Make a Snake game for Android written in Python – Part 2

FEBRUARY 4, 2015 / 2 COMMENTS

If you followed Part 1 of this tutorial, hopefully your development environment should be all set. We're ready to get down to business (in France we would say : "mettre les mains dans le cambouis". Because you can learn coding, and useless french idioms at the same time!)

The objective

In this part of the tutorial, we will make the game engine of our snake. By making the game engine, I mean :

s

Writing the classes corresponding to the skeleton of our app.

Giving them the proper methods and properties so that we can control their behavior as we wish.

Put everything in the main loop of our app and see how it goes (spoiler alert : it ought to go well since I tested the code every step of the way).

For every section, I will start by explicitly explain what we are doing then present the code and finally link to the corresponding version of the repository.

The classes

What's a snake game if you decompose its elements? Well for starter : a playground and a snake. Oh, and don't forget the fruit that pops from time to time! The snake in itself is composed of two main elements : a head, and a tail.

Thus, we will need to implement the following widgets hierarchy :

- Playground
  - Fruit
  - Snake
    - SnakeHead
    - SnakeTail

We are going to declare our classes in the python file of our application, and if need be in a .kv file in order to separate the front-end from the back-end logic and to make use of the automated binding system.

main.py

```
import kivy
```

```
kivy.require('1.8.0')  # update with your current version

# import the kivy elements used by our classes
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.properties import ObjectProperty


class Playground(Widget):
    # children widgets containers
    fruit = ObjectProperty(None)
    snake = ObjectProperty(None)


class Fruit(Widget):
    pass


class Snake(Widget):
    # children widgets containers
    head = ObjectProperty(None)
    tail = ObjectProperty(None)


class SnakeHead(Widget):
    pass


class SnakeTail(Widget):
    pass


class SnakeApp(App):

    def build(self):
        game = Playground()
        return game

if __name__ == '__main__':
    SnakeApp().run()
```

snake.kv

```
#:kivy 1.8.0

<Playground>;
    snake: snake_id
    fruit: fruit_id

    Snake:
        id: snake_id

    Fruit:
        id: fruit_id

<Snake>;
    head: snake_head_id
    tail: snake_tail_id

    SnakeHead:
        id: snake_head_id

    SnakeTail:
        id: snake_tail_id
```

Full code.

Properties

Now that we have our classes set, we can start to think about their content. To do that, we are going to define a few things.

The Playground is the root Widget. We will divide its canvas as a grid, setting the number of rows and columns as properties. This matrix will help us to position and navigate our snake. Every child widget's representation on the canvas will occupy the size 1 cell. We also need to store the score and the rhythm at which the fruit will pop (more on this later), as well as a turn counter that will be used to know when to pop the fruit and when to remove it.

Last but not least, we also need to manage input. I'll explain more precisely how in the next section. For now, we're just going to accept that we need to store the start position when the on_touch_down event is triggered and a boolean variable stating if an action was triggered by the current pattern of input.

```
class Playground(Widget):
    # children widgets containers
    fruit = ObjectProperty(None)
    snake = ObjectProperty(None)

    # grid parameters (chosent to respect the 16/9 format)
    col_number = 16
    row_number = 9

    # game variables
    score = NumericProperty(0)
    turn_counter = NumericProperty(0)
    fruit_rythme = NumericProperty(0)

    # user input handling
    touch_start_pos = ListProperty()
    action_triggered = BooleanProperty(False)
```

Note : don't forget to import the kivy Properties we add along the way.

The snake now : the Snake object in itself doesn't need to store anything else than its two children (head and tail). It will only serve as in interface so that we don't interact directly with its components.

The SnakeHead however is a different matter. We want to store its position in the grid. We also need to know which direction it is currently set to, to choose the right graphical representation as well as to navigate the snake between turns (if the direction is left, draw a left-pointing triangle on the [x-1, y] cell etc.).

Position + direction will correspond to a certain set of drawing instructions. To draw a triangle, we need to store 6 points of coordinates : (x0, y0), (x1, y1), (x2, y2). These coordinates are no more cells of the grid as the position was : they are the corresponding pixels values on the canvas.

Finally, we'll have to store the object drawn to the canvas in order to remove it later (for a game reset per example). So that we're super safe, we'll add a boolean variable indicating if indeed the object is drawn on the canvas (this way if we ask for the object to be removed wrongfully and the object was never actually drawn, nothing will happen. As opposed to our app crashing).

```
class SnakeHead(Widget):
    # representation on the "grid" of the Playground
    direction = OptionProperty(
        "Right", options=["Up", "Down", "Left", "Right"])
    x_position = NumericProperty(0)
    y_position = NumericProperty(0)
    position = ReferenceListProperty(x_position, y_position)

    # representation on the canvas
    points = ListProperty([0]*6)
    object_on_board = ObjectProperty(None)
    state = BooleanProperty(False)
```

Now for the tail. It is composed of blocks, each occupying one cell and corresponding to the positions occupied by the head during the past turns. Thus, we need to define the size of the tail which will be set by default as 3 blocks. Moreover, we'll want to store the positions of its constituent blocks, and the corresponding objects drawn on the canvas so that we can update them during each turn (ie : remove the last tail block and add a new one where the head was so that the tail moves with the head).

```
class SnakeTail(Widget):
    # tail length, in number of blocks
    size = NumericProperty(3)

    # blocks positions on the Playground's grid
    blocks_positions = ListProperty()

    # blocks objects drawn on the canvas
    tail_blocks_objects = ListProperty()
```

Finally the fruit. Its graphical behavior is similar to the head, so we'll need a state variable and a property storing the object drawn. The fruit will pop from time to time, so we have to define the number of turns during which it will stay on board (duration) and the interval between the appearances. These two values will be used to compute the fruit_rhythm in the Playground class (remember, I said we would get back to that).

```
class Fruit(Widget):
    # constants used to compute the fruit_rhythme
    # the values express a number of turns
    duration = NumericProperty(10)
    interval = NumericProperty(3)

    # representation on the canvas
    object_on_board = ObjectProperty(None)
    state = BooleanProperty(False)
```

The App class also needs a quick modification. We are going to pass the Playground as a property so that we can continue to interact with it after build() is called. I'll explain why in the next section.

```
class SnakeApp(App):
    game_engine = ObjectProperty(None)

    def build(self):
        self.game_engine = Playground()
        return self.game_engine
```

One more thing : do we have anything to add into the .kv file ? Well yes we do Barry, yes we do. We need to set the dimensions of our widgets. Remember our imaginary grid ? We'll use that to compute the width and the height of each widget. Whereas it is the fruit or the snake, the formula is the same :

width = playground width / number of columns

height = playground height / number of rows

The Snake will then pass on these values to its children. Oh and since we're at it, let's add a Label on the Playground to display the score.

```
#:kivy 1.8.0

<Playground>
    snake: snake_id
    fruit: fruit_id

    Snake:
        id: snake_id
        width: root.width/root.col_number
        height: root.height/root.row_number

    Fruit:
        id: fruit_id
        width: root.width/root.col_number
        height: root.height/root.row_number

    Label:
        font_size: 70
        center_x: root.x + root.width/root.col_number*2
        top: root.top - root.height/root.row_number
        text: str(root.score)

<Snake>
    head: snake_head_id
    tail: snake_tail_id

    SnakeHead:
        id: snake_head_id
        width: root.width
        height: root.height

    SnakeTail:
        id: snake_tail_id
        width: root.width
        height: root.height
```

Full code

Methods

Let's start with the Snake class. We want to be able to set its starting position and direction, and to make it move accordingly. The counterpart would also be good : get the current position (to check if the player lost because the snake is outbound, per example), same thing regarding the direction. We also need to be able to instruct the snake to remove its representation from the canvas. Behind the scenes, the Snake will dispatch the right instructions to its components. We wouldn't want to manually remove its children every time we want the snake gone. Its kids, its responsibility !

```python
class Snake(Widget):
...
    def move(self):
        """
        Moving the snake involves 3 steps :
            - save the current head position, since it will be used to add a
            block to the tail.
            - move the head one cell in the current direction.
            - add the new tail block to the tail.
        """
        next_tail_pos = list(self.head.position)
        self.head.move()
        self.tail.add_block(next_tail_pos)

    def remove(self):
        """
        With our current snake, removing the whole thing sums up to remove its
        head and tail, so we just have to call the corresponding methods. How
        they deal with it is their problem, not the Snake's. It just passes
        down the command.
        """
        self.head.remove()
        self.tail.remove()

    def set_position(self, position):
        self.head.position = position

    def get_position(self):
        """
        We consider the Snake's position as the position occupied by the head.
        """
        return self.head.position

    def get_full_position(self):
        """
        But sometimes we'll want to know the whole set of cells occupied by
         the snake.
        """
        return self.head.position + self.tail.blocks_positions

    def set_direction(self, direction):
        self.head.direction = direction

    def get_direction(self):
        return self.head.direction
```

We called a number of methods involving the head and the tail but didn't create them yet. For the SnakeTail, we want remove() and add_block().

```python
class SnakeTail(Widget):
...

    def remove(self):
        # reset the size if some fruits were eaten
        self.size = 3

        # remove every block of the tail from the canvas
        # this is why we don't need a is_on_board() here :
        # if a block is not on board, it's not on the list
        # thus we can't try to delete an object not already
        # drawn
        for block in self.tail_blocks_objects:
            self.canvas.remove(block)

        # empty the lists containing the blocks coordinates
        # and representations on the canvas
        self.blocks_positions = []
        self.tail_blocks_objects = []

    def add_block(self, pos):
        """
        3 things happen here :
            - the new block position passed as argument is appended to the
            object's list.
            - the list's number of elements is adapted if need be by poping
            the oldest block.
            - the blocks are drawn on the canvas, and the same process as before
            happens so that our list of block objects keeps a constant size.
        """
        # add new block position to the list
        self.blocks_positions.append(pos)

        # control number of blocks in the list
        if len(self.blocks_positions) > self.size:
            self.blocks_positions.pop(0)

        with self.canvas:
            # draw blocks according to the positions stored in the list
            for block_pos in self.blocks_positions:
                x = (block_pos[0] - 1) * self.width
                y = (block_pos[1] - 1) * self.height
                coord = (x, y)
                block = Rectangle(pos=coord, size=(self.width, self.height))

                # add new block object to the list
                self.tail_blocks_objects.append(block)

                # control number of blocks in list and remove from the canvas
                # if necessary
                if len(self.tail_blocks_objects) > self.size:
                    last_block = self.tail_blocks_objects.pop(0)
                    self.canvas.remove(last_block)
```

For the head, move() and remove(). The former will implicate two steps : changing the position according to the direction (+1 cell up, or down, or…), and rendering a Triangle at this new position. We also want to check on remove if the object we're removing is indeed on board (remember the state variable we created for that purpose ?).

```
class SnakeHead(Widget):
    # representation on the "grid" of the Playground
    direction = OptionProperty(
        "Right", options=["Up", "Down", "Left", "Right"])
    x_position = NumericProperty(0)
    y_position = NumericProperty(0)
    position = ReferenceListProperty(x_position, y_position)

    # representation on the canvas
    points = ListProperty([0] * 6)
    object_on_board = ObjectProperty(None)
    state = BooleanProperty(False)

    def is_on_board(self):
        return self.state

    def remove(self):
        if self.is_on_board():
            self.canvas.remove(self.object_on_board)
            self.object_on_board = ObjectProperty(None)
            self.state = False

    def show(self):
        """
        Actual rendering of the snake's head. The representation is simply a
        Triangle oriented according to the direction of the object.
        """
        with self.canvas:
            if not self.is_on_board():
                self.object_on_board = Triangle(points=self.points)
                self.state = True  # object is on board
            else:
                # if object is already on board, remove old representation
                # before drawing a new one
                self.canvas.remove(self.object_on_board)
                self.object_on_board = Triangle(points=self.points)

    def move(self):
        """
        Let's agree that this solution is not very elegant. But it works.
        The position is updated according to the current direction. A set of
        points representing a Triangle turned toward the object's direction is
        computed and stored as property.
        The show() method is then called to render the Triangle.
        """
        if self.direction == "Right":
            # updating the position
            self.position[0] += 1

            # computing the set of points
            x0 = self.position[0] * self.width
            y0 = (self.position[1] - 0.5) * self.height
            x1 = x0 - self.width
            y1 = y0 + self.height / 2
            x2 = x0 - self.width
            y2 = y0 - self.height / 2
        elif self.direction == "Left":
            self.position[0] -= 1
            x0 = (self.position[0] - 1) * self.width
            y0 = (self.position[1] - 0.5) * self.height
            x1 = x0 + self.width
            y1 = y0 - self.height / 2
            x2 = x0 + self.width
            y2 = y0 + self.height / 2
        elif self.direction == "Up":
            self.position[1] += 1
            x0 = (self.position[0] - 0.5) * self.width
            y0 = self.position[1] * self.height
            x1 = x0 - self.width / 2
            y1 = y0 - self.height
            x2 = x0 + self.width / 2
            y2 = y0 - self.height
        elif self.direction == "Down":
            self.position[1] -= 1
            x0 = (self.position[0] - 0.5) * self.width
            y0 = (self.position[1] - 1) * self.height
            x1 = x0 + self.width / 2
            y1 = y0 + self.height
            x2 = x0 - self.width / 2
            y2 = y0 + self.height

        # storing the points as property
        self.points = [x0, y0, x1, y1, x2, y2]

        # rendering the Triangle
        self.show()
```

What about the fruit ? We need to be able to make it pop on given coordinates, and to remove it. The syntax should start to become familiar by now.

```
class Fruit(Widget):
...

    def is_on_board(self):
        return self.state

    def remove(self, *args):
        # we accept *args because this method will be passed to an
        # event dispatcher so it will receive a dt argument.
        if self.is_on_board():
            self.canvas.remove(self.object_on_board)
            self.object_on_board = ObjectProperty(None)
            self.state = False

    def pop(self, pos):
        self.pos = pos  # used to check if the fruit is begin eaten

        # drawing the fruit
        # (which is just a circle btw, so I guess it's an apple)
        with self.canvas:
            x = (pos[0] - 1) * self.size[0]
            y = (pos[1] - 1) * self.size[1]
            coord = (x, y)

            # storing the representation and update the state of the object
            self.object_on_board = Ellipse(pos=coord, size=self.size)
            self.state = True
```

We’re almost there, don’t give up ! We need to add control for the whole game now, which will take place in the Playground class. Let’s review the logic of the game : it starts, a new snake is added on random coordinates, the game is updated to go to the next turn. We check for a possible defeat. For now, a defeat happens if the snake’s head collides with its own tail, or if it exits the screen. In case of defeat, the game is reset.

How will we handle the user’s input ? When the screen is touched, the position of the touch is stored. When the user moves its finger across the screen, the successive positions are compared to the starting position. If the move corresponds to a translation equal to 10% of the screen’s size, we consider it as an instruction and check in which direction the translation was made. We set the snake’s direction accordingly.

```
class Playground(Widget):
  ...

   def start(self):
       # draw new snake on board
       self.new_snake()

       # start update loop
       self.update()

   def reset(self):
       # reset game variables
       self.turn_counter = 0
       self.score = 0

```

```
        # remove the snake widget and the fruit if need be; its remove method
        # will make sure that nothing bad happens anyway
        self.snake.remove()
        self.fruit.remove()

    def new_snake(self):
        # generate random coordinates
        start_coord = (
            randint(2, self.col_number - 2), randint(2, self.row_number - 2))

        # set random coordinates as starting position for the snake
        self.snake.set_position(start_coord)

        # generate random direction
        rand_index = randint(0, 3)
        start_direction = ["Up", "Down", "Left", "Right"][rand_index]

        # set random direction as starting direction for the snake
        self.snake.set_direction(start_direction)

    def pop_fruit(self, *args):
        # get random coordinates for the fruit
        random_coord = [
            randint(1, self.col_number), randint(1, self.row_number)]

        # get all cells positions occupied by the snake
        snake_space = self.snake.get_full_position()

        # if the coordinates are on a cell occupied by the snake, re-draw
        while random_coord in snake_space:
            random_coord = [
                randint(1, self.col_number), randint(1, self.row_number)]

        # pop fruit widget on the coordinates generated
        self.fruit.pop(random_coord)

    def is_defeated(self):
        """
        Used to check if the current snake position corresponds to a defeat.
        """
        snake_position = self.snake.get_position()

        # if the snake bites its own tail : defeat
        if snake_position in self.snake.tail.blocks_positions:
            return True

        # if the snake it out of the board : defeat
        if snake_position[0] > self.col_number \
                or snake_position[0] < 1 \
                or snake_position[1] > self.row_number \
                or snake_position[1] < 1:
            return True

        return False

    def update(self, *args):
        """
        Used to make the game progress to a new turn.
        """
        # move snake to its next position
        self.snake.move()

        # check for defeat
        # if it happens to be the case, reset and restart game
        if self.is_defeated():
            self.reset()
            self.start()
            return

        # check if the fruit is being eaten
        if self.fruit.is_on_board():
            # if so, remove the fruit, increment score and tail size
            if self.snake.get_position() == self.fruit.pos:
                self.fruit.remove()
                self.score += 1
                self.snake.tail.size += 1

        # increment turn counter
        self.turn_counter += 1

    def on_touch_down(self, touch):
        self.touch_start_pos = touch.spos

    def on_touch_move(self, touch):
        # compute the translation from the start position
        # to the current position
        delta = Vector(*touch.spos) - Vector(*self.touch_start_pos)


        # check if a command wasn't already sent and if the translation
        # is > to 10% of the screen's size
        if not self.action_triggered \
            and (abs(delta[0]) > 0.1 or abs(delta[1]) > 0.1):
            # if so, set the appropriate direction to the snake
            if abs(delta[0]) > abs(delta[1]):
                if delta[0] > 0:
                    self.snake.set_direction("Right")
                else:
                    self.snake.set_direction("Left")
            else:
                if delta[1] > 0:
                    self.snake.set_direction("Up")
                else:
                    self.snake.set_direction("Down")
            # register that an action was triggered so that
            # it doesn't happen twice during the same turn
            self.action_triggered = True

    def on_touch_up(self, touch):
        # we're ready to accept a new instruction
        self.action_triggered = False
```

Full code.

Congratulations, if you're still reading you've passed the hardest part of the tutorial. Try to run it : nothing happens but we have our playground and the score, which is a good start. All our methods are ready. We just need to schedule them in the main loop.

The main loop

The update method of the Playground is the key here. It will handle the event scheduling for the fruit, and reschedule itself after each turn. This peculiar behavior is implemented so that we avoid any unintended update loop, and will be useful in the next part of the tutorial when we add some options to the game (like an increasing update rhythm). For now a turn will last one second.

```
    def update(self, *args):
        """
        Used to make the game progress to a new turn.
        """
        # registering the fruit poping sequence in the event scheduler
        if self.turn_counter == 0:
            self.fruit_rythme = self.fruit.interval + self.fruit.duration
            Clock.schedule_interval(
                self.fruit.remove, self.fruit_rythme)
        elif self.turn_counter == self.fruit.interval:
            self.pop_fruit()
            Clock.schedule_interval(
                self.pop_fruit, self.fruit_rythme)
...
        # schedule next update event in one turn (1'')
        Clock.schedule_once(self.update, 1)
```

Let's not forget to unscheduled all events in case of a reset. By the way, you did import the Clock, right ? 😉

```
    def reset(self):
...
        # unschedule all events (they will be properly rescheduled by the
        # restart mechanism)
        Clock.unschedule(self.pop_fruit)
        Clock.unschedule(self.fruit.remove)
        Clock.unschedule(self.update)
```

You're almost ready to play your own snake ! Are you excited ? I'm excited (well I was the first time). Phrasing!

Anyhow. Recall that we made the Playground instance a property of our main App. Why is that ? Because we need to start the game when the App in itself starts, and not when build() is called. Otherwise the sizes we set in the .kv file would be initialized at their default values (100,100). That's not what we want. We want the proper size of the screen. Here we go :

```
class SnakeApp(App):
    game_engine = ObjectProperty(None)

    def on_start(self):
        self.game_engine.start()
```

```
...
```

You can run your App now. Et voilà ! You can package it with buildozer if you want to give it a try on your phone, or wait for the next part of the tutorial that we add a nice welcome screen with some options.

Full code

Share this:

Facebook Google Twitter LinkedIn Pinterest Reddit Email

Categories: Programming Tags: android, kivy, Python, tutorial

« Make a Snake game for Android written in Python – Part 1

Make a Snake game for Android written in Python – Part 3 »

2 Comments

Lenny
FEBRUARY 9, 2015 AT 8:48 AM

Hey nice work on the tutorial , this will certanly be helpfull. You do have a typo in SnakeTail line 47, instead of 'self.tail_blocks.append' you need 'self.tail_blocks_objects.append'

REPLY

alexis.matelin (Post author)
FEBRUARY 9, 2015 AT 6:17 PM

Yes I do thank you very much for spotting that ! Correction made 😉

REPLY

Leave a Reply