PROJECT REPORT:-

NAME :- RONIT

SECTION :- K18KK

ROLL NO :- 28

COURSE TITLE :- INT-40

TOPIC :- SNAKE GAME USING AI

GITHUB LINK :- https://github.com/ronit-lab/SNAKE-GMAE/tree/master

TABLE OF CONTENT :-

ABSTRACT:-

Snake game is one of the most popular game since childhood .But it was developed earlier in such a way that player use to play this game by its own. As technology is advancing day by day AI is introduced into it . The same way I have developed this game with basic implementation of AI such that snake itself find an optimal path for its food.

I have created a snake game using AI with the help of pygame. In this game apple (snake food) is found in the random direction and snake position is also set as random as soon as snake food appear on screen. Snake find the best possible path in order to quickly reach towards its food. Following function is used to implement this reinforcement using AI .In this game my implementation consist of snake which is moving on a square board in search of apples without eating or biting itself.

Once the snake eats its apple, a new apple appears in a free position and the snake increases its length by one unit. When snake has no choice left with him other than eating or biting itself, the game comes to end. In this implementation, score is calculated as the number of apples eaten by snake or the length increased by snake.

1. Goal: - our goal is that snake tries to eat as many apples possible with finite number of steps.
2. Priority:-The priority of the snake is that it should not bite itself and second one to increase the score determined by the length of the snake.
3. Direction: - In this snake move in left , right , down or up direction.
4. The snake grows by unit one when he eats his apple.
   The growth of snake is indicated by the tip of tail that occupies the square on which the apple was.
5. The board is in the size of square
6. After an apple is eaten by the snake, another apple is placed randomly with probability that it will not occupied by the snake.

RELATED WORK OR ANALYSIS :-

Before i working on this project i search internet to find any idea related to this project. But i found that either they have used BFS or A* algorithm to find the path

But i analysed that **BFS(Best First Search)** strategy has allow only one move horizon and only considers moving the snake to the position on the board that appears close to the board i.e apple. This method is found suitable upto **only four apples** after that there will be probability that it can **bite itself** once it gets longer,this method is **not optimal** after the snake has eaten more than **four apples**.

A* search with BFS technique is optimal for this game :-

A* strategy allow multiple move horizon . Before taking action, it considers not only where the goal is and how far it is, but also the current state it has searched so far. The algorithm is guaranteed to find optimal path to the apple if one exists. However, the maximum number of nodes expanded is limited. This makes the algorithm stop if a path to goal cannot be found (for any reason). In case the maximum nodes bound is reached, the algorithm will switch back to Best First Search for that iteration.

So this game is implemented using BFS as well as A*.

CODE AND IMPLEMENTATION :-

```
import random, pygame, sys, operator, os, time

from pygame.locals import *

F = 70

WIDTH = 1800

HEIGHT = 1200

SIZE = 60

assert WIDTH% SIZE== 0

assert HEIGHT% SIZE== 0

windowwidth = int(WIDTH / SIZE)

windowheigth = int(HEIGHT / SIZE)

WHITE     = (255, 255, 255)

BLACK     = (  0,   0,   0)

RED       = (255,   0,   0)

GREEN     = (  0, 255,   0)
```

```
DARKGREEN = (  0, 155,   0)
YELLOW  = ( 40,  40,  40)
DARKGRAY  = ( 40,  40,  40)
BGCOLOR = BLACK

UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'
HEAD = 0

def main():
    global A, DISPLAY, FONT
    global window,softwindow
    window = []
    softwindow = []
    softwindow = findSoftWall()
    window = findWall()
```

```
    pygame.init()
    A = pygame.time.Clock()
    DISPLAY = pygame.display.set_mode((WIDTH, HEIGHT))
    FONT = pygame.font.Font('freesansbold.ttf', 18)
    pygame.display.set_caption('SNAKE GAME')
    showStartScreen()


    while True:
        runGame()
        showGameOverScreen()


def runGame():
    global status
    status = False
    statusCount = -1
    xaxis = 5
```

```
yaxis = 0
snake = [{'x': xaxis+ 6, 'y': yaxis},
         {'x': xaxis + 5, 'y': yaxis},
         {'x': xaxis+ 4, 'y': yaxis},
         ]

direction = RIGHT
directionList = [RIGHT]
PATH = []

apple = {'x': xaxis+8,    'y': yaxis}
lastApple = {'x':xaxis-1,   'y': yaxis -1}
PATH = calculatePath(snake,apple,True)
directionList = calcDirection(PATH)
lastWall = 0

while True:
   for event in pygame.event.get():
```

```
            if event.type == QUIT:
                terminate()
            elif event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    terminate()

        if snake[HEAD]['x'] == -1 or snake[HEAD]['x'] == windowwidth or snake[HEAD]['y'] == -1 or
snake[HEAD]['y'] == windowheigth:
            terminate()
            return

        for snakeBody in snake[1:]:
            if snakeBody['x'] == snake[HEAD]['x'] and snakeBody['y'] == snake[HEAD]['y']:
                terminate()
                return
        if snake[HEAD]['x'] == apple['x'] and snake[HEAD]['y'] == apple['y']:

            lastApple = apple
            apple = getlocation(snake)
```

```
    drawApple(apple,lastApple)
    PATH = calculatePath(snake,apple,True)
    if not PATH:
      status = True
      statusCount = 10000
    elif PATH == 'stall':
      status = True
      statusCount = int(len(snake)/2)
    else:
      directionList = calcDirection(PATH)
else:
    del snake[-1]


lastDirection = direction


if status and not directionList:
```

```
onlyDirection = calcOnlyDirection(snake)
if onlyDirection and onlyDirection == lastDirection:
  directionList.append(onlyDirection)
  print('only direction:', direction)
else:
  if safeToGo(snake,direction,lastWall):

    directionList.append(direction)
  elif (not findNewHead(direction,snake) in snake) or (findNewHead(direction,snake) in window):
    directionList.append(direction)
  else:
    lastDirection = direction

    PATH = calculatePath(snake,apple,False)
    if PATH != [] and PATH != 'stall':
      status = False
      statusCount = -1
      directionList = calcDirection(PATH)
```

```
  else:
    if checkLastWall(snake):
      lastWall = checkLastWall(snake)
    directionList.extend(findBetterDirection(snake,direction,lastWall))
    if calcArea(findNewHead(directionList[0],snake), snake, lastWall)<3:
      directionList = [lastDirection]

statusCount = statusCount - 1

if statusCount < 1:

  status = False
  prevLastWall = lastWall
  lastWall = 0
  directionList.append(lastDirection)
  PATH = calculatePath(snake,apple,True)
  if not PATH:
    status = True
```

```
        statusCount = 10000
        lastWall = prevLastWall
      elif PATH == 'stall':
        status = True
        statusCount = int(len(snake)/2)
        lastWall = prevLastWall
      else:
        directionList = calcDirection(PATH)
nextHead = findNewHead(directionList[0],snake)

if status:
  if AreaIsTooSmall(windowwidth,nextHead, snake, lastWall):
    lastWall = 0
    directionList = findNextDirection(snake, directionList[0],0)
    print('almost died, recalcualting...',snake[0],directionList)



direction = directionList.pop(0)
```

```
        newHead = findNewHead(direction, snake)
        snake.insert(0, newHead)
        DISPLAY.fill(BGCOLOR)
        drawGrid()
        drawWorm(snake)
        drawApple(apple,lastApple)
        drawScore(len(snake) - 3)
        pygame.display.update()
        A.tick(F)


def calcOnlyDirection(snaky):
    count = 4
    ways = getNeighborhood(snaky[0])
    theWay = 0
    for z in ways:
      if z in snaky:
        count = count - 1
      else:
```

```
        theWay = z
    if count == 1:
      return calcDirection([snaky[0],theWay])
    else:
      return 0

def getNextwindow(lastWall):
    walls = []

    loopcount = 0
    for _ in range(windowheigth):
      if lastWall == RIGHT:
        walls.append({'x':0, 'y':loopcount})
      if lastWall == LEFT:
        walls.append({'x':windowwidth-1, 'y':loopcount})
      loopcount = loopcount + 1

    loopcount = 0
```

```
    for _ in range(windowwidth):
      if lastWall == DOWN:
        walls.append({'x':loopcount, 'y':0})
      if lastWall == UP:
        walls.append({'x':loopcount, 'y':windowheigth-1})
      loopcount = loopcount + 1
    return walls

def safeToGo(snaky,direction,lastWall):
    listOfNo = window + snaky
    listOfNo.extend(getNextwindow(lastWall))
    head = snaky[0]
    forward = snaky[0]
    forwardLeft = snaky[0]
    forwardRight = snaky[0]
    left = snaky[0]
    right = snaky[0]
    if direction == UP:
```

```
    newHead = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] - 1}
    forward = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] - 2}
    forwardLeft = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y'] - 1}
    forwardRight = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y'] - 1}
    left = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']}
    right = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']}
elif direction == DOWN:
    newHead = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] + 1}
    forward = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] + 2}
    forwardLeft = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y'] + 1}
    forwardRight = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y'] + 1}
    left = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']}
    right = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']}
elif direction == LEFT:
    newHead = {'x': snaky[HEAD]['x'] - 1, 'y': snaky[HEAD]['y']}
    forward = {'x': snaky[HEAD]['x'] - 2, 'y': snaky[HEAD]['y']}
    forwardLeft = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y'] + 1}
    forwardRight = {'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y'] - 1}
```

```
        left = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']+1}
        right = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']-1}
    elif direction == RIGHT:
        newHead = {'x': snaky[HEAD]['x'] + 1, 'y': snaky[HEAD]['y']}
        forward = {'x': snaky[HEAD]['x'] + 2, 'y': snaky[HEAD]['y']}
        forwardLeft = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y'] - 1}
        forwardRight = {'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y'] + 1}
        left = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']-1}
        right = {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']+1}



    if (forwardLeft in listOfNo and not left in listOfNo) or (forwardRight in listOfNo and not right in
listOfNo):


      return False
    if newHead in listOfNo:
      return False
    waysToGo = []
    waysToGo = getNeighborhood(newHead)
```

```
    count = len(waysToGo)
    for z in waysToGo:
      if z in listOfNo:
        count = count - 1
    if count < 1:
      return False
    elif count < 2 and not (forward in listOfNo):
      return False
    else:
      return True

def checkLastWall(snaky):
    x = snaky[0]['x']
    y = snaky[0]['y']
    if x == 0:
      return LEFT
    elif x == windowwidth - 1:
      return RIGHT
```

```
    elif y == 0:
      return UP
    elif y == windowheigth -1:
      return DOWN
    else:
      return 0


def checkSmartTurn(snaky,listOfNo,direction1,direction2):
    if direction1 == UP or direction1 == DOWN:
      if direction2 == RIGHT:
        if {'x': snaky[HEAD]['x']+3, 'y': snaky[HEAD]['y']} in listOfNo and (not {'x': snaky[HEAD]['x']+2, 'y': snaky[HEAD]['y']} in listOfNo):
          return True
        else:
          return False
      if direction2 == LEFT:
        if {'x': snaky[HEAD]['x']-3, 'y': snaky[HEAD]['y']} in listOfNo and (not {'x': snaky[HEAD]['x']-2, 'y': snaky[HEAD]['y']} in listOfNo):
          return True
```

```
        else:
          return False
    if direction1 == LEFT or direction1 == RIGHT:
      if direction2 == UP:
        if {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']-3} in listOfNo and (not {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']-2} in listOfNo):
          return True
        else:
          return False
      if direction2 == DOWN:
        if {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']+3} in listOfNo and (not {'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']+2} in listOfNo):
          return True
        else:
          return False


def findBetterDirection(snaky, direction,lastWall):
    listOfNo = list(snaky)
    smartTurn = False
```

```
if direction == UP:
    areaLeft = calcArea({'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    areaRight = calcArea({'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    if areaLeft == 0 and areaRight == 0:
      return [direction]
    areaStraight = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']-1},snaky,lastWall)
    maxArea = max(areaLeft,areaRight,areaStraight)
    print ('Options:', 'left:',areaLeft,'right:',areaRight,'straight:',areaStraight)
    if maxArea == areaStraight:
      return [direction]
    elif maxArea == areaLeft:
      if checkSmartTurn(snaky,listOfNo,direction,LEFT):
        print('Smart Turn Enabled')
        return [LEFT, LEFT]
      else:
        return [LEFT, DOWN]
    else:
      if checkSmartTurn(snaky,listOfNo,direction,RIGHT):
```

```
      print('Smart Turn Enabled')
      return [RIGHT, RIGHT]
    else:
      return [RIGHT,DOWN]

if direction == DOWN:
    areaLeft = calcArea({'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    areaRight = calcArea({'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    if areaLeft == 0 and areaRight == 0:
      return [direction]
    areaStraight = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y']+1},snaky,lastWall)
    maxArea = max(areaLeft,areaRight,areaStraight)
    print ('Options:','left:',areaLeft,'right:',areaRight,'straight:',areaStraight)
    if maxArea == areaStraight:
      return [direction]
    elif areaLeft == maxArea:
      if checkSmartTurn(snaky,listOfNo,direction,LEFT):
        print('Smart Turn Enabled')
```

```
            return [LEFT, LEFT]
          else:
            return [LEFT, UP]
        else:
          if checkSmartTurn(snaky,listOfNo,direction,RIGHT):
            print('Smart Turn Enabled')
            return [RIGHT, RIGHT]
          else:
            return [RIGHT,UP]

    elif direction == LEFT:
        areaUp = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] - 1},snaky,lastWall)
        areaDown = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] + 1},snaky,lastWall)
        if areaUp == 0 and areaDown == 0:
          return [direction]
        areaStraight = calcArea({'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']},snaky,lastWall)
        maxArea = max(areaStraight,areaUp,areaDown)
        print ('Options:','up:',areaUp,'down:',areaDown,'straight:',areaStraight)
```

```
        if maxArea == areaStraight:
          return [direction]
        elif maxArea == areaUp:
          if checkSmartTurn(snaky,listOfNo,direction,UP):
            print('Smart Turn Enabled')
            return [UP, UP]
          else:
            return [UP,RIGHT]
        else:
          if checkSmartTurn(snaky,listOfNo,direction,DOWN):
            print('Smart Turn Enabled')
            return [DOWN, DOWN]
          else:
            return [DOWN,RIGHT]

      elif direction == RIGHT:
        areaUp = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] - 1},snaky,lastWall)
        areaDown = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] + 1},snaky,lastWall)
```

```
if areaUp == 0 and areaDown == 0:
  return [direction]
areaStraight = calcArea({'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']},snaky,lastWall)
maxArea = max(areaStraight,areaUp,areaDown)
print ('Options:','up:',areaUp,'down:',areaDown,'straight:',areaStraight)
if maxArea == areaStraight:
  return [direction]
elif areaUp ==maxArea:
  if checkSmartTurn(snaky,listOfNo,direction,UP):
    print('Smart Turn Enabled')
    return [UP, UP]
  else:
    return [UP,LEFT]
else:
  if checkSmartTurn(snaky,listOfNo,direction,DOWN):
    print('Smart Turn Enabled')
    return [DOWN, DOWN]
  else:
```

```
        return [DOWN,LEFT]


def findNextDirection(snaky, direction,lastWall):
    listOfNo = list(snaky)
    areaLeft = calcArea({'x': snaky[HEAD]['x']-1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    areaRight = calcArea({'x': snaky[HEAD]['x']+1, 'y': snaky[HEAD]['y']},snaky,lastWall)
    areaUp = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] - 1},snaky,lastWall)
    areaDown = calcArea({'x': snaky[HEAD]['x'], 'y': snaky[HEAD]['y'] + 1},snaky,lastWall)
    maxArea = max(areaLeft,areaRight,areaUp,areaDown)
    if maxArea == areaUp:
      return [UP]
    elif maxArea == areaDown:
      return [DOWN]
    elif maxArea == areaLeft:
      return [LEFT]
    else:
      return [RIGHT]
```

```
def calcArea(point, snaky, lastWall):
    nextWall = getNextwindow(lastWall)
    if point in snaky or point in window or point in nextWall:
      return 0
    tailBonus = 0
    q = []
    searchPoints = []
    searchPoints.append(point)
    while (searchPoints):
      i = searchPoints.pop()
      for z in getNeighborhood(i):
        if not z in q:
          if not (z in snaky or z in window or point in nextWall):
            searchPoints.append(z)
        if z == snaky[-1]:
          tailBonus = 200
      q.append(i)
    return len(q)+tailBonus
```

```
def AreaIsTooSmall(bound,point, snaky, lastWall):
    nextWall = getNextwindow(lastWall)
    if point in snaky or point in window or point in nextWall:
      return True
    tailBonus = 0
    q = []
    searchPoints = []
    searchPoints.append(point)
    while (searchPoints):
      i = searchPoints.pop()
      for z in getNeighborhood(i):
        if not z in q:
          if not (z in snaky or z in window or point in nextWall):
            searchPoints.append(z)
        if z == snaky[-1]:
          tailBonus = 200
      q.append(i)
```

```
    if (len(q) + tailBonus) > bound:
      return False
  return True

def calcCost(point,snaky):
  print ('calculating cost of point', point)
  neibors = getNeighborhood(point)
  for z in neibors:
    if z in snaky[1:]:
      return snaky.index(z)
  return 999

def calcDirection(path):
  '''Converting point-path to step by step direction'''
  endpoint = path[0]
  directions = []
  nextDirection = ''
  for presentpoint in path:
```

```
  if (presentpoint['x'] > endpoint['x']):
    nextDirection = RIGHT
  elif (presentpoint['x'] < endpoint['x']):
    nextDirection = LEFT
  else:
    if (presentpoint['y'] > endpoint['y']):
      nextDirection = DOWN
    elif (presentpoint['y'] < endpoint['y']):
      nextDirection = UP
    else:

      continue

  endpoint = presentpoint
  directions.append(nextDirection)

return directions
```

```
def calculatePath(snaky,apple,softCalculation):
  oldWorm = list(snaky)


  path = mainCalculation(snaky,apple,softCalculation)
  if not path:
    return []
  else:
    pathCopy = list(path)
    pathCopy.reverse()
    newWorm = pathCopy + oldWorm
    pathOut = mainCalculation(newWorm,newWorm[-1],False)
    if not pathOut:
      print('No path out, dont go for apple')
      return 'stall'
    else:
      return path

def mainCalculation(snaky,apple,softCalculation):
```

```
pointsToPath= []
discoverEdge = []
newPoints = []
exhaustedPoints = []
numberOfPoints = 1
findingPath = True
listOfNo = getList(snaky)
softListOfNo = getSoftListOfNo(snaky)
softListOfNo.extend(softwindow)
discoverEdge.append(snaky[0])
exhaustedPoints.append(snaky[0])
endpoint = discoverEdge[-1]
pointsToPath.append(endpoint)

if (apple in softwindow) or (apple in softListOfNo):
  softCalculation = False
```

```
while(findingPath and softCalculation):
  endpoint = discoverEdge[-1]
  newPoints = getNeighborhood(endpoint)
  newPoints = sorted(newPoints, key = lambda k: calcDistance(k,apple), reverse = True)
  numberOfPoints = len(newPoints)
  for point in newPoints:
    if point in softListOfNo:


      numberOfPoints = numberOfPoints -1
    elif point in exhaustedPoints:


      numberOfPoints = numberOfPoints -1
    else:
      discoverEdge.append(point)
      pointsToPath.append(endpoint)
      exhaustedPoints.append(endpoint)
      #print (point)
    #exhaustedPoints.append(point)
```

```
  if numberOfPoints == 0:
   #backtrack
   exhaustedPoints.append(discoverEdge.pop())
   exhaustedPoints.append(pointsToPath.pop())
  if apple in discoverEdge:
   findingPath = 0
  if not discoverEdge:
   softCalculation = False
   break


if not softCalculation:
  pointsToPath= []
  discoverEdge = []
  newPoints = []
  exhaustedPoints = []
  numberOfPoints = 1
  findingPath = True
```

```
listOfNo = getList(snaky)
discoverEdge.append(snaky[0])
exhaustedPoints.append(snaky[0])
endpoint = discoverEdge[-1]
pointsToPath.append(endpoint)


while(findingPath):
  endpoint = discoverEdge[-1]
  newPoints = getNeighborhood(endpoint)
  newPoints = sorted(newPoints, key = lambda k: calcDistance(k,apple), reverse = True)
  numberOfPoints = len(newPoints)
  for point in newPoints:
    if point in listOfNo:

      numberOfPoints = numberOfPoints -1
    elif point in exhaustedPoints:
```

```
    numberOfPoints = numberOfPoints -1
  else:
    discoverEdge.append(point)
    pointsToPath.append(endpoint)
    exhaustedPoints.append(endpoint)
    #print (point)
  #exhaustedPoints.append(point)
if numberOfPoints == 0:
  #backtrack
  exhaustedPoints.append(discoverEdge.pop())
  exhaustedPoints.append(pointsToPath.pop())
if apple in discoverEdge:
  findingPath = 0
if not discoverEdge:

  return []
```

```
  pointsToPath.append(apple)
  return pointsToPath

def getNeighborhood(point):
  neighborhood = []
  if point['x'] < windowwidth:
    neighborhood.append({'x':point['x']+1,'y':point['y']})
  if point['x'] > 0:
    neighborhood.append({'x':point['x']-1,'y':point['y']})
  if point['y'] < windowheigth:
    neighborhood.append({'x':point['x'],'y':point['y']+1})
  if point['y'] >0:
    neighborhood.append({'x':point['x'],'y':point['y']-1})
  return neighborhood

def calcDistance(point, apple):
```

```
  distance = abs(point['x'] - apple['x']) + abs(point['y'] - apple['y'])
  return distance

def getSoftListOfNo(snaky):
  listOfNo = []
  listOfNo.extend(getWormSurroundings(snaky))
  #listOfNo.extend(softwindow)
  #remove duplicates
  return listOfNo


def getWormSurroundings(snaky):
  listOfNo = []
  headx = snaky[0]['x']
  heady = snaky[0]['y']
  count = 0
  for z in snaky:
    if count == 0:
```

```
    listOfNo.append(z)
  else:
    dist = abs (z['x'] - headx) + abs(z['y']-heady)
    countFromBehind = len(snaky) - count
    if dist < (countFromBehind+1):
      listOfNo.append(z)
      listOfNo.append({'x':z['x']+1,'y':z['y']})
      listOfNo.append({'x':z['x']-1,'y':z['y']})
      listOfNo.append({'x':z['x'],'y':z['y']+1})
      listOfNo.append({'x':z['x'],'y':z['y']-1})
      listOfNo.append({'x':z['x']+1,'y':z['y']+1})
      listOfNo.append({'x':z['x']-1,'y':z['y']-1})
      listOfNo.append({'x':z['x']-1,'y':z['y']+1})
      listOfNo.append({'x':z['x']+1,'y':z['y']-1})
  count = count + 1
seen = set()
newList = []
for d in listOfNo:
```

```
    t = tuple(d.items())
    if t not in seen:
        seen.add(t)
        newList.append(d)
  return newList



def getList(snaky):
  listOfNo = []
  headx = snaky[0]['x']
  heady = snaky[0]['y']
  count = 0
  for z in snaky:
    dist = abs (z['x'] - headx) + abs(z['y']-heady)
    countFromBehind = len(snaky) - count
    count = count + 1
    if dist < (countFromBehind+1):
```

```
    listOfNo.append(z)
  listOfNo.extend(window)
  #print ('List of No Go:')
  #print (listOfNo)
  return listOfNo



def findWall():
  walls = []
  #append LEFT RIGHT walls
  loopcount = 0
  for _ in range(windowheigth):
    walls.append({'x':-1       , 'y':loopcount})
    walls.append({'x':windowwidth, 'y':loopcount})
    loopcount = loopcount + 1
  #append TOP BOTTOM walls
  loopcount = 0
  for _ in range(windowwidth):
```

```
    walls.append({'x':loopcount, 'y':-1})
    walls.append({'x':loopcount, 'y':windowheigth})
    loopcount = loopcount + 1
  #print (walls)
  return walls

def findSoftWall():
  walls = []
  #append LEFT RIGHT walls
  loopcount = 0
  for _ in range(windowheigth):
    walls.append({'x':0       , 'y':loopcount})
    walls.append({'x':windowwidth-1, 'y':loopcount})
    loopcount = loopcount + 1
  #append TOP BOTTOM walls
  loopcount = 0
  for _ in range(windowwidth):
    walls.append({'x':loopcount, 'y':0})
```

```
    walls.append({'x':loopcount, 'y':windowheigth-1})
    loopcount = loopcount + 1
  #print (walls)
  return walls


def drawEdgeOfDiscovery(points):
    for point in points:
        x = point['x'] * SIZE
        y = point['y'] * SIZE
        snakySegmentRect = pygame.Rect(x, y, SIZE, SIZE)
        pygame.draw.rect(DISPLAY, ORANGE, snakySegmentRect)
    endpointRect = pygame.Rect(points[-1]['x']*SIZE, points[-1]['y']*SIZE, SIZE, SIZE)
    pygame.draw.rect(DISPLAY, (255,255,255), snakySegmentRect)



def sectionBreak():
```

```
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')
  print('AAAAAAAAAAAAAAAAAAAA')



def pauseGame():
  pauseGame = True
  while (pauseGame):
    for event in pygame.event.get():
      if event.type == KEYDOWN:
        if event.key == K_SPACE:
          pauseGame = False

def oppositeDirection(direction):
```

```
    if direction == UP:
        return DOWN
    elif direction == DOWN:
        return UP
    elif direction == LEFT:
        return RIGHT
    elif direction == RIGHT:
        return LEFT

def findNewHead(direction,snake):
    if direction == UP:
        newHead = {'x': snake[HEAD]['x'], 'y': snake[HEAD]['y'] - 1}
    elif direction == DOWN:
        newHead = {'x': snake[HEAD]['x'], 'y': snake[HEAD]['y'] + 1}
    elif direction == LEFT:
        newHead = {'x': snake[HEAD]['x'] - 1, 'y': snake[HEAD]['y']}
    elif direction == RIGHT:
        newHead = {'x': snake[HEAD]['x'] + 1, 'y': snake[HEAD]['y']}
```

```
    return newHead


def drawPressKeyMsg():
    pressKeySurf = FONT.render('Press a key to play.', True, DARKGRAY)
    pressKeyRect = pressKeySurf.get_rect()
    pressKeyRect.topleft = (WIDTH - 200, HEIGHT - 30)
    DISPLAY.blit(pressKeySurf, pressKeyRect)


def press():
    if len(pygame.event.get(QUIT)) > 0:
        terminate()

    keyUpEvents = pygame.event.get(KEYUP)
    if len(keyUpEvents) == 0:
```

```
        return None
    if keyUpEvents[0].key == K_ESCAPE:
        terminate()
    return keyUpEvents[0].key


def showStartScreen():
    tilteFonting = pygame.font.Font('freesansbold.ttf', 100)
    titlesurfing1 = tilteFonting.render('SNAKE GAME!', True, WHITE, DARKGREEN)
    titlesurfing2 = tilteFonting.render('SNAKE GAME!', True, GREEN)

    angle1 = 0
    angle2 = 0
    while True:
        DISPLAY.fill(BGCOLOR)
        rotatingsurfing1 = pygame.transform.rotate(titlesurfing1, angle1)
        rotatingrectangle1 = rotatingsurfing1.get_rect()
        rotatingrectangle1.center = (WIDTH / 2, HEIGHT / 2)
```

```
DISPLAY.blit(rotatingsurfing1, rotatingrectangle1)

rotatingsurfing2 = pygame.transform.rotate(titlesurfing2, angle2)
rotatingrectangle2 = rotatingsurfing2.get_rect()
rotatingrectangle2.center = (WIDTH / 2, HEIGHT / 2)
DISPLAY.blit(rotatingsurfing2, rotatingrectangle2)

drawPressKeyMsg()

if press():
    pygame.event.get()
    return
pygame.display.update()
A.tick(F)
angle1 += 3
angle2 += 7
```

```
def terminate():
    print('YOU DIED!')
    pauseGame()
    pygame.quit()
    sys.exit()


def getlocation(snaky):
    location = {'x': random.randint(0, windowwidth - 1), 'y': random.randint(0, windowheigth - 1)}
    while(location in snaky):
        location = {'x': random.randint(0, windowwidth - 1), 'y': random.randint(0, windowheigth - 1)}
    return location


def showGameOverScreen():
    gamefonts = pygame.font.Font('freesansbold.ttf', 150)
    gamesurfing = gamefonts.render('Game', True, WHITE)
    overSurf = gamefonts.render('Over', True, WHITE)
```

```
    gameRect = gamesurfing.get_rect()
    overRect = overSurf.get_rect()
    gameRect.midtop = (WIDTH / 2, 10)
    overRect.midtop = (WIDTH / 2, gameRect.height + 10 + 25)

    DISPLAY.blit(gamesurfing, gameRect)
    DISPLAY.blit(overSurf, overRect)
    drawPressKeyMsg()
    pygame.display.update()
    pygame.time.wait(500)
    press()

    while True:
        if press():
            pygame.event.get()
            return

def drawScore(score):
```

```
    scoresurfing = FONT.render('Score: %s' % (score), True, WHITE)
    scorerectangleangle = scoresurfing.get_rect()
    scorerectangleangle.topleft = (WIDTH - 120, 10)
    DISPLAY.blit(scoresurfing, scorerectangleangle)

def drawWorm(snake):
    for coordinate in snake:
        x = coordinate['x'] * SIZE
        y = coordinate['y'] * SIZE
        snakySegmentRect = pygame.Rect(x, y, SIZE, SIZE)
        pygame.draw.rect(DISPLAY, DARKGREEN, snakySegmentRect)
        snakyrectangle = pygame.Rect(x + 4, y + 4, SIZE - 8, SIZE - 8)
        pygame.draw.rect(DISPLAY, GREEN, snakyrectangle)




def drawApple(coordinate,lastApple):
```

```
    x = coordinate['x'] * SIZE
    y = coordinate['y'] * SIZE
    appleRect = pygame.Rect(x, y, SIZE, SIZE)
    pygame.draw.rect(DISPLAY, RED, appleRect)



def drawGrid():
    for x in range(0, WIDTH, SIZE):
        pygame.draw.line(DISPLAY, YELLOW, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, SIZE):
        pygame.draw.line(DISPLAY, YELLOW, (0, y), (WIDTH, y))
```

IMPLEMENTATION :-

This project is written on python language in order to implement it you should

1) install pip install pygame
2) once it is install you simply run it in jupyter notebook or any other python platform
3) This project is divided into following functions
   def main( ):
   it consist of rungame( ) as well as show start screen fuction

4) rungame( ): define the directions in which snake can move and optimally find the path to its food whenever food arrives on screen without biting itself
5) showstartscreen( ): it is fuction which show riotating snake game screen with the help of transform.
6) defcalcOnlyDirection(): this will find the score . The score is depend upon how many apples snake eat and by how much amount its length increases.
7) drawGrid( ),drawApple( ),drawworm():- functions are used to draw grids that appear on screen . apple which is snake food is also built with the help of drawApple( ) whereas drawWorm( ) is use to built our snake.

FUNCTION COMPLEXITY :-

Function Name	**NLOC**	**Complexity**	**Token #**
main	16	2	92
runGame	104	33	711
calcOnlyDirection	13	4	61
getNextwindow	17	7	123
safeToGo	53	15	783
checkLastWall	13	5	59
checkSmartTurn	23	17	297
findBetterDirection	91	29	882

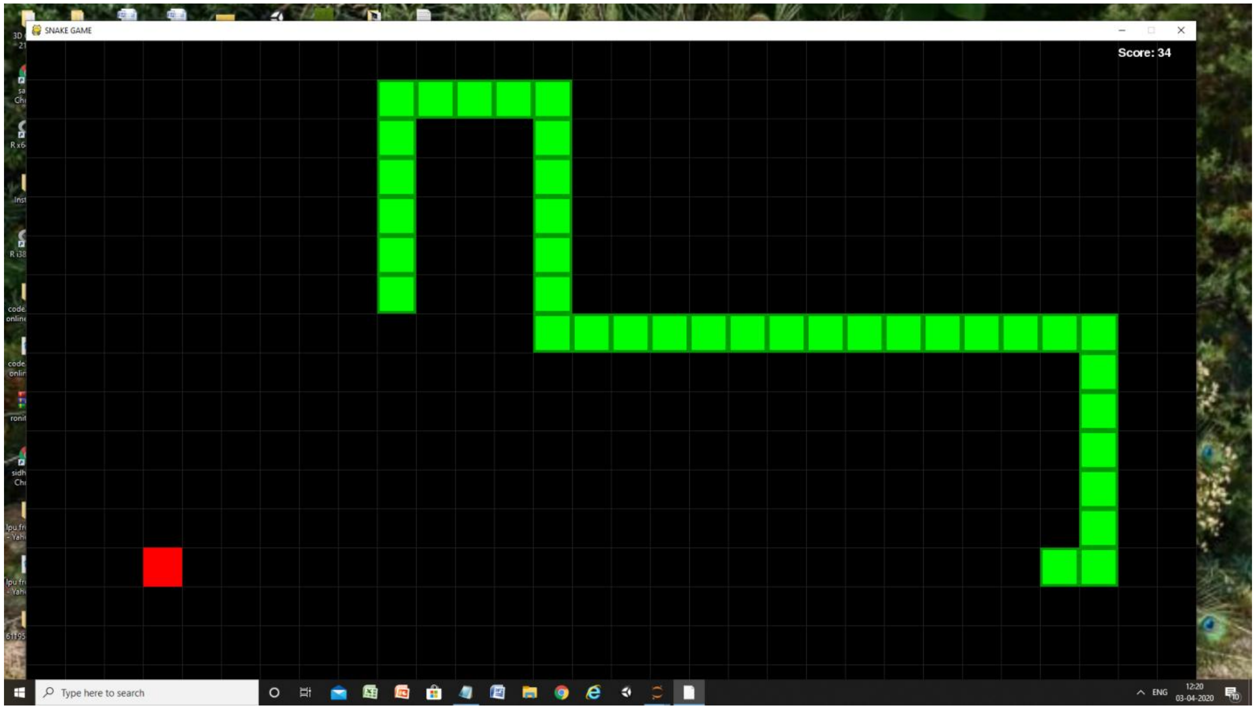
findNextDirection	15	4	188
calcArea	18	11	120
AreaIsTooSmall	20	12	131
calcCost	7	3	42
calcDirection	19	6	104
calculatePath	15	3	78
mainCalculation	73	19	432
getNeighborhood	11	5	131
calcDistance	3	1	36
getSoftListOfNo	4	1	20
getWormSurroundings	30	6	325
getList	13	3	92
findWall	13	3	100
findSoftWall	13	3	102
drawEdgeOfDiscovery	8	2	102
sectionBreak	8	1	32
pauseGame	7	5	40
oppositeDirection	9	5	33

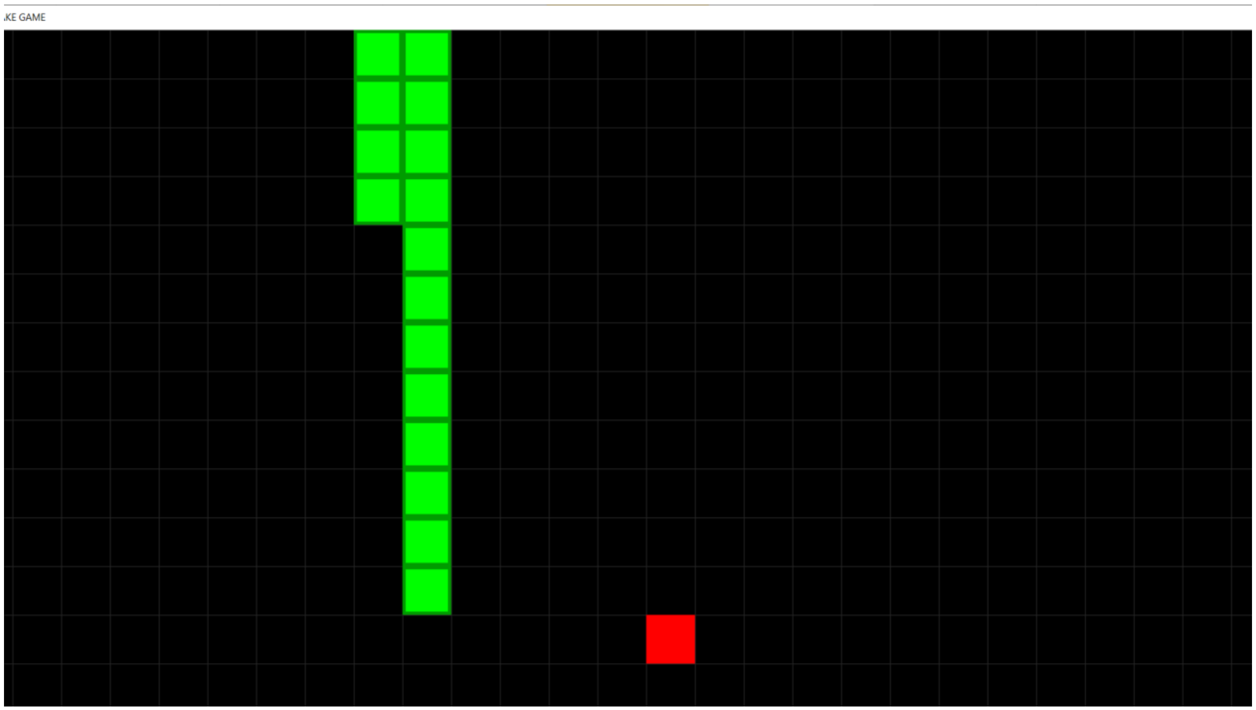
findNewHead	10	5	129
drawPressKeyMsg	5	1	44
press	9	4	62
showStartScreen	24	3	172
terminate	5	1	21
getlocation	5	2	72
showGameOverScreen	18	3	135
drawScore	5	1	47
drawWorm	8	2	86
drawApple	5	1	49
drawGrid	5	3	72

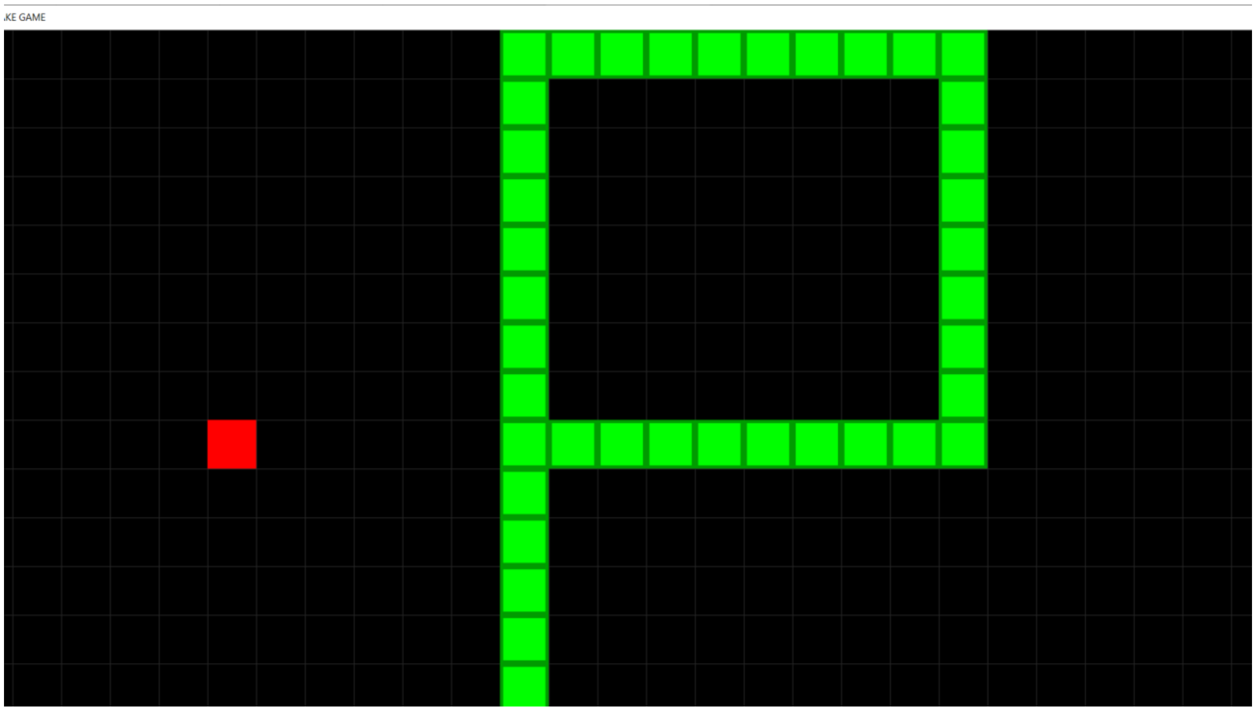
SCREENSHOT:-

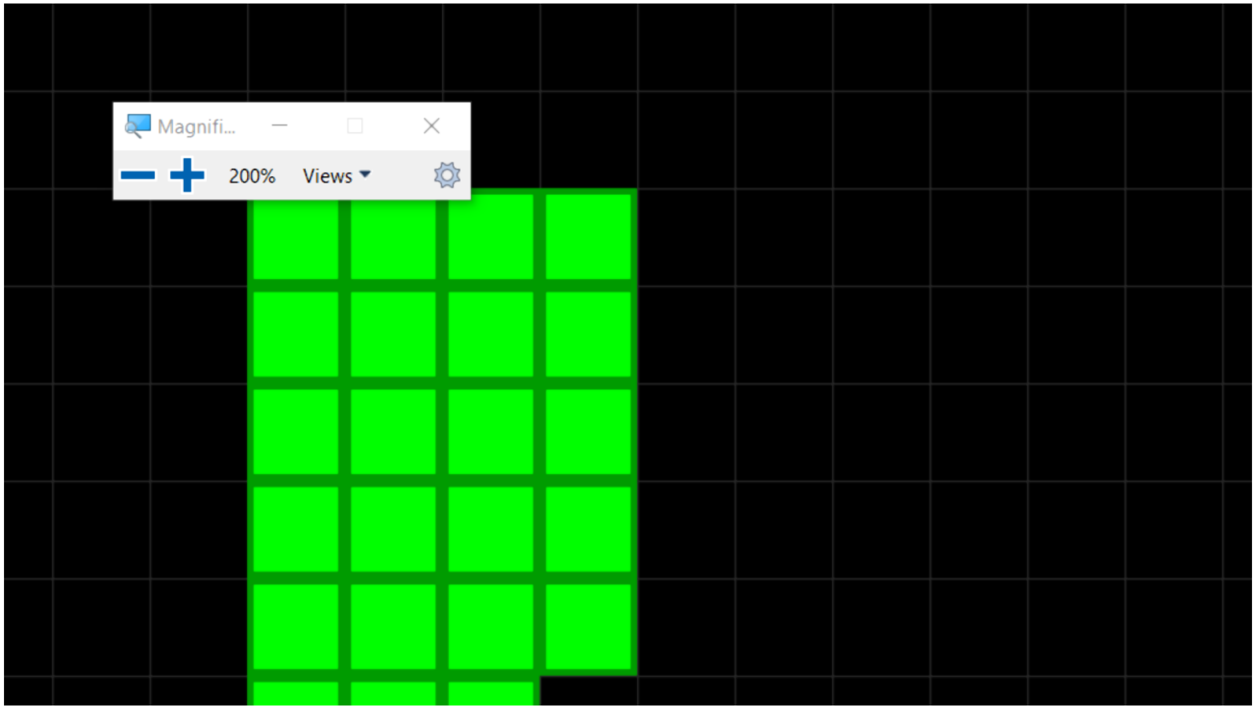
ROTATING SCREEN WITH SNAKE GAME

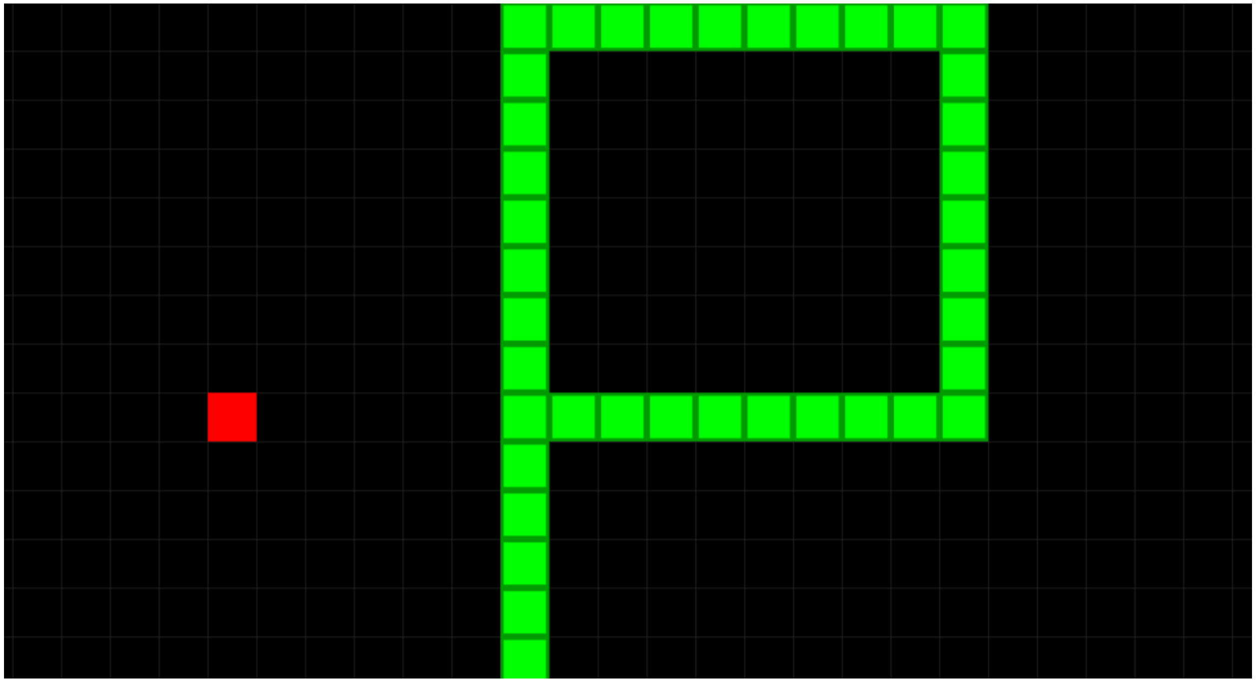
Apple occur randomly on screen and snake will eat eat apple and its length increased by one unit.

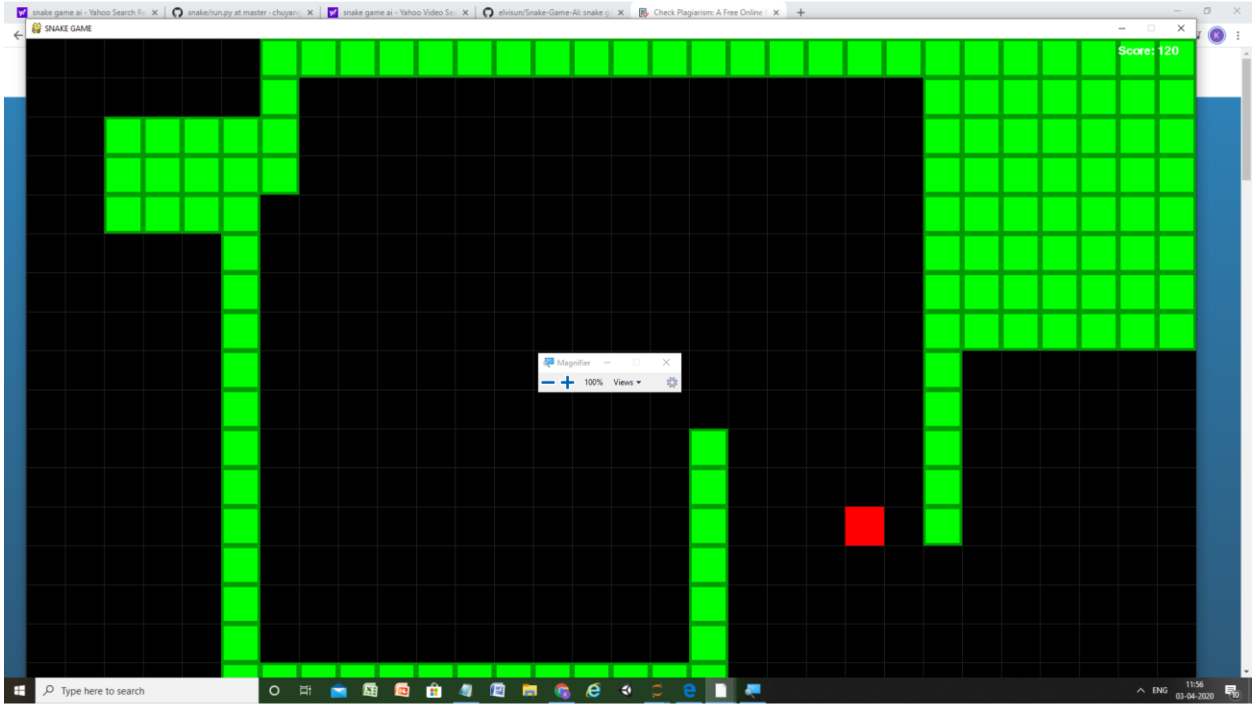

SNAKE GAME
Score: 34
Type here to search
ENG
12:20
03-04-2020


.KE GAME

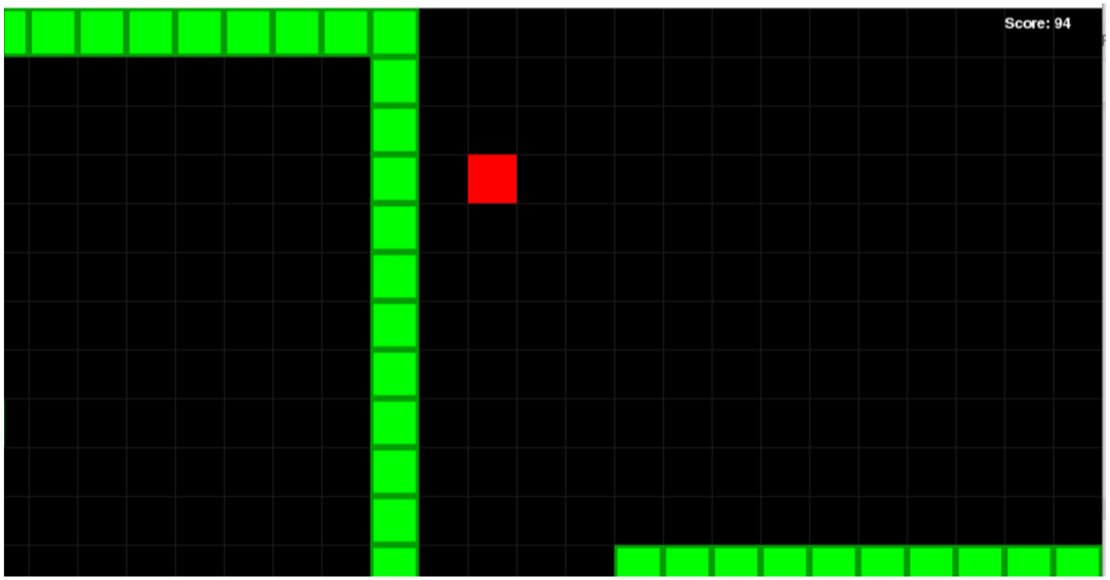

.KE GAME


Magnifi...
200%
Views





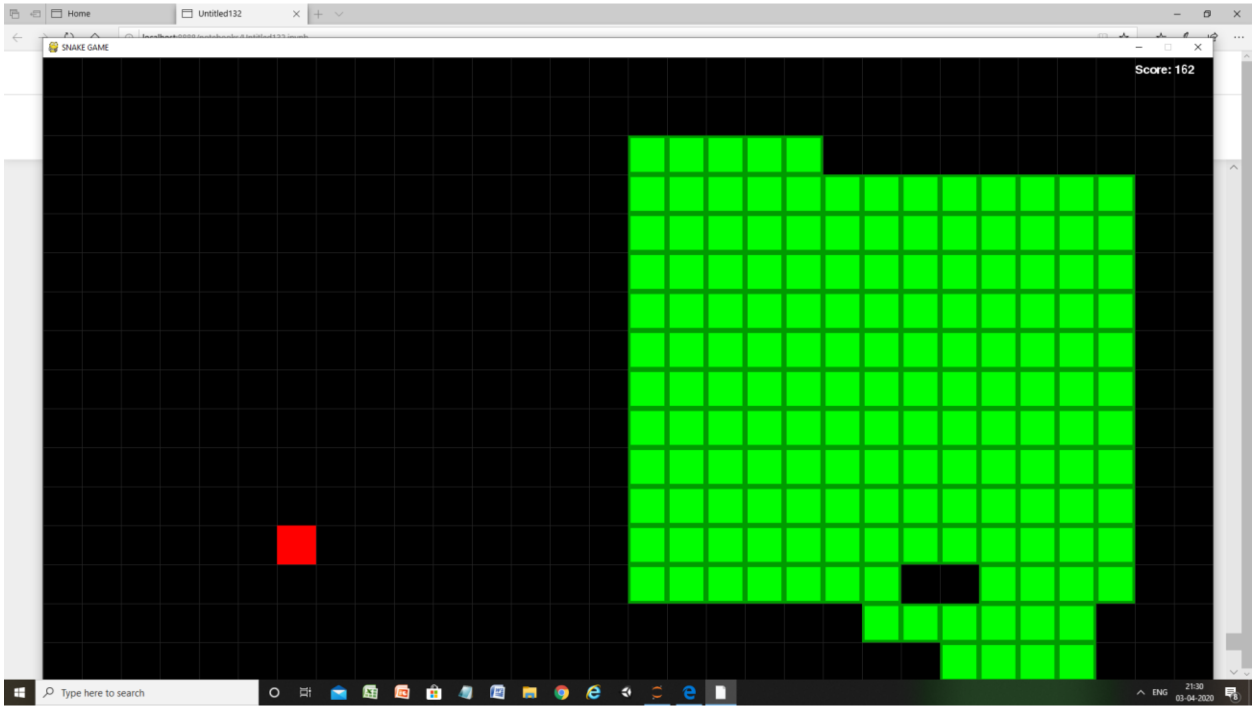
Score is represented on the top of right side of screen.

ss 

This will calculate the path of snake and when snake die screen become pause at that time……

```
almost died, recalcualting... {'x': 29, 'y': 19} ['up']
almost died, recalcualting... {'x': 29, 'y': 18} ['up']
almost died, recalcualting... {'x': 29, 'y': 17} ['up']
almost died, recalcualting... {'x': 29, 'y': 16} ['up']
almost died, recalcualting... {'x': 29, 'y': 15} ['up']
almost died, recalcualting... {'x': 29, 'y': 14} ['up']
almost died, recalcualting... {'x': 29, 'y': 13} ['up']
almost died, recalcualting... {'x': 29, 'y': 12} ['up']
almost died, recalcualting... {'x': 29, 'y': 11} ['up']
almost died, recalcualting... {'x': 29, 'y': 10} ['up']
almost died, recalcualting... {'x': 29, 'y': 9} ['up']
almost died, recalcualting... {'x': 29, 'y': 8} ['up']
almost died, recalcualting... {'x': 29, 'y': 7} ['up']
almost died, recalcualting... {'x': 29, 'y': 6} ['up']
almost died, recalcualting... {'x': 29, 'y': 5} ['up']
almost died, recalcualting... {'x': 29, 'y': 4} ['up']
almost died, recalcualting... {'x': 29, 'y': 3} ['up']
almost died, recalcualting... {'x': 29, 'y': 2} ['up']
almost died, recalcualting... {'x': 29, 'y': 1} ['up']
almost died, recalcualting... {'x': 29, 'y': 0} ['up']
YOU DIED!
```

AT last when no path available snake bit itself……game paused at that time.in order to exit screen press cross tab with space bar



IMPORTANT LIBRARIES USED :-

1**) Import random** :- generates a random module number between 0.0 to 1.0 . The function doesn’t need any arguments.

2) **import pygame :-** It is a simple import statement that imports the pygame and sys modules so that our program can use the functions in them.All of the pygame functions dealing with graphics,sound,and other features that pygame provides are in pygame module

3) **import operator :-** operator module exports a set of efficient functions corresponding to the intrinsic operators of python.

4) **import os** :- The main purpose of os module is to interact with the system operating system.The primary use it to create folders, remove folders and sometime change the working directory.

5) **import time :-** this module is used to handle time related tasks.

TEAM RESPONSIBILITY :- This is individual project made by me alone. I have created this snake game using AI.T have use python to write this code and AI part is implemented using BFS as well as A*.

REFERENCES :-

www.python.org

www.realpython.com

www.pythonspot.com

https://www.stackoverflow.com/