Practicum 2: Tetris

1 Tetris

For this practicum, you will implement Tetris in Python. Tetris is a computer game where the player needs to arrange the pieces that fall from above to form full lines and thus score points. When a line is complete, the line vanishes and thus makes more space on the playing field. However, when the playing field is filled up with pieces and the pile reaches the top, the game is lost. If you do not know the game or need more background, please consult the Web and Wikipedia: `http://en.wikipedia.org/wiki/Tetris`. You can see the game in action on: `http://www.freetetris.org/` and `http://www.tetrisfriends.com/`.

You need to implement the basic version of the Tetris game which requires the standard core gameplay but does not require scoring, levels, menu's or highscore lists. Below follows a more detailed description of the basic version you need to implement.

- Your implementation of the game must be as close as possible to the original game, unless mentioned otherwise.
- You can make the game start right away when your code is executed, so you do not need to create a game menu.
- There are 7 possible pieces, as shown in Figure 1. Each *piece* consists of 4 *blocks*, arranged in a structure.
- At all times, there is only one piece that is falling down. This piece is the current piece. The piece is chosen at random.
- While the current piece is landing, the player can rotate it (e.g. with the Up Arrow key), move it left (Left Arrow) or right (Right Arrow) and drop it down (e.g. Down Arrow). The dropping can be implemented either as moving the piece one block down per key press or dropping all the way down at once. The choice is up to you, whichever you find easier.
- Use a playing field with dimensions 20x10 blocks (height x width).
- You only need to draw the main playing field (where the pieces are moving). You do not need to show the next piece. You also do not need to implement scoring, different levels, so you do not need to display the score nor the current level to the user or change the speed of the game.

- When a row (horizontal line of blocks) is complete (contains no empty spaces), the complete row must be removed. When a row is removed, all the blocks above this row are moved down with one block.

- Make sure pieces can not be in illegal positions. The pieces must never exceed the boundaries of the playing field (except at the top) nor can they overlap previously landed pieces. You do not need to implement a "wall kick" or a "floor kick". Wall kick normally occurs when the falling piece is rotated in such a way that it would exceed the boundaries of the playing field. You can detect such rotations and "kick" the piece off the wall (or floor), that is, move it away such that the rotated piece fits in the playing field. However, if it is easier for you, you can also ignore commands that would bring the piece in illegal positions. In any case, it is not allowed to have a piece in an illegal position at any time.

- Do not forget to detect when the game is lost, that is, when the pieces pile up higher than the top of the playing field. You can simply terminate the program execution (`sys.exit()`) in this case.

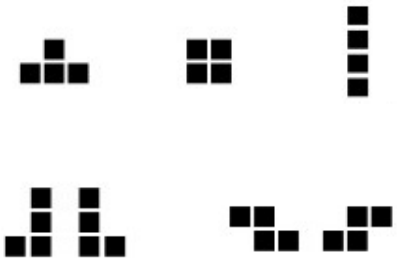

Figure 1: The different pieces and their shapes in the standard Tetris.

2 Implementing Tetris with Tkinter

You can use `tkinter` to implement Tetris. The `tkinter` package in Python provides an easy way to draw on a canvas (very similar to the `graphics` package you have been using so far) but it also provides an easy way to handle user interaction (key presses, mouse movements, etc.). A tutorial on `tkinter` can be found here: `http://effbot.org/tkinterbook/`. The official documentation is here: `https://docs.python.org/3.4/library/tkinter.html`. There is no need to install `tkinter` as it normally comes with Python. You do not need to learn `tkinter`. We will point you towards the relevant resources in further explanation but you do not need to know how it all works. Your goal is to write a working Tetris game.

At the end of this document, a piece of code is provided which you can use as a starting point to implement the game. It also shows how to use various relevant `tkinter` objects.

However, the code is written in such a way that you do not need to change much `tkinter`-related code so that you can start writing the actual game logic right away.

When implementing the game, try to separate your code in three parts: (1) game representation, (2) drawing and (3) game control. The **representation** part of your code should handle the internal representation of the state of the game, in our case which blocks are where, what the current piece is, etc. The **drawing** part of your code takes care of rendering the game, that is, it implements the logic to display the model to the user. The **control** part of your code handles user interaction and other events, in our case, the detection and handling of keyboard events to control the blocks, the choice of the next block, removing a line when it's full. Keeping this separation of drawing and game logic in mind will help you to write cleaner code that is easier to adapt (you will need to make adaptations during the defense of this practicum).

2.1 Representation

Think about the best way to represent the game. You need to keep track of the current state of the game (blocks that are already filled, information about the current piece) and state-independent models (shapes of different kinds of pieces). Chose your data structures wisely.

The provided code (see end of this document) has a method `initialize()` which is called once when the game is started. The data structure this method returns is passed on to other functions (`onloop()`, `onkey()`, `draw()`) as the `model` argument. You can change the `initialize()` method so it returns your own representation (which may contain both the current state description as well as state-independent models).

2.2 Drawing

Write a function that, given your model (representation), draws the game user interface. Use `tkinter`'s `Canvas` to draw the playing field. The `Canvas` provides the same functionality as the `graphics` package you have been using so far. Read more about `Canvas` here: `http://effbot.org/tkinterbook/canvas.htm`.

Tip: One approach to draw the Tetris playing field is to think of it as a grid of rectangles that change their color. We already provided an essential part (see `draw()` function) of the drawing code for this approach in the provided code at the end of this assignment. You can use this code.

2.3 Control

Write the controlling code that handles the game logic. It should detect the user input (e.g. arrows) to control the current block and change the model accordingly. It should also detect special events in the game, like when a row is completely filled and respond in the correct way (remove the row). Every change in the model should be propagated to the drawing part such that it updates what the user sees.

The provided code (see end of this document) already implements detection of key press events and the game loop. What follows here provides more explanation about these concepts and how they are implemented in the provided code.

Detecting key presses and other user input is quite straightforward with `tkinter`, as described here: `http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm`. To capture keyboard events, a function must be bound to the key press event. The function bound to the event receives the event as argument:

```
def keypress(event):
        print(event.keycode)
```

This function can be bound to a `Widget` `w`'s key press event with:

```
w.bind('<Key>', keypress)
```

The `keypress` argument in this line is the function we defined previously. After the binding code is executed, the `keypress` function is called by `tkinter` whenever the user presses a key. You need to determine which key is pressed and to define what to do then (look at the `onkey()` function). Please note that it is not your code that calls `keypress`, but `tkinter`. The code you write in the `onkey` function (which is called from `keypress`) thus *reacts* to external events.

In Tetris, the current piece should keep falling down, one block at a time, even without any user interaction. To accomplish that, you need to execute the code to lower the current piece's position with one block every `X` milliseconds. This game loop is implemented in the provided code (see end of this document) using:

```
w.after(X, callback)
```

This line ensures that the function called `callback` is called `X` milliseconds after executing this line. In the `callback` function, the same line can be used again to ensure that the `callback` function is fired again after `X` milliseconds, thus enabling the game loop:

```
def callback(w, X):
        # do stuff (like lower current piece with one position)
        w.after(X, callback, w, X)
```

The third and fourth arguments of `after` here specifies the arguments that will be passed to `callback` when it is called in `X` milliseconds. More information on `after` can be found on: `http://effbot.org/tkinterbook/widget.htm#Tkinter.Widget.after-method`. You need to update the model at regular intervals, which can be done in the `onloop()` function. Please note that it is not your code that calls `gameloop`, but `tkinter`. The code you write in the `onloop` (which is called from `gameloop`) functions thus *reacts* to external events.

3 Adaptability test (Optional)

To test the adaptability of your implementation of Tetris, change the game as follows. Assign each new piece a random color. When the piece has reached its destination at the bottom,

the squares (blocks) representing the piece must retain their color, also when a line is filled up and is removed.

4 Some tips

- Use the debugger! Following the execution step-by-step will give you more insight in your code and will help you find mistakes. Try the debugger on the provided code to see when different functions are called, what is actually contained in variables in different functions and how the values of the variables change. This will help you understand the provided code.
- There is a forum on Toledo for this course to ask questions and discuss the problems.
- Think before you implement. If you feel your current implementation is not very adaptable, change your implementation. Do not hack a hack onto a hack.
- You will have 2 hours to implement an adaptation during the defense so make sure you know your code, so:
  - write clear code
  - use comments
  - separate your code into several clearly defined functions
  - avoid code duplication
  - if you find code duplication, try to extract the duplicated code and put it in a separate function
  - use clear variable names
- Below is the code to get you started. It is also available as a separate file on Toledo. Run this code to see what it does. Normally, you would not need to change anything in the `main()` function. You may want to change the update interval and the canvas dimensions. You can start writing your code in functions above the `main()` function.
- If you have more detailed questions about the exact requirements, contact us and check Toledo regularly.

```
from tkinter import *

def initialize():
    # called once when the game is started (main() executed)
    # [ put your own model/representation
    #   initialization here ]
    return {"dimensions": (10,20),
            "square": {"x":0, "y":0}}
    # the data structure returned from this method
    # is passed as parameter ''model'' to the functions
    # draw(), onkey() and onloop() below

def draw(model, canvas):
    # called after onkey() and onloop(), so every
    # X milliseconds and after each time the user
    # presses a key
    canvas.delete(ALL)
    # clear canvas
    block_height = 20
    block_margin = 4
    dimensions = model["dimensions"]
    square = model["square"]
    for x in range(dimensions[0]):
        for y in range(dimensions[1]):
            color = "#f2f2f2"
            # default color of empty block
            if x==square["x"] and y==square["y"]:
                color = "#555"
                # color of filled block
            rect = canvas.create_rectangle(
                x*block_height+(x+1)*block_margin,
                y*block_height+(y+1)*block_margin,
                (x+1)*block_height+(x+1)*block_margin,
                (y+1)*block_height+(y+1)*block_margin,
                fill=color, outline=color)
            # draws a rectangle
    # draws rectangle grid

def onkey(model, keycode):
    # called when user presses a key
    # [ put your own code here ]
    print(keycode)
```

```
    square = model["square"]
    square["x"] += 1

def onloop(model):
    # called every X milliseconds
    # [ put your own code here ]
    square = model["square"]
    square["y"] += 1

########################################################################
# normally, you would not need to change anything in main #
def main(update_interval, canvas_dimensions):
    def keypress(event, model, canvas):
        onkey(model, event.keycode)
        draw(model, canvas)
    def gameloop(X, model, master, canvas):
        master.after(X, gameloop, X, model, master, canvas)
        onloop(model)
        draw(model, canvas)
    model = initialize()
    # initialize your model
    master = Tk()
    # initialize top level widget
    canvas = Canvas(master, width=canvas_dimensions[0],
            height=canvas_dimensions[1], background="white")
    # initialize canvas
    canvas.pack()
    master.bind("<Key>", lambda e: keypress(e, model, canvas))
    # bind the keypress() function to a key press event
    # while passing the model and the canvas as arguments too
    gameloop(update_interval, model, master, canvas)
    # start the gameloop
    master.mainloop()
    # enables event handling etc. by tkinter

########################################################################

if __name__ == "__main__":
    update_interval = 500
    canvas_dimensions = (300,500)
    # [ you might want to adjust these settings ]
    main(update_interval, canvas_dimensions)
```