Today in CS 42

HW9 (due Monday): Last Python

```python
class App:
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()
        self.button = Button(frame, text="QUIT", fg="red", command=frame.quit)
        self.button.pack(side=LEFT)
        self.hi_there = Button(frame, text="Hello", command=self.say_hi)
        self.hi_there.pack(side=LEFT)

    def say_hi(self):
        print "hi there, everyone!"
```

Python graphics with tkinter

Event-driven programming and the return of functions as first-class citizens
Designing a GUI in Python

```python
import Tkinter
tk = Tkinter
```

GUI layout + drawing the game board

Responding to user input

Game play
Designing a GUI in Python

GUI layout

Buttons

Canvas

Label
import Tkinter

tk = Tkinter

def main():
    ''' Start the game. '''
    root = tk.Tk()  # Create the main window

    app = Spampede(root)  # Create a Spampede object

    root.mainloop()  # Give the main window control
class Spampede(object):
    ''' A class that implements the game of Spampede using the tkinter library.''
    PADDING = 5    # The space between the edge of the canvas and the board
    WIDTH = 800     # The width of the gameboard
    HEIGHT = 600    # The height of the gameboard
    # Game colors
    BGCOLOR = "white"
def __init__(self, master):  # master is the main window
    ''' Initialize the Spampede application '''
    frame = tk.Frame(master)  # Create the main frame, inside master
    frame.pack()  # Pack it into the master window

    buttonFrame = tk.Frame(frame)  # Create a frame for buttons
    buttonFrame.pack(side=tk.TOP)  # Put that frame at the top of
    # the main frame

    startb = tk.Button(buttonFrame, text="Start", command=self._go)
    startb.pack(side=tk.LEFT)
    pauseb = tk.Button(buttonFrame, text="Pause", command=self._pause)
    pauseb.pack(side=tk.LEFT)
    resetb = tk.Button(buttonFrame, text="New Game", command=self._reset)
    resetb.pack(side=tk.LEFT)

    self.canvas = tk.Canvas(frame, width=self.WIDTH, height=self.HEIGHT,
                            bg=self.BGCOLOR)
    self.canvas.pack(fill=tk.BOTH)

    self.message = "Welcome to Spampede v0.42!"
    self.messageLabel = tk.Label(frame, text=self.message)
    self.messageLabel.pack(side=tk.BOTTOM)
import Tkinter

tk = Tkinter

# More code here not shown...
# then, in __init__:

buttonFrame = tk.Frame(frame)  # Create a frame for buttons

buttonFrame.pack(side=tk.TOP)  # Put that frame at the top of the main frame

startb = tk.Button(buttonFrame, text="Start", command=self._go)

startb.pack(side=tk.LEFT)

Create a new Frame object to hold the buttons

Create a new Button to be the start button and set some of its attributes
Attributes of graphical objects

import Tkinter

tk = Tkinter

# More code here not shown...
# then, in __init__:

buttonFrame = tk.Frame(frame)  # Create a frame for buttons
buttonFrame.pack(side=tk.TOP)  # Put that frame at the top of
    # the main frame
startb = tk.Button(buttonFrame, text="Start", command=self._go)

But how do you know what you can specify here?
Answer: Look it up.
http://effbot.org/tkinterbook/button.htm

A widget's parent is always the first argument

startb.pack(side=tk.LEFT)

you can also change attributes later using config, e.g.

self.messageLabel.config(text=myNewText)
The pack geometry manager

```python
import tkinter

tk = tkinter

# More code here not shown...
# then, in __init__:

buttonFrame = tk.Frame(frame)  # Create a frame for buttons
buttonFrame.pack(side=tk.TOP)  # Put that frame at the top of
# the main frame

pack tells tkinter to show the item in the frame.
Optional arguments: side (values: TOP, BOTTOM, LEFT, RIGHT, CENTER)
    fill (values: X, Y, BOTH) – see canvas for example

startb = tk.Button(buttonFrame, text="Start", command=self._go)
startb.pack(side=tk.LEFT)
```

There are other geometry managers (e.g., grid), but pack is the simplest.

http://effbot.org/tkinterbook/pack.htm
The **Canvas** hosts the game

# Create the canvas object and save it. This is where you will
# draw all of the components of the gameboard.
self.canvas = tk.Canvas(frame, width=self.WIDTH,
                        height=self.HEIGHT, bg=self.BGCOLOR)

self.canvas.pack(fill=tk.BOTH)

Try it! Download the Spampede starter file
and play around with it. Change attributes
of objects, add buttons, etc. Or create your
own new GUI from scratch.
Game control overview

Application run

_\texttt{\_init\_}() \rightarrow \texttt{\_reset()} \rightarrow \texttt{\_cycle()} \rightarrow \texttt{\_keyPressed()}(*)

1. \texttt{\_updateCentipede()}\n
2. \texttt{\_updateSpam()}\n
3. \texttt{\_drawEnvironment()}\n
4. \texttt{\_displayMessage()}\n
5. \texttt{\_cycleNum+=1}\n
6. \texttt{after...}
def _reset(self):
    ''' This is the function that will run when the user
    presses the New Game button. Resets the game.'''
    # set up the maze here
    self.theMaze = SpamMaze()

    # You may also want to initialize the pede's direction here
    # and to other initializations that need to happen each time
    # a new game is started.

    self.currentColor = "green"
    self._drawEnvironment()
    self._displayMessage()

Application run
__init__() → _reset() → cycle() → keyPressed()

1. _updateCentipede()
2. _updateSpam()
3. _drawEnvironment()
4. _displayMessage()
5. _cycleNum+=1
6. after...
def _cycle(self):
    ''' This is the main "loop" that controls the gameplay. '''
    if self.running:
        self._updateCentipede()  # update the Spampede deque
        self._updateSpam()        # update the Spam deque
        self._drawEnvironment()   # draw things to buffer
        self._displayMessage()    # display messages
        self.cycleNum += 1        # add a cycle
    # Wait a little while and then cycle again.
    self.canvas.after(self.sleepTime, self._cycle)
# You might use this method to move the centipede one square
# Also, this method can check if the centipede runs
# into a can of spam, a wall, itself, etc. and act appropriately.
def _updateCentipede(self):
    ''' Update the centipede. You don't have to do this
    every cycle if you want a slower game. '''
    return

# You might use this method to add/delete spam cans periodically
def _updateSpam(self):
    ''' Possibly add or remove spam. You won't want to do this
    every cycle. '''
    return

Application run

__init__() → _reset() → cycle() → _keyPressed()

1. _updateCentipede()
2. _updateSpam()
3. _drawEnvironment()
4. _displayMessage()
5. _cycleNum+=1
6. after...
Game control details

```python
def _drawEnvironment(self):
    ''' Draw the state of the game on the board.
       Note that (0,0) is in the top left corner of the board
       and +y is down.'''
    self._clear()
    r3 = self.canvas.create_rectangle(40,50,100,100)
    self.canvas.itemconfig(r3, fill="red")
    r2 = self.canvas.create_rectangle(250, 100, 350, 200, fill=self.currentColor)
    if (self.cycleNum+1) % 42 == 0:
        self.currentColor = "magenta"
```

Application run:

- `__init__()` → `__reset()` → `cycle()`
  - 1. `_updateCentipede()`
  - 2. `_updateSpam()`
  - 3. `_drawEnvironment()`
  - 4. `_displayMessage()`
  - 5. `_cycleNum+=1`
  - 6. `after…`

Issues?
Double Buffering

individual drawing commands

Off screen buffer

raster copy

Screen

fillRect ...

image
possibly slow to draw...

avoiding flicker...

screen
but fast to copy
Double Buffering

Avoiding flicker...

Individual drawing commands:
- fillRect...

Off screen buffer:
- raster copy

Screen:
- image
- possibly slow to draw...

Python:
- automatically updates all at once

_Gustav Verbeek_

_drawEnvironment()_ 
_displayMessage()_
def _displayMessage(self):
    """ Display a text message to the user.'""
    self.messageLabel.config(text=self.message)
import Tkinter
tk = Tkinter

def main():
    ''' Start the game. '''
    root = tk.Tk()  # Create the main window

    app = Spampede(root)  # Create a Spampede object

    root.mainloop()  # Give the main window control

Application run

___init__() ➔ __reset__ ➔ cycle() ➔ __displayMessage() ➔ __drawEnvironment() ➔ __updateSpam() ➔ __updateCentipede() ➔ keyPressed() ➔ after... ➔ __cycleNum+=1
Event-driven programming

```python
import Tkinter
tk = Tkinter

def main():
    ''' Start the game. '''
    root = tk.Tk()
    app = Spampede(root)
    root.mainloop()
```
import Tkinter
tk = Tkinter

def main():
    ''' Start the game. '''
    root = tk.Tk()
    app = Spampede(root)
    root.mainloop()

The program is watching a queue waiting for an “event” to show up.

Events may be processed internally, but you can also ask the system to call a specific function (a “callback” function)
Event-driven programming

```python
buttonFrame.pack(side=tk.TOP)  # Put that frame at the top of
# the main frame

startb = tk.Button(buttonFrame, text="Start", command=self._go)
```

Passing the function in as an argument will register it to be called whenever a click event happens on this button.
Event-driven programming

startb = tk.Button(buttonFrame, text="Start", command=self._go)
startb.pack(side=tk.LEFT)

pauseb = tk.Button(buttonFrame, text="Pause", command=self._pause)
pauseb.pack(side=tk.LEFT)

resetb = tk.Button(buttonFrame, text="New Game", command=self._reset)
resetb.pack(side=tk.LEFT)

self.canvas = tk.Canvas(frame, width=self.WIDTH, height=self.HEIGHT,
                        bg=self.BGCOLOR)
self.canvas.pack(fill=tk.BOTH)

self.message = "Welcome to Spampede v0.42!"
self.messageLabel = tk.Label(frame, text=self.message)
self.messageLabel.pack(side=tk.BOTTOM)

# Capture all keystroke events. Handle them with the self._keyPressed
# function, which you will need to add to.
master.bind_all("<Key>", self._keyPressed)
def _go(self):
    ''' This is the function that will run when the user presses the Start button. (Re)Starts the game.'''
    self.running = True
    self._cycle()

def _pause(self):
    ''' This is the function that will run when the user presses the Pause button. Pauses the game.'''
    self.running = False
def _cycle(self):
    ''' This is the main "loop" that controls the gameplay. '''
    while self.running:
        self._updateCentipede()   # update the Spampede deque
        self._updateSpam()        # update the Spam deque
        self._drawEnvironment()   # draw things to buffer
        self._displayMessage()    # display messages
        self.cycleNum += 1        # add a cycle

Assuming all helpers were written correctly, what would this code do?
A. Correctly play the game
B. Move the pede but not allow the user to control it
C. Move the pede and allow the user to control it, but not allow
   the user to stop the game
D. Cause the main window to "hang"
def _cycle(self):
    ''' This is the main "loop" that controls the gameplay. '''
    if self.running:
        self._updateCentipede()  # update the Spampede deque
        self._updateSpam()       # update the Spam deque
        self._drawEnvironment()  # draw things to buffer
        self._displayMessage()   # display messages
        self.cycleNum += 1       # add a cycle
        # Wait a little while and then cycle again.
        self.canvas.after(self.sleepTime, self._cycle)

_after_ schedules an event to occur at least self.sleepTime milliseconds in the future.
IDLE and TkInter

```python
import Tkinter
tk = Tkinter

def main():
    ''' Start the game. '''
    root = tk.Tk()           # Create the main window

    app = Spampede(root)     # Create a Spampede object

    root.mainloop()          # Give the main window control

IDLE is written in TkInter! So why does this cause a problem…?
1) Where is the code that is drawing the small red square?

2) What do the first four arguments to \texttt{create_rectangle(int,int,int,int)} mean?

3) What is the point of the \texttt{PADDING} member variable?

4) What keypress(es) lead to drawing the large square dark \texttt{blue}?
   Which keypress is \textit{incorrectly} reported on the screen?

5) What code draws the text of the latest keypress onto the screen?

6) What's your favorite color?

7) How long does it take for the large square to return to \texttt{magenta} color?

8) What happens in the code when "Pause" and "Start" are pressed?
   Where/how does \texttt{sleepTime} determine the speed of the application's updates?
1) Do the Maze and SpamMaze parts of the assignment first...

2) Make sure you can change, and test Spampede.py

3) Get a bird’s-eye view of the code...
   self.theMaze
   self.dir the current direction
   the snake is heading
   (doesn't exist yet)

4) Write _drawEnvironment (and test...) one direction at a time...

5) Write _updatePede and _keyPressed (and test...) one direction at a time...

6) Build up to reversing and AI (note that AI is almost completely implemented by multiBFS)

EXTRA FEATURES!!