto a target cell:

1. The target has an egg, a link that's not part of a snake (no owner). Eat it.
2. The target has a link that's part of another snake. Attack it.
3. The target has no link, so it's empty. Move to there.

owner

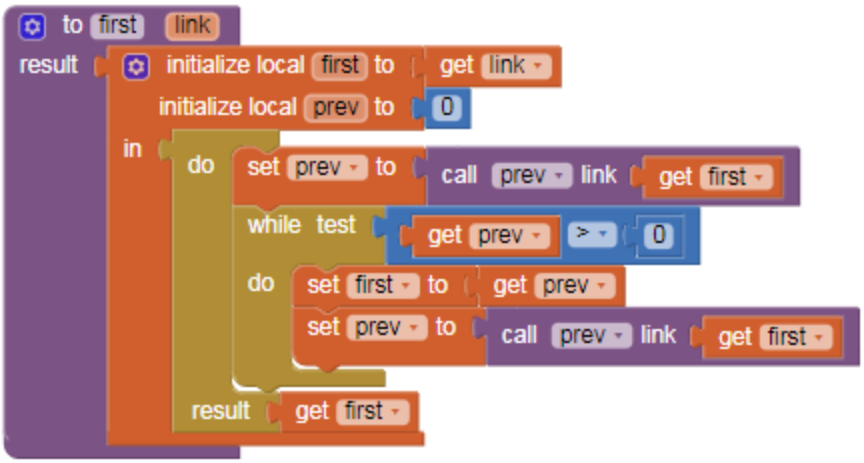


To find the owner (snake, if any) of an rrrccc cell, we first look up the link (if any) pointing to that cell. If a link is found (non-zero), then we look up the first link in whatever chain that link might be part of, then try to look up that head link in our list of snake heads (global **snakes**). Failure at any stage returns 0.

link

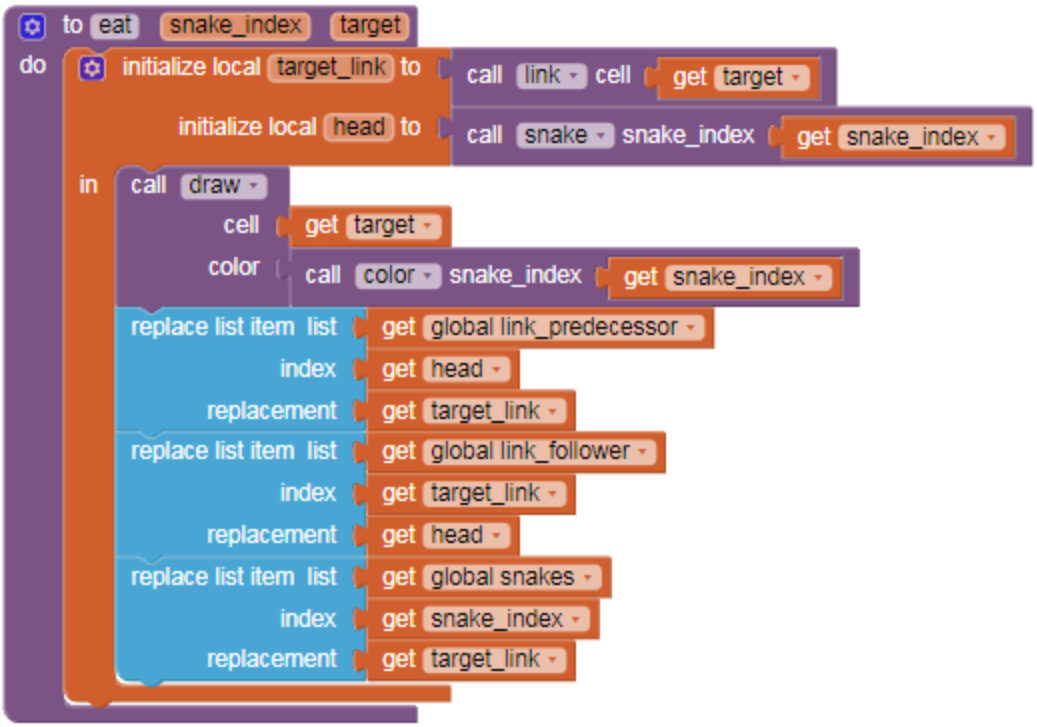
This function accepts a cell (row, column) and returns the link (index into links list) of the link on that cell, or 0 if the cell is empty.

first



This function accepts a link number, and follows the previous links to the very head of the chain (snake) holding this link.

eat

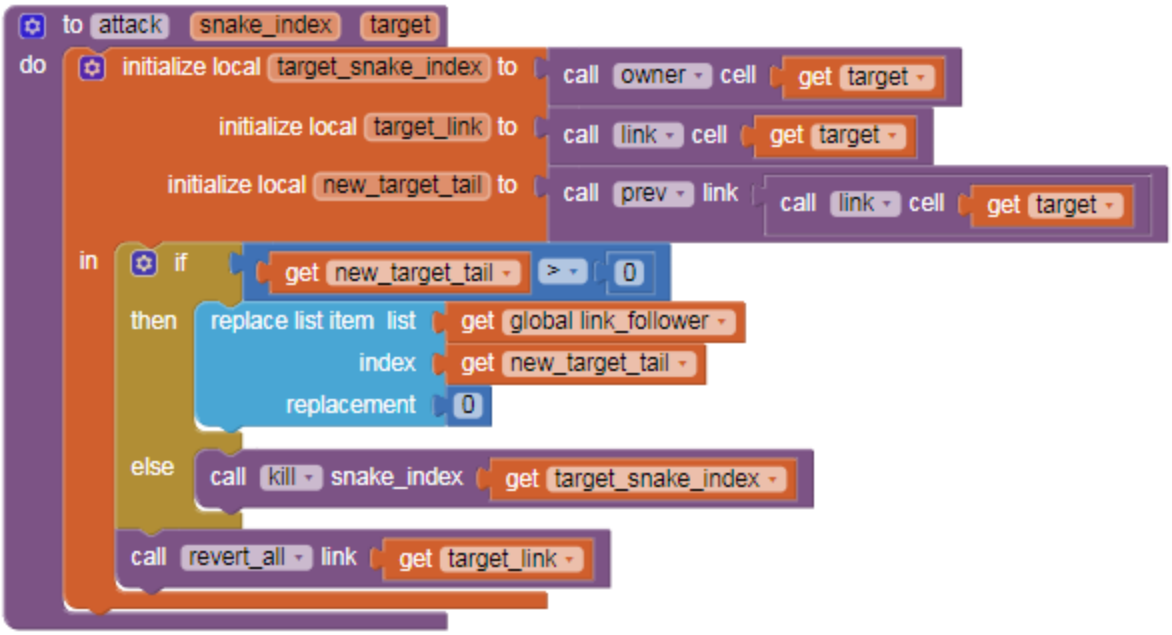


This procedure is for the case where a snake encounters an egg, a solitary link, and absorbs it into itself (eats it.) The **target** parameter is the cell (location) holding the target link to be eaten. The mouth of the snake is in its **head**, the link we can look up in the snake table.

To show that the link has been eaten, we change its color to that of the snake eating it.
To absorb the link into the snake,

- we make it the predecessor of the old head link,
- Point the follower of the new head back to the old head, and
- Update the snakes list to point to the new snake head link.

attack

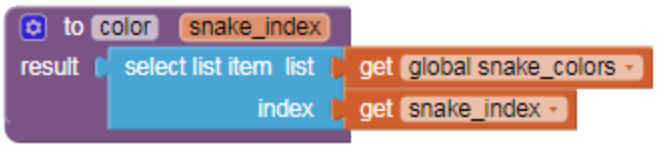


This is the case where a snake bites another snake at a target cell location. First we need to identify the victim, the owner of the target cell. We also identify which link is at the target cell. Because we are merciful, we will allow the target snake to escape and heal if it has been bitten anywhere behind its head, so we will need to identify the link that will serve as its new tail link. That link is the previous link to the bitten link.

If we have a new tail for the target snake, we seal it off by setting its follower link to 0. Otherwise, we have eaten the head of another snake, and must kill that snake entirely.

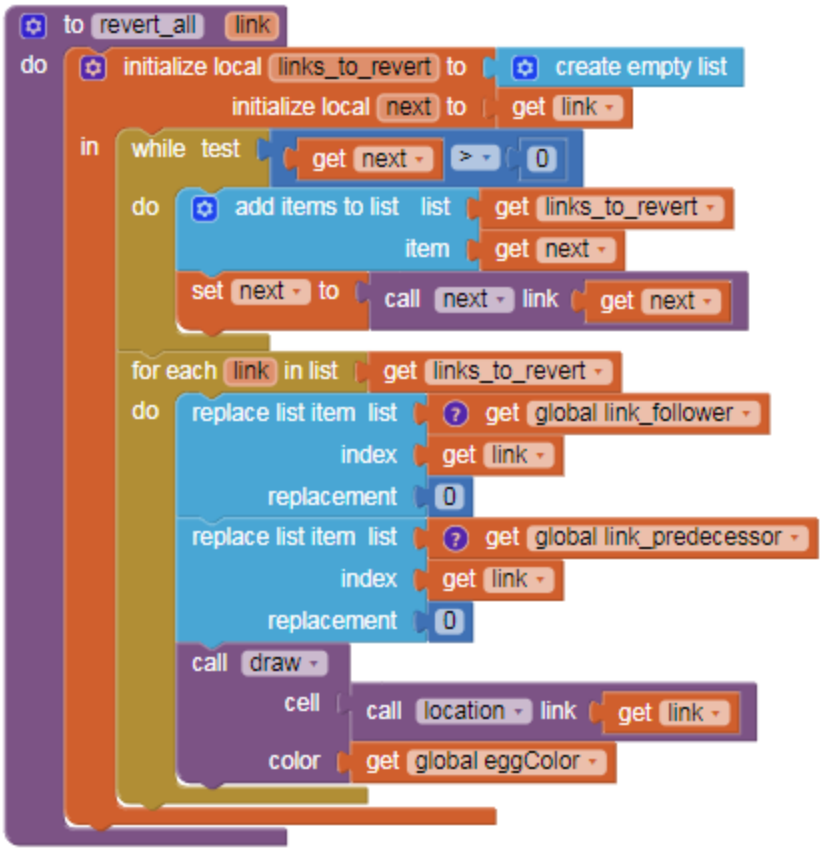
Regardless, the bitten target link and its followers in that snake are now loose food, and must revert to eggs.

color



This function returns the color of a snake, based on the lookup table **snake_colors**.

revert_all



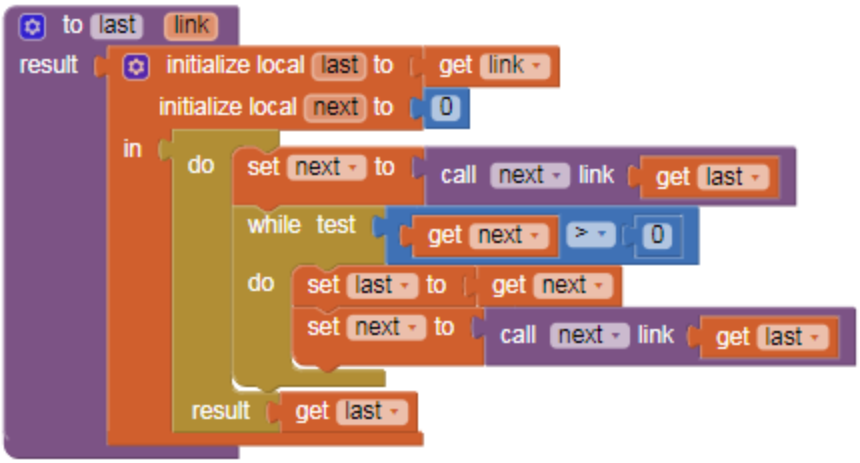
This procedure accepts the head link of a chain of links, that need to be de-chained and reverted back to eggs. It is a two phase operation:

- Gather the links in the chain into a list, following the next links,
- Revert each link in the list to an egg, by clearing its predecessor and follower links and redrawing it black.

kill

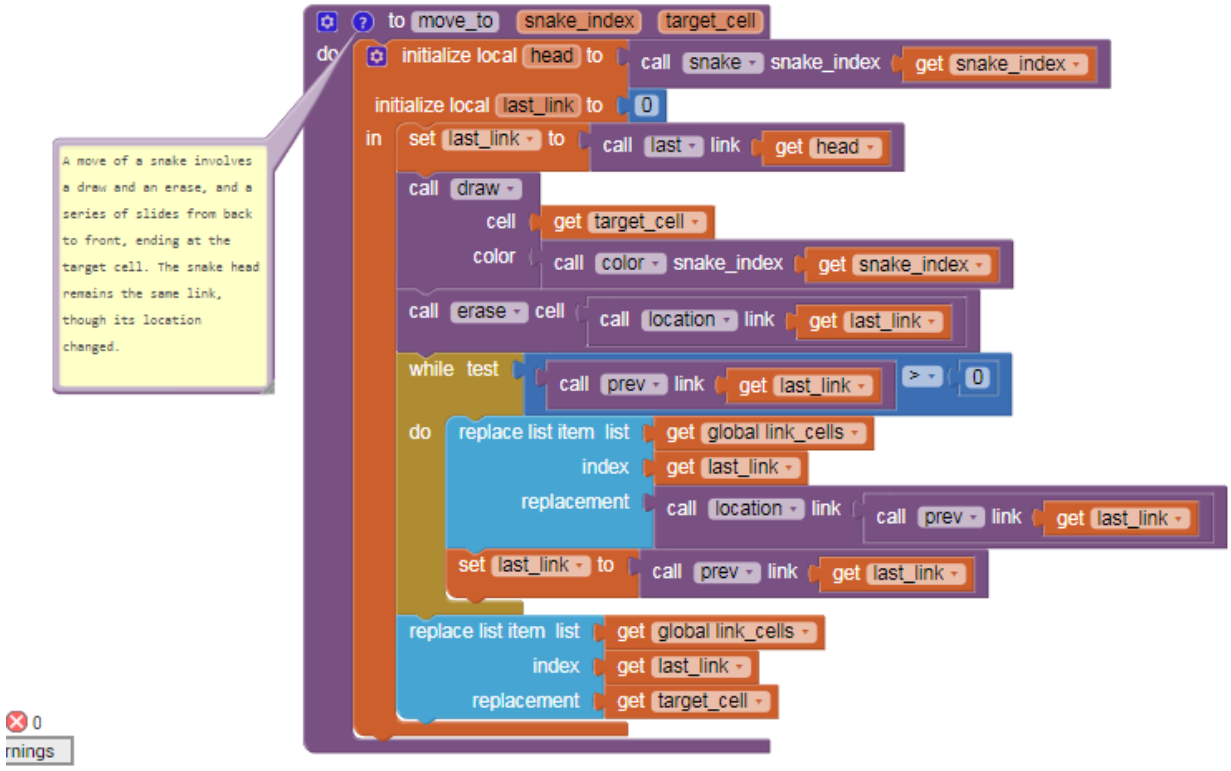
This procedure kills a snake, by removing it from the snakes list, and cleaning up any matching lists. It does no link manipulation; that's left to the calling attack routine.

last



This routine follows the follower chain of a link, returning the last link in the chain.

move_to



This is complex, simulating the movement of a caterpillar. First the drawing is handled at the new head of the snake, and at its old tail. Then we start a wave at the tail, lifting each link and dropping it onto the location of its predecessor, pushing the wave forward towards the head of

the snake, until we reach the head of the snake (prev = 0). All this movement happens in the list link_cells, which maps the link numbers into their cell locations.

Gallery link

http://ai2.appinventor.mit.edu/?galleryId=5592523749457920

Other Projects

https://docs.google.com/document/d/1acg2M5KdunKjJgM3Rxpy_Rf6vT6OozxdIWglgbmzroA/edit?usp=sharing