# John Conway's Game of Life
# A rather naive version written using the Zelle graphics Package.
# Bill Campbell

# So, our 2-D grid of cells is cells * cells in size, but actually
# we maintain a border of empty cells around this grid
# (making the counting of neighbors simpler; the border cells are
# always unpopulated). So the grid is actually (cells+2) * (cells+2).

# In this version we represent our 2-D grids as lists of lists;
# each row is represented as a nested list.
#
# For example, the 2-D
#  1 2 3
#  4 5 6
#  7 8 9
# would be represented as a list [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
#
# Of course, since we are representing our cells by 0's and 1's,
# 0 for unpopulated, 1 for populated, a completely populated
# grid
#  1 1 1
#  1 1 1
#  1 1 1
# would be represented as [[1, 1, 1], [1, 1, 1], [1, 1, 1]].
#
# Or, because we wrap a border of unpopulated cells around the grid
# (so that every cell on the grid proper has 8 neighbors) we have
# the grid
#  0 0 0 0 0
#  0 1 1 1 0
#  0 1 1 1 0
#  0 1 1 1 0
#  0 0 0 0 0
# represented as
# [[0,0,0,0,0],[0,1,1,1,0],[0,1,1,1,0],[0,1,1,1,0],[0,0,0,0,0]].
#
# (In this example, cells = 3. It is more in our simulation, eg 40.)

from graphics import *

cellSize = 10  # pixels

cells = 40      # along one side

gridSize = cellSize * cells  # along one side (in pixels)

nextGen = []    # the next generation is always computed
lastGen = []    # from the last generation

win = None      # The main window

def isMarked(x, y):
    """Is the cell (x, y) populated in the last generation?""
    return lastGen[x+1][y+1]

def mark(x, y):
    """Populate cell(x, y) in the next generation""
    rec = Rectangle(Point(x, y),
                     Point((x+1), (y+1)))
    rec.setFill("black")
def unmark(x, y):
    """Unpopulate cell (x, y) in the next generation."""
    rec = Rectangle(Point(x, y),
                    Point(x+1, y+1))
    rec.setFill("white")
    rec.draw(win)
    nextGen[x+1][y+1] = False

def neighbors(x, y):
    """The number of populated neighbors for cell (x, y)
    in the last generation.
    """
    def nbr(x, y):
        return lastGen[x+1][y+1]
    # Recall True is 1, False is 0
    return (nbr(x-1, y-1) +
            nbr(x, y-1) +
            nbr(x+1, y-1) +
            nbr(x-1, y) +
            nbr(x+1, y) +
            nbr(x-1, y+1) +
            nbr(x, y+1) +
            nbr(x+1, y+1))

def getCellClicked():
    """The (x, y) for the grid cell that was clicked on,
    where x is in the range 0..39 and y is in the range 0..39.
    Also notice that if y < 0, then we are clicking below the grid.
    """
    p = win.getMouse()
    x = int(p.getX())
    y = int(p.getY())
    print "(\"" + str(x) + ",\"" + str(y) + ")"
    return (x, y)

def nextGeneration():
    """Compute the next generation of grid cells from the last
generation, following the rules of life (and death):
1. A live cell in one generation
   a. dies (from overcrowding) if it has > 3 neighbors,
   b. dies (from isolation) if it has < 2 neighbors, and
   c. survives to the next generation otherwise.
2. An unpopulated cell in one generation
   a. is populated in the next generation if it has 3 neighbors,
   b. and remains unpopulated otherwise.
As we compute one generation from the last, we use mark() and
unmark() to populate cells in the next generation based
isMarked() for cells in the last. Notice only changes are drawn;
this minimizes painting on the window's canvas.
"
    global lastGen # so as not to be confused as defining a local var.
    # lastGen is a copy of nextGen (but has its own identity).
    lastGen = [row[:] for row in nextGen]
    for i in range(0, cells):
        for j in range(0, cells):
            n = neighbors(i, j)
            if isMarked(i, j):
                if n < 2:
                    unmark(i, j) # from isolation
                elif n > 3:
                    unmark(i, j) # from overcrowding
            else: # unmarked in last generation
                if n == 3:
                    mark(i, j)
def life():
    """ Run a simulation for John Conway's game of life.

    In this version, we paint the grid onto the window's canvas. Clicking on an unpopulated cell populates it; clicking on a populated cell unpopulates it. Once we one is satisfied with the initial configuration, clicking in the space below the grid on the window starts the game; successive clicks in this same region advance the game to successive generations.

    The only way to end (currently) is to close the window, which raises an error exception because the game is waiting on a mouse click on the window's canvas.
    """

    # Create window and draw the grid.
    global win
    global lastGen
    global nextGen  # So that these are not confused as local variables.
    win = GraphWin("John Conway's Game of Life", gridSize, gridSize+100)
    win.setCoords(0,-10,cells,cells)

    for i in range(0, cells+1):
        Line(Point(0, i), Point(cells, i)).draw(win)
        Line(Point(i, 0), Point(i, cells)).draw(win)

    # Initialize the cells (all unpopulated).
    # Notice here and only here, these are the same lists.
    lastGen = nextGen = [[0] * (cells+2) for i in range(cells+2)]

    # User clicks on cells to populate them.
    while True:
        x,y = getCellClicked()
        print("("+str(x)+","+str(y)+")")
        if y < 0:
            break  # clicked beneath the grid
        elif isMarked(x,y):
            unmark(x,y)
        else:
            mark(x,y)

    # Now, compute and show successive generations.
    while True:
        getCellClicked()  # Could animate this with time.sleep(1.0).
        nextGeneration()

    life()