Christos Constantinou 2021

ULTIMATE TIC-TAC-TOE

A PHYSICAL INTERACTIVE GAME OF TIC-TAC-TOE USING A LAUNCHPAD S AND A.I.

Summary: Play an interactive version of tic tac toe using a launchpad. You have 3 modes to choose from Hard AI (Impossible), Easy AI (beatable) and 1v1(against human).

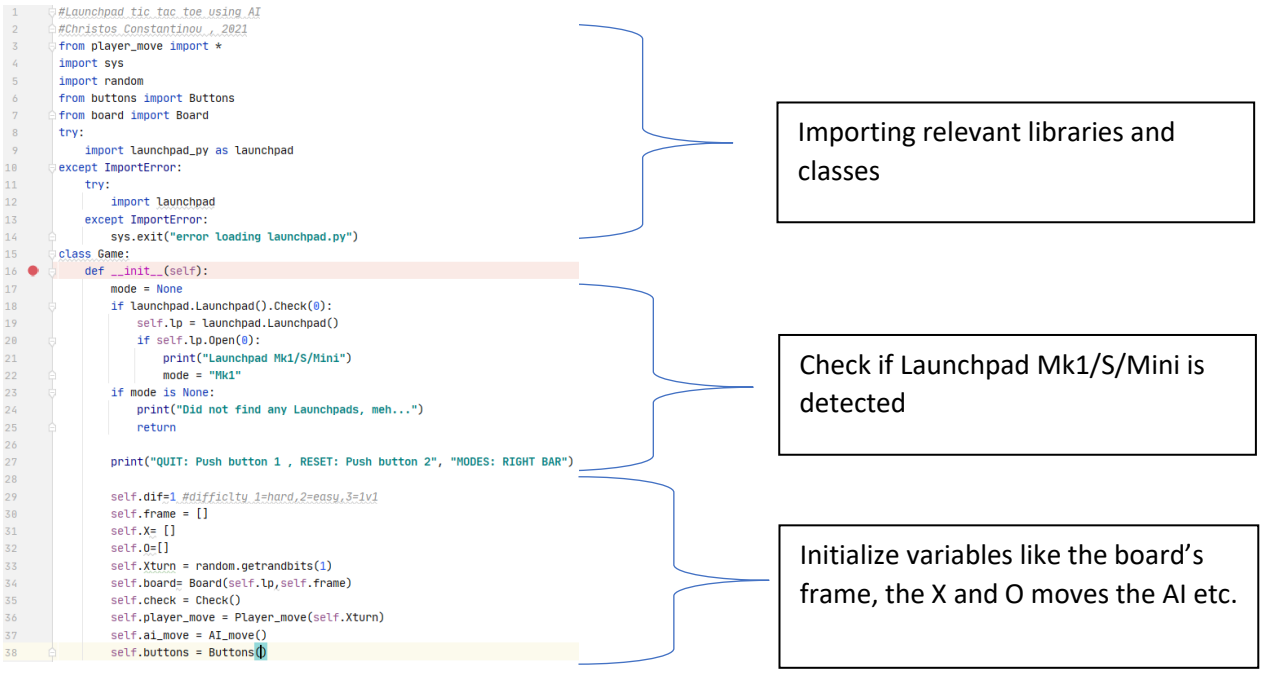
Installation: -Simply plug in your launchpad and run the code

Requirements: -external python library "launchpad.py"

-Launchpad S/Mk1/Mini (doesn't work with Mk2+ models)

Code Breakdown:

main.py (main Game class):



```python
#Launchpad tic tac toe using AI
#Christos Constantinou , 2021
from player_move import *
import sys
import random
from buttons import Buttons
from board import Board
try:
    import launchpad_py as launchpad
except ImportError:
    try:
        import launchpad
    except ImportError:
        sys.exit("error loading launchpad.py")
class Game:
    def __init__(self):
        mode = None
        if launchpad.Launchpad().Check(0):
            self.lp = launchpad.Launchpad()
            if self.lp.Open(0):
                print("Launchpad Mk1/S/Mini")
                mode = "Mk1"
        if mode is None:
            print("Did not find any Launchpads, meh...")
            return

        print("QUIT: Push button 1 , RESET: Push button 2", "MODES: RIGHT BAR")

        self.dif=1 #difficlty 1=hard,2=easy,3=1v1
        self.frame = []
        self.X= []
        self.O=[]
        self.Xturn = random.getrandbits(1)
        self.board= Board(self.lp,self.frame)
        self.check = Check()
        self.player_move = Player_move(self.Xturn)
        self.ai_move = AI_move()
        self.buttons = Buttons()
```

Importing relevant libraries and classes

Check if Launchpad Mk1/S/Mini is detected

Initialize variables like the board's frame, the X and O moves the AI etc.

```python
#game_loop
    def game_loop(self):
        events = self.lp.ButtonStateRaw()
        self.difficulty(events)
        Buttons.button_actions(self.buttons,events,self.lp,self.frame,self.X,self.O)
        Player_move.move(self.player_move,self.lp,self.frame,self.X,self.O,events,self.dif)
        Check.check_cases(self.check,self.lp,self.frame,self.X,self.O)
    def difficulty(self,events):
        if events == [205, True]:
            self.dif=1
        if events == [206, True]:
            self.dif=2
        if events == [207, True]:
            self.dif=3
if __name__ == "__main__":
    game = Game()
    while True:
        game.game_loop()
```

Executes events like registering button touch, setting difficulty, checking for win, registering moves etc.

Method for setting difficulty by pressing the top buttons.

Loops the game.

board.py (sets up the board):

```python
class Board:
    def __init__(self,lp,frame):
        self.lp=lp
        self.frame=frame
        #making the board frame and the interactive buttons
        for i in range(200):
            if (i-2)%16==0:
                self.lp.LedCtrlRaw(i, 0, 3)
                self.frame.append(i)
            if (i-5)%16==0:
                self.lp.LedCtrlRaw(i, 0, 3)
                self.frame.append(i)
        for i in range(32,40):
            self.lp.LedCtrlRaw(i, 0, 3)
            self.frame.append(i)
        for i in range(80,88):
            self.lp.LedCtrlRaw(i, 0, 3)
            self.frame.append(i)
        #launchpad side and top inactive buttons
        self.frame.append(8)
        self.frame.append(24)
        self.frame.append(40)
        self.frame.append(56)
        self.frame.append(72)
        self.frame.append(88)
        self.frame.append(104)
        self.frame.append(120)
        #launchpad top active buttons
        self.lp.LedCtrlRaw(200,2,0)
        self.lp.LedCtrlRaw(201,0,2)
        self.lp.LedCtrlRaw(205,2,0)
        self.lp.LedCtrlRaw(206,0,2)
        self.lp.LedCtrlRaw(207,2,2)
```

Everything in “frame” becomes inaccessible for the player to touch, also lights up the frame of the board (lines 6-18)

Switches on the LEDs for the menu buttons

buttons.py (sets the actions of the menu buttons):

```python
from board import Board

class Buttons:
    def __init__(self):
        pass
    def button_actions(self,events,lp,frame,X,O):
        self.X=X
        self.O=O
        if events == [200, True]:
            lp.Reset()
            lp.Close()
            print("bye ...")
        if events == [201, True]:
            lp.Reset()
            self.board = Board(lp, frame)
            self.X.clear()
            self.O.clear()
        if events == [205, True]:
            lp.LedCtrlString("HARD", 3,0, direction=-1, waitms=20)
            lp.Reset()
            self.board = Board(lp, frame)
            self.X.clear()
            self.O.clear()
        if events == [206, True]:
            lp.LedCtrlString("EASY", 0,3, direction=-1, waitms=20)
            lp.Reset()
            self.board = Board(lp, frame)
            self.X.clear()
            self.O.clear()
        if events == [207, True]:
            lp.LedCtrlString("1v1", 3,3, direction=-1, waitms=20)
            lp.Reset()
            self.board = Board(lp, frame)
            self.X.clear()
            self.O.clear()
```

Turn off/reset

Change modes

Whenever the program resets it resets the board and it clears the registered X/O moves

player_move.py (handles the player movement):

```python
#X and O moving
def move(self,lp,frame,X,O,events,dif):
    self.lp=lp
    self.frame=frame
    self.X=X
    self.O=O
    self.events=events
    if not X and not O:
        self.ai_move.avail_move = self.moves.copy()
    for i in range(self.range):
        if self.Xturn:
            if i in self.moves and i not in self.X and i not in self.O:
                if self.events == [i, True]:
                    self.lp.LedCtrlRaw(i, 3, 0)
                    self.lp.LedCtrlRaw(i+1, 3, 0)
                    self.lp.LedCtrlRaw(i + 16, 3, 0)
                    self.lp.LedCtrlRaw(i + 16+1, 3, 0)
                    self.X.append(i)
                    self.ai_move.avail_move.remove(i)
                    self.Xturn=False
        elif not self.Xturn and dif==1 or dif ==2:
            if self.ai_move.avail_move != []:
                j = AI_move.ai_move(self.ai_move,self.X,self.O,dif)
                if j not in self.X and j not in self.O and Check.checkWIN(self.check,self.X) != True :
                    self.lp.LedCtrlRaw(j, 2, 2)
                    self.lp.LedCtrlRaw(j + 1, 2, 2)
                    self.lp.LedCtrlRaw(j + 16, 2, 2)
                    self.lp.LedCtrlRaw(j + 16 + 1, 2, 2)
                    self.O.append(j)
                    self.ai_move.avail_move.remove(j)
                    self.Xturn = True
        elif not self.Xturn and dif==3:
            if i in self.moves and i not in self.X and i not in self.O:
                if self.events == [i, True]:
                    self.lp.LedCtrlRaw(i, 2,2)
                    self.lp.LedCtrlRaw(i + 1, 2, 2)
                    self.lp.LedCtrlRaw(i + 16, 2, 2)
                    self.lp.LedCtrlRaw(i + 16 + 1, 2, 2)
                    self.O.append(i)
                    self.ai_move.avail_move.remove(i)
                    # print(self.ai_move.avail_move)
                    self.Xturn = True
```

If its X's (Player 1's) turn the player can click on the top left corner of the place that they want to move. The move is added to X and the 2x2 block that represents the move lights up

If the difficulty is 1 or 2 means that O is an AI. The AI makes its move by calling ai_move and does the same actions as above

If the difficulty is 3 it means O is a 2nd player so the behaviour is similar to player 1

ai_move.py (handles the AI movement with minmax):

```python
#ai move
def ai_move(self,X,O,dif):
    best_score= -math.inf
    move=None
    bombMove=None
    for i in self.avail_move:
        nextA=copy.deepcopy(self.avail_move)
        O.append(i)
        #bombmove is used on the easy difficulty to check if the final move wins
        if Check.checkWIN(self.check, O):
            bombMove=i
        nextA.remove(i)
        score=self.minimax(X,O,nextA,0,False,-math.inf,math.inf)
        # print(i,":",score)
        O.remove(i)
        nextA.append(i)
        if score>best_score:
            best_score=score
            move=i
    #return best move :minmax
    if dif==1:
        return move
    #easy AI: first move always random, final move always optimal, inbetween has 33% chance of choosing randomly
    elif dif==2:
        if O==[]:
            return self.avail_move[random.randrange(0,len(self.avail_move))]
        if bombMove!= None:
            return bombMove
        else:
            ranMove=[move,move,self.avail_move[random.randrange(0,len(self.avail_move))]]
            return random.choice(ranMove)
```

Tests the next move and saves the score using the minimax algorithm

Hard: returns best move

Easy: first move always random, last move always optimal. In between it has a 33% chance of picking randomly

```python
#minmax algorith with alpha beta pruning
    def minimax(self,X,O,avail_move,depth,isMax,alpha,beta):
        if Check.checkWIN(self.check,X):
            skore=-10
            return skore
        if Check.checkWIN(self.check,O):
            skore=10
            return skore
        if len(X)==5 or len(O)==5:
            skore=0
            return skore
        else:
            if isMax:
                best_score = -math.inf
                for i in avail_move:
                    nextA= copy.deepcopy(avail_move)
                    nextO = copy.deepcopy(O)
                    nextO.append(i)
                    nextA.remove(i)
                    score = self.minimax(X,nextO,nextA,depth+1,False,alpha,beta)
                    nextO.remove(i)
                    nextA.append(i)
                    if score > best_score:
                        best_score = score
                    alpha= max(alpha,score)
                    if beta<=alpha:
                        break
                return best_score
            else:
                best_score = math.inf
                for i in avail_move:
                    nextX = copy.deepcopy(X)
                    nextA = copy.deepcopy(avail_move)
                    nextX.append(i)
                    nextA.remove(i)
                    score = self.minimax(nextX,O,nextA,depth+1,True,alpha,beta)
                    nextX.remove(i)
                    nextA.append(i)
                    if score < best_score:
                        best_score = score
                    beta= min(score,beta)
                    if beta<=alpha:
                        break
                return best_score
```

Minmax algorithm with alpha beta pruning. The method returns the score if it detects a win/draw. If not it calculates the next move using recursion and the alpha beta variables to avoid unnecessary moves that slow down the program

check.py (checks if terminal state is reached and handles it appropriately):

```
        #winning cases
        self.winH1 = [0, 3, 6]
        self.winH2 = [48, 51, 54]
        self.winH3 = [96, 99, 102]
        self.winV1 = [0, 48, 96]
        self.winV2 = [3, 51, 99]
        self.winV3 = [6, 54, 102]
        self.winD1 = [0, 51, 102]
        self.winD2 = [6, 51, 96]
    #checking if win state is reached
    def checkWIN(self,symbol):
        self.symbol = symbol
        if all(i in symbol for i in self.winH1):
            return True
        if all(i in symbol for i in self.winH2):
            return True
        if all(i in symbol for i in self.winH3):
            return True
        if all(i in symbol for i in self.winV1):
            return True
        if all(i in symbol for i in self.winV2):
            return True
        if all(i in symbol for i in self.winV3):
            return True
        if all(i in symbol for i in self.winD1):
            return True
        if all(i in symbol for i in self.winD2):
            return True

    def fill(self,array):
        array2=copy.deepcopy(array)
        for i in range(0,3):
            array2.append(array[i]+1)
            array2.append(array[i] + 16)
            array2.append(array[i] + 17)
        return array2

    def returnWIN(self,symbol):
        self.symbol = symbol
        if all(i in symbol for i in self.winH1):
            return self.winH1
        if all(i in symbol for i in self.winH2):
            return self.winH2
        if all(i in symbol for i in self.winH3):
            return self.winH3
        if all(i in symbol for i in self.winV1):
            return self.winV1
        if all(i in symbol for i in self.winV2):
```

Winning states

Checks if win is reached, returns true

“fill” fills the winning moves’ block. “returnWIN” returns the winning state’s values. This is implemented to visually show the winning state

```
#actions after win case
def check_cases(self,lp,frame,X,O):
    self.lp=lp
    self.frame=frame
    self.X=X
    self.O=O
    if self.checkWIN(self.X) == True:
        filled = self.fill(self.returnWIN(self.X))
        self.lp.LedAllOn(0)
        for i in range(len(filled)):
            self.lp.LedCtrlRaw(filled[i],3,0)
        time.sleep(1)
        self.lp.LedCtrlString("X WINS!", 3, 0, direction=-1, waitms=50)
        self.lp.Reset()
        self.board = Board(self.lp, self.frame)
        self.X.clear()
        self.O.clear()
    if self.checkWIN(self.O) == True:
        self.lp.LedAllOn(0)
        filled = self.fill(self.returnWIN(self.O))
        for i in range(len(filled)):
            self.lp.LedCtrlRaw(filled[i],2,2)
        time.sleep(1)
        self.lp.LedCtrlString("O WINS!", 3, 3, direction=-1, waitms=50)
        self.lp.Reset()
        self.board = Board(self.lp, self.frame)
        self.X.clear()
        self.O.clear()
    if len(self.X)==5 or len(self.O)==5:
        time.sleep(1)
        self.lp.LedCtrlString("DRAW", 0, 3, direction=-1, waitms=50)
        self.lp.Reset()
        self.board = Board(self.lp, self.frame)
        self.X.clear()
        self.O.clear()
```

Checks win for X and O. If the length of registered moves reaches 5 on either it means the board is full and no winner has been found so its a draw.

For every case the board shows the terminal state, displays the appropriate message, and resets the board for the next game