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

# Hertendi Áron Levente #

# #

# # # # #

# ### ### ### #

# #

# Tetrícia #

# #

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

"""

Tetrícia

Some function descriptions are fully or partially taken from the 2009 Tetris Guideline:

https://www.dropbox.com/s/g55gwls0h2muqzn/tetris\_guideline\_docs\_2009.zip?dl=0

https://tetris.fandom.com/wiki/Tetris\_Guideline

"""

from tkinter import \*

import tkinter.ttk as ttk

from random import shuffle, randrange

import threading, time

from pynput.keyboard import Key, Listener

from chat\_gui import \*

class I:

"""I - Tetromino behaviour description"""

def generate():

return {'type': I, 'coords': [(x,20) for x in range(3,7)], 'rot': 'N', 'color':'#00ffff', 'name':'I'}

Defaults={'W':[(4,3),(4,4),(4,5),(4,6)],

'N':[(3,5),(4,5),(5,5),(6,5)],

'E':[(5,3),(5,4),(5,5),(5,6)],

'S':[(3,4),(4,4),(5,4),(6,4)]}

Points={'W':[(4,5),(4,5),(4,5),(4,3),(4,6)],

'N':[(4,5),(3,5),(6,5),(3,5),(6,5)],

'E':[(4,5),(5,5),(5,5),(5,6),(5,3)],

'S':[(4,5),(6,5),(3,5),(6,4),(3,4)],}

class T:

"""T - Tetromino behaviour description"""

def generate():

return {'type': T, 'coords': [(x,20) for x in range(3,6)]+[(4,21)], 'rot': 'N', 'color':'#800080', 'name':'T'}

Defaults={'W':[(4,3),(4,4),(4,5),(3,4)],

'N':[(3,4),(4,4),(5,4),(4,5)],

'E':[(4,5),(4,4),(4,3),(5,4)],

'S':[(5,4),(4,4),(3,4),(4,3)]}

Points={'W':[(4,4),(3,4),(3,3),(4,6),(3,6)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(5,4),(5,3),(4,6),(5,6)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class L:

"""L - Tetromino behaviour description"""

def generate():

return {'type': L, 'coords': [(x,20) for x in range(3,6)]+[(5,21)], 'rot': 'N', 'color':'#ffa500', 'name':'L'}

Defaults={'W':[(4,3),(4,4),(4,5),(3,5)],

'N':[(3,4),(4,4),(5,4),(5,5)],

'E':[(4,5),(4,4),(4,3),(5,3)],

'S':[(5,4),(4,4),(3,4),(3,3)]}

Points={'W':[(4,4),(3,4),(3,3),(4,6),(3,6)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(5,4),(5,3),(4,6),(5,6)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class J:

"""J - Tetromino behaviour description"""

def generate():

return {'type': J, 'coords': [(x,20) for x in range(3,6)]+[(3,21)], 'rot': 'N', 'color':'#0000ff', 'name':'J'}

Defaults={'W':[(4,3),(4,4),(4,5),(3,3)],

'N':[(3,4),(4,4),(5,4),(3,5)],

'E':[(4,5),(4,4),(4,3),(5,5)],

'S':[(5,4),(4,4),(3,4),(5,3)]}

Points={'W':[(4,4),(3,4),(3,3),(4,6),(3,6)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(5,4),(5,3),(4,6),(5,6)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class S:

"""S - Tetromino behaviour description"""

def generate():

return {'type': S, 'coords': [(3,20),(4,20), (4,21),(5,21)], 'rot': 'N', 'color':'#00ff00', 'name':'S'}

Defaults={'W':[(4,3),(4,4),(3,4),(3,5)],

'N':[(3,4),(4,4),(4,5),(5,5)],

'E':[(4,5),(4,4),(5,4),(5,3)],

'S':[(5,4),(4,4),(4,3),(3,3)]}

Points={'W':[(4,4),(3,4),(3,3),(4,6),(3,6)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(5,4),(5,3),(4,6),(5,6)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class Z:

"""Z - Tetromino behaviour description"""

def generate():

return {'type': Z, 'coords': [(4,20),(5,20), (3,21),(4,21)], 'rot': 'N', 'color':'#ff0000', 'name':'Z'}

Defaults={'W':[(4,4),(4,5),(3,3),(3,4)],

'N':[(4,4),(5,4),(3,5),(4,5)],

'E':[(4,4),(4,3),(5,5),(5,4)],

'S':[(4,4),(3,4),(5,3),(4,3)]}

Points={'W':[(4,4),(3,4),(3,3),(4,6),(3,6)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(5,4),(5,3),(4,6),(5,6)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class O:

"""O - Tetromino behaviour description"""

def generate():

return {'type': O, 'coords': [(4,20),(5,20), (4,21),(5,21)], 'rot': 'N', 'color':'#ffff00', 'name':'O'}

Defaults={'W':[(4,4),(5,4),(4,5),(5,5)],

'N':[(4,4),(5,4),(4,5),(5,5)],

'E':[(4,4),(5,4),(4,5),(5,5)],

'S':[(4,4),(5,4),(4,5),(5,5)]}

Points={'W':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'N':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'E':[(4,4),(4,4),(4,4),(4,4),(4,4)],

'S':[(4,4),(4,4),(4,4),(4,4),(4,4)]}

class GameDashboard(Frame):

"""

Definition of the frame containing all the information the player needs for the gameplay, except other player's boards.

init(master, blocksize=30, level=1)

master is the parent of the frame

blocksize is the size of the x\*x block of which each Tetromino is built of, scales the canvas

level is the initial game difficulcity (speed and scoring multiplier)

"""

def \_\_init\_\_(self, master, mixer,sounds, blocksize=30, level=1):

Frame.\_\_init\_\_(self)

##Blocksize in pixels

self.blocksize = blocksize

self.mixer=mixer

##The level to start the game on

self.level=level

self.sounds=sounds

##Different lock object for gameplay and network interferences

self.netLock = threading.Lock()

##Different lock object for gameplay and network interferences

self.gameLock = threading.Lock()

##Is the player in the game right now?

self.ingame = False

self.paused = False

self.online=False

self.gameThread=None

##Default background color

self.bg="black"

##The main canvas and the map of the game

self.can = Canvas(self, width=10\*blocksize, height=20\*blocksize+5, bg=self.bg)

self.can.create\_line(0,0,10\*blocksize, 0, fill="white")

self.can.yview\_scroll(22, 'units')

##The hold canvas

self.hold\_can = Canvas(self, width=6\*blocksize, height=4\*blocksize, bg=self.bg)

##Canvas of the Queue

self.queue\_can = Canvas(self, width=6\*blocksize, height=20\*blocksize+5, bg=self.bg)

self.queue\_can.create\_rectangle(0,0,7\*blocksize,blocksize\*3.33, fill="cyan")

self.bag=Bag(self.queue\_can, blocksize)

#Widget placements

self.hold\_can.grid(row=0, column=0,columnspan=2, padx=5, pady=5, sticky=N)

self.can.grid(row=0, column=2, pady=5, rowspan=5)

self.queue\_can.grid(row=0, column=3, rowspan=5, padx=5, pady=5, sticky = N)

#Font

self.font=font.Font(family='Comic Sans MS', size=12, weight='bold', slant='roman')

##Frame containing our labels

self.label\_frame=Frame(self)

self.label\_frame.grid(row=4, column=0)

Label(self.label\_frame,text="Incoming Lines: \n\n",font=self.font).grid(row=0, column=0, sticky="NW")

self.l\_attacks=Label(self.label\_frame,text="0",font=self.font, fg="red")

self.l\_attacks.grid(row=0, column=1, sticky="NW")

Label(self.label\_frame,text="Points: ",font=self.font).grid(row=3, column=0, sticky="SW")

Label(self.label\_frame,text="Level: ",font=self.font).grid(row=4, column=0, sticky="SW")

Label(self.label\_frame,text="Lines cleared: ",font=self.font).grid(row=5, column=0, sticky="SW")

self.l\_points=Label(self.label\_frame,text="0",font=self.font)

self.l\_levels=Label(self.label\_frame,text="0",font=self.font)

self.l\_lines=Label(self.label\_frame,text="0",font=self.font)

self.l\_points.grid(row=3, column=1, sticky="SW")

self.l\_levels.grid(row=4, column=1, sticky="SW")

self.l\_lines.grid(row=5, column=1, sticky="SW")

#Bindings

#self.bind("<Destroy>", self.\_destroy)

#self.master.protocol("WM\_DELETE\_WINDOW", self.\_destroy)

self.master.bind("<space>", self.button\_space)

self.master.bind("c", self.button\_c)

self.master.bind("<Shift\_L>", self.button\_shift\_l)

#Sound

self.bgvar=IntVar()

self.bgvar.set(1)

self.chmusic=ttk.Checkbutton(self,text="Music", command=self.swmusic,variable=self.bgvar, state=ACTIVE)

self.chmusic.grid(row=1, column=0)

self.bgmusic=StringVar()

self.bgmusic.set("bg2")

Label(self,text="Music volume", font=self.font).grid(row=2, column=0,sticky="S")

self.vol = ttk.Scale(self, from\_=0, to=100, orient=HORIZONTAL,command=self.set\_vol)

self.vol.grid(row=3,column=0,sticky=N)

self.vol.set(100)

self.choosemusic=ttk.Button(self, text="Chose",command=self.choose\_music)

self.choosemusic.grid(row=2, column=0,sticky=N)

##Play button to start playing

self.startButton = ttk.Button(self, text="\nPLAY\n", command=self.start\_new\_game, width=int(0.85\*blocksize))

self.startButton.grid(row=0, column=0,padx=8, sticky="S")

def set\_vol(self,evt):

"""Set the volume of all music"""

self.mixer.Channel(0).set\_volume(float(evt)/100)

def choose\_music(self):

"""Toplevel window to costumize connection settings"""

self.can.focus\_set()

self.costumize=Toplevel(self)

self.costumize.grab\_set()

b\_confirm=ttk.Button(self.costumize, command= lambda :self.close\_choose(), text="OK")

b\_confirm.grid(row=0, column=3, rowspan=2,padx=10)

Radiobutton(self.costumize, text="Ievan Polkka", variable=self.bgmusic, value="bg1").grid(row=0)

Radiobutton(self.costumize, text="Ao no kanata", variable=self.bgmusic, value="bg").grid(row=1)

Radiobutton(self.costumize, text="Коробейники", variable=self.bgmusic, value="bg2").grid(row=2)

Radiobutton(self.costumize, text="Коробейники piano", variable=self.bgmusic, value="bg3").grid(row=3)

self.costumize.title('Music settings')

self.costumize.geometry("+%d+%d" % (self.master.winfo\_rootx()+50,

self.master.winfo\_rooty()+50))

self.costumize.protocol("WM\_DELETE\_WINDOW", self.close\_choose)

self.costumize.resizable(0,0)

self.costumize.transient(self.costumize.master)

def close\_choose(self):

"""Called upon closing the music selecting dialog"""

self.costumize.destroy()

self.swmusic()

def swmusic(self):

"""Toggle the background/endgame music"""

self.can.focus\_set()

if 'selected' in self.chmusic.state() and self.ingame:

self.mixer.Channel(0).play(self.sounds[self.bgmusic.get()],loops=-1)

else:

self.mixer.Channel(0).stop()

def \_destroy(self):

"""Window closed event handler"""

if self.ingame:

self.gameThread.call\_quit()

def start\_new\_game(self):

"""Start a solo game"""

self.can.focus\_set()

if not self.ingame:

self.set\_levels(self.level)

self.can.delete(ALL)

self.can.create\_line(0,0,10\*self.blocksize, 0, fill="white")

self.queue\_can.delete(ALL)

self.queue\_can.create\_rectangle(0,0,7\*self.blocksize,self.blocksize\*3.33, fill="cyan")

self.hold\_can.delete(ALL)

self.ingame=True

self.swmusic()

self.bag.start()

self.gameThread=GameEngine(self, self.can, self.blocksize, self.level,self.bag, self.mixer,self.sounds,self.online)

self.gameThread.start()

def button\_space(self, event):

"""keyboard.space Event handler"""

if self.ingame and not self.paused:

self.gameThread.call\_hard\_drop()

def button\_c(self, event):

"""keyboard.c Event handler"""

if self.ingame and not self.paused:

self.gameThread.call\_hold()

def button\_shift\_l(self, event):

"""keyboard.l\_shift Event handler"""

if self.ingame and not self.paused:

self.gameThread.call\_hold()

def set\_points(self,points):

"""Update the corresponding stats tracking label"""

self.l\_points.config(text="%d"%points)

def set\_levels(self,level):

"""Update the corresponding stats tracking label"""

self.l\_levels.config(text="%d"%level)

def set\_lines(self,lines):

"""Update the corresponding stats tracking label"""

self.l\_lines.config(text="%d"%lines)

def set\_attacks(self,lines):

"""Update the corresponding stats tracking label"""

self.l\_attacks.config(text="%d"%lines)

class GameEngine(threading.Thread):

"""The main gameplay's Thread"""

def \_\_init\_\_(self, boss, can, blocksize, level,bag, mixer,sounds,online=False):

threading.Thread.\_\_init\_\_(self)

self.setDaemon(True)

self.mixer=mixer

self.sounds=sounds

self.boss = boss

self.can = can

self.blocksize = blocksize

self.bag=bag

self.online=online

self.soft\_drop\_flag=False

#Game Phase tracker

self.phase="Inactive"

#Window closed?

self.abandon=False

#Hold slot

self.hold\_slot=None

#Game Matrix

#A: Active Tetromino

#B: Block

#OGM: Object Game Matrix

self.GM = [[0]\*40 for x in range(10)]

self.OGM=[[0]\*40 for x in range(10)]

#Initialize the active Tetromino's namespace

self.active=None

#The ghost piece

self.ghost=None

self.bonuses=["Single","Double","Triple","Tetris","Mini T-Spin","Mini T-Spin Single","T-Spin","T-Spin Single","T-Spin Double","T-Spin Triple"]

self.multiplier=[100,300,500,800,100,200,400,800,1200,1600]

self.attacks=[0,1,2,4,0,0,0,2,4,6]

if self.online:

self.players=len(self.boss.master.players)

if self.players>1:

self.attacks=[0,0,1,2,0,0,0,1,2,5]

self.score={}

self.\_gameScore=0

self.\_lineScore=0

self.\_levelScore=level

self.\_newAttacks=0

self.gameScore=0

self.levelScore=level

self.lineScore=0

self.newAttacks=0

self.B2B=False

# When any single button is then released, the Tetrimino should again move in the direction still held,

# with the Auto-Repeat delay of roughly 0.3 seconds applied once more.

self.auto\_repeat\_delay=0.21

self.auto\_repeat\_speed=0.03

# The method to repeat

self.to\_repeat=None

# Can we be repeating yet?

self.timer\_repeat=0

# secondary timer for the repeat phase

self.last\_repeat=0

# Keyboard listening

self.up\_released=True

self.ctrl\_l\_released=True

self.pressed=False

self.held={}

self.kb\_listen=Listener(on\_press=self.on\_press, on\_release=self.on\_release)

self.kb\_listen.start()

#Line attacks

self.gap\_position=randrange(0,10)

self.lift\_count=0

#Online victory

self.win=False

@property

def gameScore(self):

"""I'm the 'gameScore' property."""

return self.\_gameScore

@gameScore.setter

def gameScore(self, value):

"""I'm the setter of 'gameScore' property."""

self.\_gameScore = value

self.boss.set\_points(self.\_gameScore)

@property

def lineScore(self):

"""I'm the 'lineScore' property."""

return self.\_lineScore

@lineScore.setter

def lineScore(self, value):

"""I'm the setter of 'lineScore' property."""

self.\_lineScore = value

self.boss.set\_lines(self.\_lineScore)

@property

def levelScore(self):

"""I'm the 'levelScore' property."""

return self.\_levelScore

@levelScore.setter

def levelScore(self, value):

"""I'm the setter of 'levelScore' property."""

self.\_levelScore = value

self.boss.set\_levels(self.\_levelScore)

@property

def newAttacks(self):

"""I'm the 'newAttacks' property."""

return self.\_newAttacks

@newAttacks.setter

def newAttacks(self, value):

"""I'm the setter of 'newAttacks' property."""

self.\_newAttacks = value

self.boss.set\_attacks(self.\_newAttacks)

def on\_press(self, key):

"""pynput event handler"""

if self.pressed==key:

return True

elif key==Key.up and self.up\_released==False:

return True

elif key==Key.ctrl\_l and self.ctrl\_l\_released==False:

return True

else:

#Don't want to gain the lock unnecessarily

if key not in (Key.left, Key.right, Key.up, Key.ctrl\_l, Key.down):return True

self.boss.gameLock.acquire()

if key==Key.left:

self.pressed=key

self.reset\_auto\_repeat\_cooldowns()

self.held[key]=True

self.to\_repeat=self.move\_left

elif key==Key.right:

self.pressed=key

self.reset\_auto\_repeat\_cooldowns()

self.held[key]=True

self.to\_repeat=self.move\_right

elif key==Key.up:

self.up\_released=False

self.call\_rotate\_cw()

elif key==Key.ctrl\_l:

self.ctrl\_l\_released=False

self.call\_rotate\_ccw()

elif key==Key.down:

self.mixer.Channel(1).play(self.sounds["move"])

if not self.call\_soft\_drop():

self.call\_soft\_drop()

self.boss.gameLock.release()

def on\_release(self,key):

"""pynput event handler"""

if self.pressed==key:

self.boss.gameLock.acquire()

self.pressed=False

self.reset\_auto\_repeat\_cooldowns()

self.held[key]=False

self.boss.gameLock.release()

# Check if another key was held through a press-release

for key in self.held:

if self.held[key]:

self.on\_press(key)

break

elif key in self.held:

self.held[key]=False

elif key==Key.up:

self.up\_released=True

elif key==Key.ctrl\_l:

self.ctrl\_l\_released=True

elif key==Key.down:

self.call\_soft\_drop(False)

def reset\_auto\_repeat\_cooldowns(self):

"""Set timers to 0 when an action just occured"""

self.last\_repeat=0

self.timer\_repeat=0

def call\_quit(self):

"""Call this to shut down the Engine Thread"""

self.abandon=True

def won(self):

"""Call this when the player won"""

self.win=True

def run(self):

"""Run, run, run, run-runrunrunruuuuuuun!"""

try:

while True:

self.boss.ingame=True

self.phase = "generation"

if self.generation\_phase():

self.phase="falling"

locked=False

while not locked:

if self.phase=="falling":

locked=self.falling\_phase()

if not locked:

self.phase="locking"

elif self.phase=="locking":

locked=self.lock\_phase()

if not locked:

self.phase="falling"

else:

# if self.phase="pattern":

# self.pattern\_phase()

raise "This should've never occur!"

self.phase="pattern"

self.pattern\_phase()

self.eliminate\_phase()

else:

break

except AbandonException:

print("Player abandoned.")

except GameOverException:

print("The game was over.")

if self.online:

while self.boss.master.playing:

self.check\_opponents()

print("Exiting thread")

except Exception as e:

print(e)

def call\_hold(self):

"""Set flag to hold the piece"""

if self.phase not in ("falling", "locking"):

return

if self.hold==None:

self.hold=True

def call\_soft\_drop(self, logical=True):

"""Set flag for a Soft Drop"""

self.soft\_drop\_flag=logical

return self.soft\_drop\_flag

def soft\_drop(self):

"""Method which excecutes a soft drop whenever off cooldown."""

#self.soft\_drop\_flag=False

now=time.time()

if now-self.last\_linedrop<(self.speed/20):

return

else:

if not self.touching\_surface():

self.gameScore+=1

self.last\_linedrop=now

self.linedrop()

self.send\_coords()

self.send\_stats()

def call\_hard\_drop(self):

"""Set flag for a Hard Drop"""

if self.phase not in ("falling", "locking"):

return

self.hard\_drop\_flag=True

def hard\_drop(self):

"""Method which excecutes a Hard Drop"""

#How much is it possible to drop?

distance=self.distance\_from\_surface()

if distance>0:

self.spin\_last=False

[self.linedrop() for x in range(distance)]

self.gameScore+=distance\*2

self.send\_coords()

self.lock\_down()

self.hard\_drop\_flag=False

def distance\_from\_surface(self):

"""Finds the lowest distance between an open surface and the active Tetromino"""

#How much is it possible to drop?

min\_d=40

for x,y in self.active['coords']:

y0=0

if min\_d>y-y0:

min\_d=y-y0

for y0 in range(0,y):

if self.GM[x][y0]=='B':

if min\_d>y-y0-1:

min\_d=y-y0-1

return min\_d

def get\_quarter(self, original, direction):

"""Original quarter: NSWE: 1 char\nDirection: Left=-1, Right=+1"""

qrtrs=['E', 'N', 'W', 'S']

return qrtrs[(qrtrs.index(original)+direction)%4]

def call\_rotate\_cw(self):

"""Set flag for a Clockwise Rotation"""

if self.phase not in ("falling", "locking"):

return

self.rotate\_cw\_flag=True

def call\_rotate\_ccw(self):

"""Set flag for a Clockwise Rotation"""

if self.phase not in ("falling", "locking"):

return

self.rotate\_ccw\_flag=True

def rotate(self, counter\_clockwise=False):

"""Clockwise rotation event = RIGHT"""

#Reset flag

if counter\_clockwise:

self.rotate\_ccw\_flag=False

direction=1

else:

self.rotate\_cw\_flag=False

direction=-1

bs=self.blocksize

src=self.active['rot']

dest=self.get\_quarter(src,direction)

new\_coords=self.SRS(src, dest, self.active['coords'], self.GM,self.active['type'])

if new\_coords==None:

return False

#Rotation succeeded

if self.mixer:

self.mixer.Channel(2).play(self.sounds["rotate"])

self.last\_action=time.time()

self.active['rot']=dest

for x,y in self.active['coords']:

self.GM[x][y]=0

self.active['coords']=new\_coords[:]

for x,y in new\_coords:

self.GM[x][y]='A'

#Visual

#Replace the objects

for i in range(4):

x,y=self.active['coords'][i]

self.can.coords(self.active['objects'][i], 2+(bs\*x),-(y-19)\*bs,2+bs+(bs\*x), -(y-20)\*bs)

#Adjust ghost

self.ghost\_adjust()

return True

def SRS(self,from\_, to\_, coords, matrix, Tetromino):

"""Super Rotation System - coordinate calculor"""

for P in range(5):

#return I.west(P,I.Points['N'][P])

#new\_coords=I.west(P,I.Points['N'][P])

p=Tetromino.Points[to\_][P]

dest=Tetromino.Points[from\_][P]

shift=(dest[0]-p[0],dest[1]-p[1])

new\_coords=[]

for x,y in Tetromino.Defaults[to\_]:

new\_coords.append((x+shift[0], y+shift[1]))

#The difference between the old and the new coordinates

map\_shift=[(new\_coords[x][0]-Tetromino.Defaults[from\_][x][0], new\_coords[x][1]-Tetromino.Defaults[from\_][x][1]) for x in range(4)]

#The real new coordinates

map\_coords=[(coords[x][0]+map\_shift[x][0], coords[x][1]+map\_shift[x][1]) for x in range(4)]

#Test these

failed=False

for x,y in map\_coords:

if x<0 or y<0 or x>9 or y>40 or matrix[x][y]=='B':

failed=True

break

if failed:continue

return map\_coords

def move\_right(self):

"""Attemps to move the Tetromino one block right. Returns True if successful and False if not."""

for x,y in self.active['coords']:

if x==9:return False

if self.GM[x+1][y]=='B':return False

self.last\_action=time.time()

self.spin\_last=False

#Backend

#Writing into both the self.active and the main matrix

new\_coords=[]

for x,y in self.active['coords']:

new\_coords.append((x+1,y))

self.GM[x][y]=0

self.active['coords']=new\_coords[:]

for x,y in new\_coords:

self.GM[x][y]='A'

#Visual

for block in self.active['objects']:

self.can.move(block, self.blocksize, 0)

if self.mixer:

self.mixer.Channel(1).play(self.sounds["move"])

self.ghost\_adjust()

return True

def move\_left(self):

"""Attemps to move the Tetromino one block left. Returns True if successful and False if not."""

for x,y in self.active['coords']:

if x==0:return False

if self.GM[x-1][y]=='B':return False

self.last\_action=time.time()

self.spin\_last=False

#Backend

#Writing into both the self.active and the main matrix

new\_coords=[]

for x,y in self.active['coords']:

new\_coords.append((x-1,y))

self.GM[x][y]=0

self.active['coords']=new\_coords[:]

for x,y in new\_coords:

self.GM[x][y]='A'

#Visual

for block in self.active['objects']:

self.can.move(block, -self.blocksize, 0)

if self.mixer:

self.mixer.Channel(1).play(self.sounds["move"])

self.ghost\_adjust()

return True

def ghost\_adjust(self):

"""This method adjusts the ghost piece to the new position. Each turn or horizontal move action should call this."""

distance=self.distance\_from\_surface()

bs=self.blocksize

i=0

for x,y in self.active['coords']:

self.can.coords(self.ghost[i],2+(bs\*x),-(y-19-distance)\*bs,2+bs+(bs\*x), -(y-20-distance)\*bs)

i+=1

def generation\_phase(self, from\_hold=False):

"""

The generation time of a Tetrimino is 0.2 seconds after the Lock Down of the previous Tetrimino.

This slight delay happens as soon as the Completion Phase is finished.

Generation time may change depending on the handling of the target platform.

As soon as a Tetrimino is generated, three things immediately happen:

1) the Tetrimino drops one row if no existing Block is in its path,

2) the Tetrimino enters the Falling Phase where the player is able to move and rotate it, and

3) the Ghost Piece (if turned on) appears below, North Facing.

If an existing Block is in the Tetrimino’s path, the Tetrimino does not drop one row immediately,

however, a few pixels of the generated Tetrimino are shown (hardware permitting)

to help the player manipulate it above the Skyline.

"""

if self.win:

self.eliminate=[x for x in range(40)]

self.eliminate\_phase()

self.send\_won()

self.boss.ingame=False

if self.online:

self.boss.master.chat.write("You won!\nScore: %s"%self.gameScore)

else:

messagebox.showinfo("You won!","Score: %s"%self.gameScore)

raise GameOverException()

if self.newAttacks>0:

for x in range(self.newAttacks):

self.lift()

time.sleep(0.05)

self.newAttacks=0

bs=self.blocksize

#Game speed

self.speed=(0.8 - ((self.levelScore - 1) \* 0.007))\*\*(self.levelScore - 1)

#Score in each turn will be logged in a dictionary

self.score={}

#Lowest line tracker

self.lowest\_line\_reached=21

#Reset Flags

#Did a hard drop occur?

self.hard\_drop\_flag=False

#Soft drop?

#Rotate?

self.rotate\_cw\_flag=False

self.rotate\_ccw\_flag=False

#Lines to eliminate

self.eliminate=[]

#Action counter in locking phasee

self.counter=0

#Timing the soft drop

self.last\_linedrop=0

#Timing of the last action (move/rotate, NOT drop)

self.last\_action=time.time()

#Did it lock down after a spin?

self.spin\_last=False

if self.ghost:

for i in self.ghost:

self.can.delete(i)

#Can the player Hold this round

#Hold

#None: Last piece was taken from bag

#True: The flag was set to True, to make the Engine call generation\_phase again.

#False: The last piece was taken from the Hold, so the player cannot swap again. (The flag wont be set to true in this case in the call)

if from\_hold:

for x,y in self.active['coords']:

self.GM[x][y]=0

for i in self.active['objects']:

self.can.delete(i)

held=self.active['type']

if self.hold\_slot==None:

self.active=self.bag.next().generate()

else:

self.active=self.hold\_slot.generate()

self.hold\_slot=held

self.hold=False

self.place\_hold(self.hold\_slot)

self.send\_hold()

else:

# Pick up the next tetromino from the Next Queue

self.hold=None

self.active=self.bag.next().generate()

if self.block\_out(self.active['coords']):

self.game\_over()

self.active['objects']=[]

for x,y in self.active['coords']:

self.GM[x][y]='A'

self.active['objects'].append(self.can.create\_rectangle(2+(bs\*x),-(y-19)\*bs,2+bs+(bs\*x), -(y-20)\*bs, fill=self.active['color']))

distance=self.distance\_from\_surface()

self.ghost=[]

for x,y in self.active['coords']:

self.ghost.append(self.can.create\_rectangle(2+(bs\*x),-(y-19-distance)\*bs,2+bs+(bs\*x), -(y-20-distance)\*bs, outline=self.active['color']))

for i in self.active['objects']:

self.can.tag\_raise(i)

#Update the server

self.send\_mino()

return 1

def place\_hold(self, tetromino):

"Places the tetromino to the hold canvas"

self.boss.hold\_can.delete(ALL)

curr=tetromino.generate()

bs=self.blocksize\*0.85

dx=-(bs\*(5/3))

dy=0

if curr['type']==O:

dx=-(bs\*2)

elif curr['type']==I:

dx=-(bs\*2)

dy=-bs\*0.5

[self.boss.hold\_can.create\_rectangle(dx+(bs\*(x+0.5)),dy+(bs\*3.5)-(y-19)\*bs,dx+bs+(bs\*(x+0.5)), dy+(bs\*3.5)-(y-20)\*bs, fill=curr['color']) for x,y in curr['coords']]

def falling\_phase(self):

"During falling, the player can rotate, move sideways, soft drop, hard drop or hold the Tetromino"

while self.phase=="falling":

self.check\_opponents()

#coordinates

c1=self.active['coords'].copy()

if self.abandon:raise AbandonException()

if self.boss.paused: continue

if self.hold:

self.generation\_phase(True)

#Hard Drop?

if self.hard\_drop\_flag:

self.hard\_drop()

return True

#Atop Surface?

if self.touching\_surface():

#return to main cycle

return False

now=time.time()

if now-self.last\_linedrop>=self.speed:

self.last\_linedrop=now

self.linedrop()

self.send\_coords()

continue

#Soft Drop?

if self.soft\_drop\_flag:

self.soft\_drop()

if self.rotate\_cw\_flag:

act=self.rotate()

if act:self.spin\_last=True

elif self.rotate\_ccw\_flag:

act=self.rotate(True)

if act:self.spin\_last=True

self.boss.gameLock.acquire()

if self.pressed:

now=time.time()

if self.timer\_repeat==0:

self.timer\_repeat=now

self.to\_repeat()

elif self.last\_repeat==0:

if now-self.timer\_repeat>=self.auto\_repeat\_delay:

self.last\_repeat=now

self.to\_repeat()

elif now-self.last\_repeat>=self.auto\_repeat\_speed:

self.last\_repeat=now

self.to\_repeat()

self.boss.gameLock.release()

#Send to server if the coordinates have changed

if self.active['coords']!=c1:

self.send\_coords()

raise "Only the run() method should set the phase flag"

def linedrop(self):

"Let the tetromino fall down one line. WARNING: This function does not check circumstances."

#Backend

#Writing into both the self.active and the main matrix

self.spin\_last=False

new\_coords=[]

for x,y in self.active['coords']:

new\_coords.append((x,y-1))

self.GM[x][y]=0

self.active['coords']=new\_coords[:]

for x,y in new\_coords:

self.GM[x][y]='A'

if y<self.lowest\_line\_reached:

self.lowest\_line\_reached=y

self.counter=0

#Visual

for block in self.active['objects']:

self.can.move(block, 0, self.blocksize)

def lock\_phase(self):

"During lock phase the player still rotate or move according to Extendended Placement Lockdown\nLock after: 0.5s\nAction limit: 15 actions"

self.last\_action=time.time()

while self.phase=="locking":

self.check\_opponents()

c1=self.active['coords'].copy()

if self.abandon:raise AbandonException()

if self.boss.paused: continue

if self.hold:

self.generation\_phase(True)

return False

#Still Atop Surface?

if not self.touching\_surface():

#return to main cycle

self.spin\_last=False

return False

if self.counter==15:

self.lock\_down()

return True

#Hard Drop?

if self.hard\_drop\_flag:

self.hard\_drop()

return True

#Soft Drop?

#Once the surface is reached, Soft Drop should not auto-repeat, rather just wait out the 0.5 sec to lock down.

#rotation?

if self.rotate\_cw\_flag:

act=self.rotate()

if act:

self.counter+=1

self.spin\_last=True

elif self.rotate\_ccw\_flag:

act=self.rotate(True)

if act:

self.counter+=1

self.spin\_last=True

elif self.pressed:

do=False

now=time.time()

if self.timer\_repeat==0:

self.timer\_repeat=now

do=True

elif self.last\_repeat==0:

if now-self.timer\_repeat>=self.auto\_repeat\_delay:

self.last\_repeat=now

do=True

elif now-self.last\_repeat>=self.auto\_repeat\_speed:

self.last\_repeat=now

do=True

if do:

act=self.to\_repeat()

if act:

self.counter+=1

self.spin\_last=False

if self.active['coords']!=c1:

self.send\_coords()

if time.time()-self.last\_action>=0.5:

self.lock\_down()

return True

def lock\_down(self):

"Sets the Tetromino to the permanent state, deletes the ghost piece"

for x,y in self.active['coords']:

self.GM[x][y]='B'

self.OGM[x][y]=self.active['objects'][self.active['coords'].index((x,y))]

for x in self.ghost:

self.can.delete(x)

if self.mixer:

self.mixer.Channel(4).play(self.sounds["lock"])

self.send\_lock()

def pattern\_phase(self):

"""

In this phase, the engine looks for patterns made from Locked Down Blocks in the Matrix. Once a pattern has been matched, it can trigger any number of Tetris variant-related effects.

The classic pattern is the Line Clear pattern.

This pattern is matched when one or more rows of 10 horizontally aligned Matrix cells are occupied by Blocks.

The matching Blocks are then marked for removal on a hit list.

Blocks on the hit list are cleared from the Matrix at a later time in the Eliminate Phase.

This phase takes up no apparent game time.

"""

#Check for Lock Out

lockout=True

for x,y in self.active['coords']:

if y<=20:

lockout=False

break

if lockout:

self.game\_over()

tspin=False

mini\_tspin=False

#recognize T-Spins

if self.active['type']==T and self.spin\_last:

center=self.active['coords'][1]

leg=self.active['coords'][3]

diff=(leg[0]-center[0], leg[1]-center[1])

aleg=(center[0]-diff[0], center[1]-diff[1])

ab=[(leg[0]-diff[1], leg[1]-diff[0]), (leg[0]+diff[1],leg[1]+diff[0])]

cd=[(aleg[0]-diff[1], aleg[1]-diff[0]), (aleg[0]+diff[1],aleg[1]+diff[0])]

bs=self.blocksize

ab\_count=self.surfaces(ab)

cd\_count=self.surfaces(cd)

if ab\_count==2 and cd\_count>=1:

tspin=True

if cd\_count==2 and ab\_count==1:

mini\_tspin=True

#Find and mark lines for elimination

lines=0

clear=['B']\*10

for y in range(40):

line=[self.GM[x][y] for x in range(10)]

if line==clear:

lines+=1

self.eliminate.append(y)

#Bonuses are name of lineclears, tspins, or both with tspin name in front

bonus=""

if mini\_tspin:

bonus+="Mini T-Spin"

elif tspin:

bonus+="T-Spin"

if lines>0 and len(bonus)>0:

bonus+=" "

if lines>0:

bonus+=self.bonuses[lines-1]

#Increase the level if another 10 lines has been cleared

if self.levelScore<15:

before=self.lineScore//10

self.lineScore+=lines

after=self.lineScore//10

if after>before:self.levelScore+=1

#Back-To-Back recognition

apply\_b2b=False

if bonus in ("Tetris", "T-Spin Single", "T-Spin Double", "T-Spin Triple", "Mini T-Spin Single"):

if self.B2B:

self.send\_bonus("Back-To-Back "+bonus+"!")

apply\_b2b=True

else:

self.send\_bonus(bonus+"!")

self.B2B=True

elif bonus not in ("Mini T-Spin", "T-Spin"):

self.B2B=False

points=0

attack=0

if len(bonus)>0:

points=self.multiplier[self.bonuses.index(bonus)]

attack=self.attacks[self.bonuses.index(bonus)]

if apply\_b2b:

points\*=1.5

attack+=1

self.gameScore+=points

if self.newAttacks>=attack:

self.newAttacks-=attack

attack=0

elif self.newAttacks<attack:

attack-=self.newAttacks

self.newAttacks=0

if attack>0:

self.send\_attack(attack)

def surfaces(self, list\_):

"Determines how many coordinates is Surface"

counter=0

for x,y in list\_:

if y==-1 or x==-1 or x==10:

counter+=1

elif self.GM[x][y]=='B':

counter+=1

return counter

def eliminate\_phase(self):

"""

Involves animation. Note that I did the scoring in the pattern phase.

Any Minos marked for removal, i.e., on the hit list, are cleared from the Matrix in this phase.

If this results in one or more complete 10-cell rows in the Matrix becoming unoccupied by Minos,

then all Minos above that row(s) collapse, or fall by the number of complete rows cleared from the Matrix.

Points are awarded to the player according to the Tetris Scoring System,[...].

"""

self.send\_elim()

self.send\_stats()

self.clear\_marked\_lines()

def clear\_marked\_lines(self):

"Clears any line marked by having it's id in the sorted list self.eliminate"

if self.eliminate==[]:

return

elim=self.eliminate.copy()

for i in range(len(elim)):

y=elim[i]

for x in range(10):

del self.GM[x][y]

self.GM[x].append(0)

for j in range(len(elim)):

elim[j]-=1

#threading.Thread(target=self.clear\_line\_animation).start()

#time.sleep(0.0001)

if self.mixer:

self.mixer.Channel(3).play(self.sounds["clear"])

self.clear\_line\_animation()

def clear\_line\_animation(self):

"Method, removes the marked blocks visually. Note: This should take up no gametime, but the fact it does, only awards multiple line clears by a brief pause, which can be helpful at high levels."

elim=self.eliminate.copy()

rgb\_fact=0

while rgb\_fact<255:

for x in range(10):

for y in elim:

self.can.itemconfig(self.OGM[x][y], fill ="#%02x%02x%02x" % (rgb\_fact,rgb\_fact,rgb\_fact))

rgb\_fact+=16

time.sleep(0.005)

for i in range(len(elim)):

y=elim[i]

for x in range(10):

self.can.delete(self.OGM[x][y])

self.OGM[x][y]=0

for y1 in range(y+1,40):

if self.OGM[x][y1]:

self.OGM[x][y1-1]=self.OGM[x][y1]

self.can.move(self.OGM[x][y1], 0, self.blocksize)

self.OGM[x][y1]=0

time.sleep(0.01)

for j in range(len(elim)):

elim[j]-=1

def block\_out(self, coords):

"Game Over Condition - Is it possible to place the new Tetromino?"

for x,y in coords:

if self.GM[x][y]!=0:

return True

return False

def touching\_surface(self):

"Is the Tetromino directily on a surface?"

for x,y in self.active['coords']:

#Is it on the ground?

if y==0:return True

#Is it on top of another [B]lock?

if self.GM[x][y-1]=='B':

return True

def game\_over(self):

self.boss.ingame=False

self.send\_over()

if self.mixer:

self.mixer.Channel(0).play(self.sounds["over"], fade\_ms=8000)

#self.eliminate=[x for x in range(40)]

self.eliminate=[y for y in range(21)]

self.clear\_line\_animation()

self.eliminate=[y for y in range(20)]

self.clear\_line\_animation()

time.sleep(4)

if self.online:

self.boss.master.chat.write("Game over!\nYour score: %s"%self.gameScore)

else:

messagebox.showinfo("Game over!","Score: %s"%self.gameScore)

raise GameOverException()

def receive\_attacks(self,amount):

self.newAttacks+=amount

def lift(self):

"Manages receiving counter attacks"

#The game may be over here

self.check\_topout()

bs=self.blocksize

#New gap position after every 8 garbage line

if self.lift\_count%8==0:

new = randrange(0,10)

while new==self.gap\_position:

new = randrange(0,10)

self.gap\_position=new

self.lift\_count+=1

for x in range(10):

for y in range(40):

if self.OGM[x][y]!=0:

self.can.move(self.OGM[x][y], 0, -self.blocksize)

for x in range(10):

del self.GM[x][39]

del self.OGM[x][39]

if x == self.gap\_position:

self.GM[x][0:0]=[0]

self.OGM[x][0:0]=[0]

continue

self.GM[x][0:0]=['B']

self.OGM[x][0:0]=[self.can.create\_rectangle(2+(bs\*x),(19)\*bs,2+bs+(bs\*x), (20)\*bs, fill="darkgray", outline="gray")]

self.send\_lift(self.gap\_position)

def check\_opponents(self):

"""Check it all"""

if not self.online:return

for i in self.boss.master.players:

self.boss.master.players[i].run()

def check\_topout(self):

"""Only gets checked before a lift, ends the game if there's a tetromino part found in the 40th line"""

for x in range(10):

if self.GM[x][39]=='B':

self.game\_over()

def send\_coords(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

str1="["

for x,y in self.active['coords']:

str1+="("+str(x)+","+str(y)+"),"

str1+="]"

self.boss.master.update\_server("#GAME#COORDS#"+str1)

def send\_mino(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#NEW#"+self.active['name'])

def send\_lock(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#LOCK#0")

def send\_hold(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#HOLD#0")

def send\_over(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#OVER#0")

def send\_elim(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

str1="["

for y in self.eliminate:

str1+=str(y)+','

str1+="]"

self.boss.master.update\_server("#GAME#ELIM#"+str1)

def send\_stats(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#STAT#[%d,%d,%d]"%(self.gameScore, self.levelScore, self.lineScore))

def send\_attack(self, lines):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#ATTACK#%d"%(lines))

def send\_lift(self, gap):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#LIFT#%d"%(gap))

def send\_won(self):

"""Format the string such that it can be evaluated later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#WON#0")

def send\_bonus(self, bonus):

"""Format the string such that it can be announced later and forward it"""

if not self.online:return

self.boss.master.update\_server("#GAME#ANNOUNCE#%s"%bonus)

self.boss.master.chat.write("You had a "+bonus)

class Bag:

"""

Tetris uses a “bag” system to determine the sequence of Tetriminos that appear during game play.

This system allows for equal distribution among the seven Tetriminos.

"""

def \_\_init\_\_(self, queue\_can,blocksize):

self.minos=[T, I, J, L, S, Z, O]

self.queue\_can=queue\_can

self.next\_queue=[]

self.bag=[]

##tkinter graphical objects

self.objects=[]

##Original blocksize

self.orig\_box=blocksize

self.blocksize=blocksize\*(8/10)

def start(self):

"""Initialize the first Bag and Next Queue"""

self.bag=self.minos.copy()

shuffle(self.bag)

self.next\_queue=self.bag[:6]

for i in self.next\_queue:

self.queue\_forward(i, False)

del self.bag[:6]

def next(self):

"""Return the next tetromino for the Game Engine, and step the que forward"""

ret=self.next\_queue[0]

del self.next\_queue[0]

self.next\_queue.append(self.bag[0])

del self.bag[0]

if len(self.bag)==0:

self.bag=self.minos.copy()

shuffle(self.bag)

self.queue\_forward(self.next\_queue[-1])

return ret

def queue\_forward(self, mino, delete=True):

"""Delete (optionally) the top tetromino, Move each Tetromino up by one in the que, and place the next to the end of queue\nqueue\_forward(mino, delete=True)\nmino:Tetromino-type ref\ndelete: delete the top piece"""

if delete:

for i in self.objects[0]:

self.queue\_can.delete(i)

del self.objects[0]

for i in self.objects:

for j in i:

self.queue\_can.move(j, 0, -self.orig\_box\*3.33)

bs=self.blocksize

curr=mino.generate()

dx=self.orig\_box\*1.33

self.objects.append([self.queue\_can.create\_rectangle(-dx+(bs\*(x+1)),self.orig\_box\*19-(y-19)\*bs,-dx+bs+(bs\*(x+1)), self.orig\_box\*19-(y-20)\*bs, fill=curr['color']) for x,y in curr['coords']])

class AbandonException(Exception):

"""Exception occurs when the player quits during an ongoing (either paused or unpaused) game. This serves the purpose of closing down threads."""

pass

class GameOverException(Exception):

"""Exception occurs when the game is over. This serves the purpose of closing down threads."""

pass

if \_\_name\_\_ == '\_\_main\_\_':

from pygame import mixer # Load the required library

import os

os.system('cls')

root=Tk()

mixer.pre\_init(44100, 16, 2, 4096)

mixer.init()

sounds={"lock":mixer.Sound("effects/lock.ogg"),

"rotate":mixer.Sound("effects/rotate.ogg"),

"move":mixer.Sound("effects/move.ogg"),

"clear":mixer.Sound("effects/lineclear.ogg"),

"over":mixer.Sound("music/gameover.ogg"),

"bg":mixer.Sound("music/bg.OGG"),

"bg1":mixer.Sound("music/bg1.OGG"),

"bg2":mixer.Sound("music/bg2.OGG"),

"bg3":mixer.Sound("music/bg3.OGG")

}

fr=GameDashboard(root, mixer=mixer, sounds=sounds)

fr.grid(row=0, column=0)

root.title("Tetrícia")

root.resizable(0,0)

root.mainloop()