**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
|  | **Acknowledgement ...........................................................................** | **2** |
|  | **Abstract............................................................................................** | **3** |
|  | **Chapter 1 Introduction ..................................................................** | **4** |
| 1.1 | Project Introduction ........................................................................... | 4 |
| 1.2 | Scope .................................................................................................. | 4 |
| 1.3 | Objective ............................................................................................ | 5 |
|  | **Chapter 2 System Requirements Study ........................................** | **6** |
| 2.1 | User Characteristics ........................................................................... | 6 |
| 2.2 | Tools & Technology Used ................................................................. | 6 |
|  | **Chapter 3 System Design ...............................................................** | **7** |
| 3.1 | Project Flow ....................................................................................... | 7 |
|  | **Chapter 4 Implementation Planning ............................................** | **8** |
| 4.1 | Implementation Environment ............................................................ | 8 |
| 4.2 | Program / Modules Specification……............................................... | 8 |
|  | **Chapter 5 Working of Model ………………………...................** | **8** |
| 5.1 | Solution step – 1……………............................................................. | 11 |
| 5.2 | Solution step – 2………………………............................................. | 12 |
|  | **Chapter 6 Source code & Screen shots of Project .....................** | **17** |
| 6.1 | Source Code ……………….............................................................. | 17 |
| 6.2 | Snap Shot…………………. ............................................................ | 19 |
|  |  |  |
|  | **Chapter 7 Limitations & Future Enhancements ....................** | **22** |
|  | **Chapter 9 Conclusion& References............................................** | **23** |

**ACKNOWLEDGEMENT**

I hereby take this opportunity to thank each and every one who has helped me in creating and formulating this report of subject Seminar. I especially thank our faculties for guiding me through whole period of preparation and presentation. I express my gratitude towards my guide **Prof. Pinal Shah and HOD Parth Shah**, for giving me moral and academic support. At last I thank all those who directly or indirectly help me in preparing the report. I would like to thank all my friends, colleague and classmates for all the thoughtful and mind stimulating discussions I had, which prompted me to think beyond the obvious. Last but not least I would like to thank my family members who provide me enormous support during this works directly and indirectly.

Umang R. Gohel

(16IT024)

**ABSTRACT**

The “Intelligent Classic Snake” is an simple python base game for the purpose of the enjoyment. It is a small game for the purpose of enjoyment. Basically it’s an old mobile game which is in old Nokia phones. This project aims to bring the fun and simplicity of snake game with some new features. It will include computer controlled intelligent opponents whose aim will be to challenge the human players. It will also have the multiplayer feature that will allow more than one players to play the game. This game aims to change the way people think of traditional snake game. It will offer the experience of commercial multilayer games to the player retaining the simplicity of traditional snake game.

**1.0** **Introduction**

**1.1** **Project Overview**

It is a small game for the purpose of enjoyment. Basically it’s an old mobile game which is in old Nokia phones. This project aims to bring the fun and simplicity of snake game with some new features. It will include computer controlled intelligent opponents whose aim will be to challenge the human players. It will also have the multiplayer feature that will allow more than one players to play the game.

**1.2** **Scope**

This game aims to change the way people think of traditional snake game. It will offer the experience of commercial multilayer games to the player retaining the simplicity of traditional snake game.

**1.3** **Objective**

* This game aims to change the way people think of traditional snake game. It will offer the
* experience of commercial multilayer games to the player retaining the simplicity of traditional
* snake game.
* **The major objectives of this project are:**
* Create a snake game that will have all the functionality of traditional snake games.
* Introduce multilayer functionality in the game that will allow several players to play a game simultaneously. It should be able to give the experience of a real time multiplayer game to the players.
* Introduce computer controlled intelligent opponent (unique feature of this game) to make the game more challenging and interesting. The movement and action of these intelligent opponents will be controlled by computer whose aim will be to eat the food before human players capture it.

**2.0 System Requirements Study**

**2.1** **User Characteristics**

* Users :
  1. Player 1
  2. Player 2
* System :

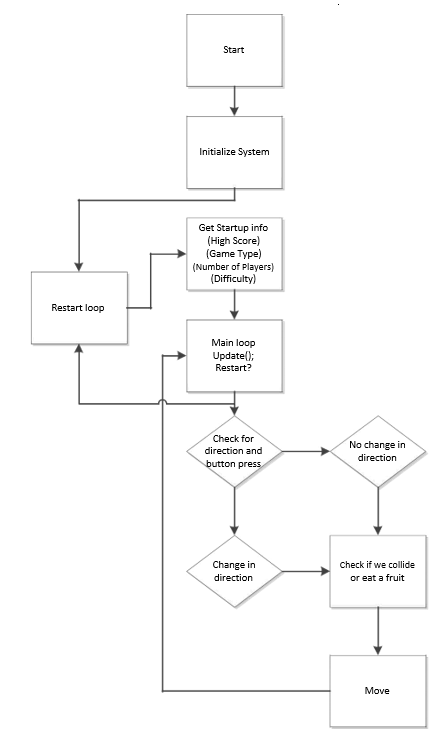
1. Automated computer intelligent system which play with the user 

**2.2** **Tools & Technology Used**

* Anaconda
* Spyder
* Py Game Library

**3.0** **System Design**

**3.1** **Project Flow**



**Figure 1: Theory of Operation**

**4.0 Implementation Planning**

**4.1** **Implementation Environment**

* Spyder

**4.2** **Program / Modules Specification**

* + - * Input Handler
      * Game Field Matrix
      * Game Field Canvas
      * User Interface Components
      * Game Controller
      * Intelligent Autonomous Opponent Snakes

**4.2.1 Input Handler**

It manages the task of sampling key strokes from local player and forwarding it to Game Controller when requested. It maintains a queue of size 2 so that quick keystrokes are not lost. It is active only when the game is in running mode.

**4.2.2** **Game Field Matrix**

Game Controller maintains the complete state of the game using game field matrix. It is a 2D array of size 58x58 (equal to the game field dimensions). Each game field object has a unique identifier in the game field. Game Controller updates the cells of this matrix in each cycle to register the changes that occur in the game.

**4.2.3** **Game Field Canvas**

It represents the game field as seen by the player. Game Controller analyzes the game field matrix in each cycle and updates the game field canvas to represent the state of game in that game cycle. Double Buffering [using java.awt.Canvas.createBufferStrategy()] has been implemented to avoid flickering of game field. Each block in the game field has dimension 10x10 pixels.

The update of game field canvas occurs in the way similar to the refreshing technique of a cathode ray monitor. The game field matrix and game field canvas are updated in separate thread. The update of game field canvas starts by scanning each column of the 1st row in the game field matrix, then 2nd row and so on upto the 58th row. The game field is refreshed twice during each game cycle (two refresh cycle for game field canvas for each game cycle). This is done make the movements in the game field smoother.

**4.2.4** **User Interface Components**

This module includes all the components, except game field, visible to the player. The look and feel of default swing components have been overridden to give the feel of a game to the players. MIT OCW's course[1](#page9) “[6.831] User Interface Design and Implementation” was very helpful during design of most of the user interface components of this game.

**4.2.5** **Game Controller**

It is the most important component of the iSnake client application. It coordinates the working of all the other modules in the application and handles all the messages received from the game server. Game Controller maintains the game cycle when the game is running. This game cycle is synchronized with the game cycle of the game server. All the updates to game field matrix are done during this cycle time. After expiry of each game cycle the game field is repainted to reflect the changes in the game field.

**4.2.6 Intelligent Autonomous Opponent Snakes**

These are computer controlled snakes, in the game, whose aim is to challenge the human players. We have two implementations of path finding algorithms to create intelligent autonomous opponent snakes. These algorithms return the shortest possible path from given source (S) and target (T) coordinate pair considering the obstacles (if any) present in the game field. The code name[3](#page12) for these two implementations are:

* Blackmamba *(refer to ANNEX ­ A)*
* Viper *(refer to ANNEX ­ B)*

A detailed paper describing the algorithm used by these two implementation is present in ANNEX A and ANNEX B. To know which of the two implementations perform better, we profiled them using a simple JUnit test and the results were plotted using gnuplot:

**5.0 Working of Model**

**5.1.1 solving step -1**

|  |
| --- |
| In this problem, we want to write a game where a graphical representation of a |
| snake moves across the screen. When it encounters a piece of food, the |
| snake grows longer and we gain a point. If it hits the wall or runs into itself, we |
| die. The snake is controlled by a MakeyMakey controller. To write this |
| program we are going to need: |
| ● A way of representing the snake |
| ● A way of representing the food |
| ● A way to display the score, |
| ● a way for our instructions to reach the snake, |
| ● and a way to know when we’ve run into something and died |
| Our system is going to involve working with both hardware and software, and |
| so we will need to understand what we have available in hardware that can |
| assist us. |
| The MakeyMakey board produces information that can be interpreted by our |
| computers as keystrokes, as if they were typed on a keyboard. The 6 options |
| available on the board face are the four directional arrows (UP, DOWN, LEFT, |
| RIGHT).  **5.1.2 Solution step – 2**  **Here are the requirements (functional requirements) for how the snake moves.**  1. The snake must appear to move around the screen  2. The snake must turn in response to user input  3. The snake will increase in length if it eats food  4. The snake will die if it runs over itself  5. The snake will die if it runs into the walls  6. The snake never stops moving  **5.1.3 To keep the game interesting.**  1. The snake must move at a speed fast enough to be interesting but slow enough to control.  2. The code must be object-oriented (curriculum requirement)  3. The game should still be playable with a really, really long snake. |
|  |
|  |
|  |

**5.1.4 Solution step – 3**

**Let's look at how a program to run the whole game might look:**

1. Draw the playing area with bounding rectangle, set the counter to zero and display it.

2. Draw the snake in a starting position.

3. Draw the food in a starting location.

4. On user input, change snake direction.

5. Move the snake one move

6. If the snake is over food, eat it, increase the score, grow, move the food

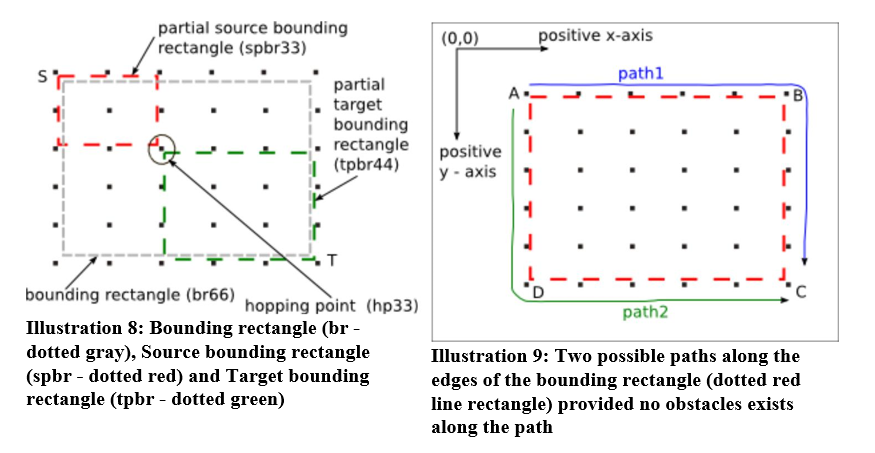
7. else if the snake is over itself or in a wall, die.

8. Go back to 4.

9. Until the snake dies

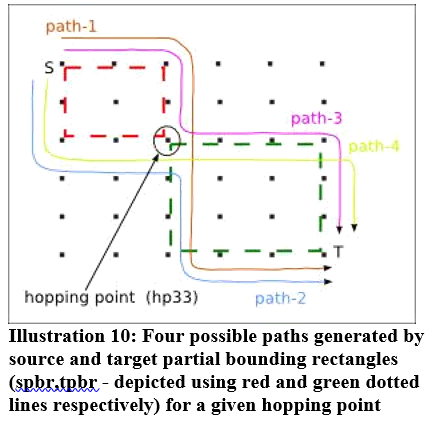
**5.1.5. Numbering of Paths**

* For any given bounding rectangle (br) there are only two possible paths as show in Illustration 9. Path numbering convention applied is:
* Path1: The x­coordinate of the path first changes followed by change in y­ coordinate
* Path2: The y­coordinate of the path first changes followed by change in x­ coordinate
* A source (S) and target (T) can be placed in four possible ways such that they lie on opposite ends of the diagonal. Considering the four position (A,B,C,D) as depicted in Illustration 9, we have the following four cases:



**5.1.6 Paths generated by partial bounding rectangles**

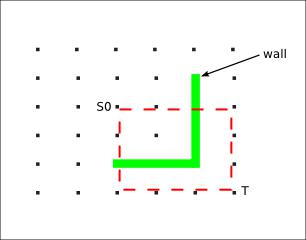
Four paths are generated by the source and target partial bounding rectangles for the given hopping point.



* obstacles (wall) are present along the paths formed using the bounding and partial bounding rectangles.
* No possible path can be found using all the possible combinations of the partial bounding rectangles.
* the bounding rectangle is not a square

**5.2 Backtracking**

If above path does not give a path from 'S' to 'T' the technique of backtracking will be applied. To illustrate the process of backtracking let us consider the scenario shown in Illustration 12.



**Illustration 12: Game field scenario when all the**

**paths formed by bounding and partial bounding**

**rectangles contain obstacle.**

The paths generated from the bounding rectangle (br44) and all the possible partial bounding rectangles contain obstacle. Hence the process discussed above will not result in any path from 'S0' to 'T'. For such scenario we can apply the process of backtracking.

**6.0 Source code & Screen shots of Project**

**6.1.1 Code**

import turtle

import time

import random

delay = 0.1

# Score

score = 0

high\_score = 0

# Set up the screen

wn = turtle.Screen()

wn.title("Snake Game by @TokyoEdTech")

wn.bgcolor("green")

wn.setup(width=600, height=600)

wn.tracer(0) # Turns off the screen updates

# Snake head

head = turtle.Turtle()

head.speed(0)

head.shape("square")

head.color("black")

head.penup()

head.goto(0,0)

head.direction = "stop"

# Snake food

food = turtle.Turtle()

food.speed(0)

food.shape("circle")

food.color("red")

food.penup()

food.goto(0,100)

segments = []

# Pen

pen = turtle.Turtle()

pen.speed(0)

pen.shape("square")

pen.color("white")

pen.penup()

pen.hideturtle()

pen.goto(0, 260)

pen.write("Score: 0 High Score: 0", align="center", font=("Courier", 24, "normal"))

# Functions

def go\_up():

if head.direction != "down":

head.direction = "up"

def go\_down():

if head.direction != "up":

head.direction = "down"

def go\_left():

if head.direction != "right":

head.direction = "left"

def go\_right():

if head.direction != "left":

head.direction = "right"

def move():

if head.direction == "up":

y = head.ycor()

head.sety(y + 20)

if head.direction == "down":

y = head.ycor()

head.sety(y - 20)

if head.direction == "left":

x = head.xcor()

head.setx(x - 20)

if head.direction == "right":

x = head.xcor()

head.setx(x + 20)

# Keyboard bindings

wn.listen()

wn.onkeypress(go\_up, "w")

wn.onkeypress(go\_down, "s")

wn.onkeypress(go\_left, "a")

wn.onkeypress(go\_right, "d")

# Main game loop

while True:

wn.update()

# Check for a collision with the border

if head.xcor()>290 or head.xcor()<-290 or head.ycor()>290 or head.ycor()<-290:

time.sleep(1)

head.goto(0,0)

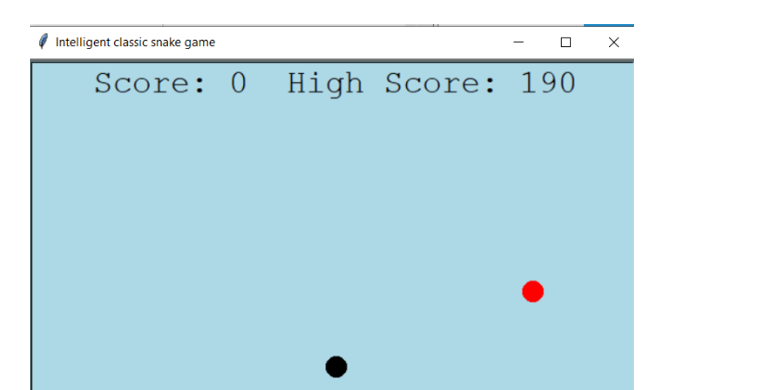
head.direction = "stop"

# Hide the segments

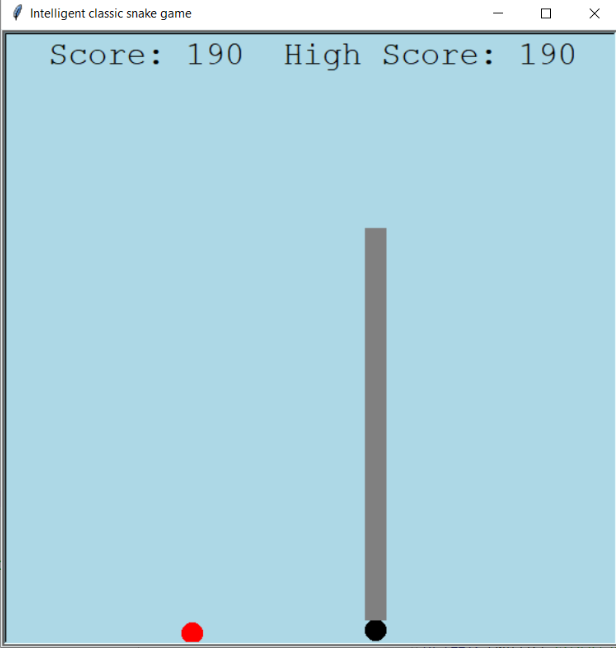
for segment in segments:

segment.goto(1000, 1000)

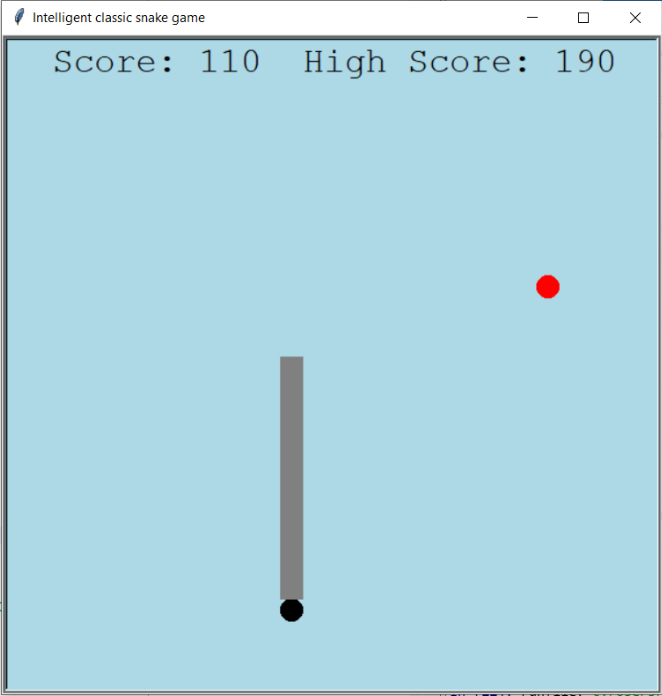
**6.1.2 Snap shot**

****

**Figure 6.1: GUI of game**

****

**Figure 6.2: picture of game - 1**

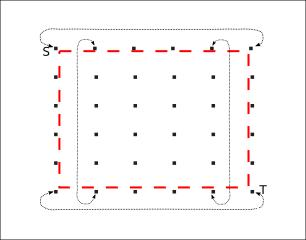
****

**Figure 6.3: picture of game - 2**

**7.0 Limitations**

The limitations of present implementation of Intelligent Classic Snake are:

**7.1.1** This algorithm does not consider the transparent game field boundary (entry to one side of the field causes exit in the opposite side of the field as depicted in Illustration 22) during path calculation. Due to this limitation the computed path is not optimal.



**Illustration 22: Transparent boundary of the game**

**field**

**7.1.2** The hopping points are always taken from the diagonal line joining the source and target. Because of property of the algorithm, it is not able to compute paths when a complex structure of wall, as shown in Illustration 23, is present. This is the reason why Blackmamba implementation enters infinite recursion for such obstacles

**7.1.3** If the number of FOR increases to too high, then it will obviously be tough and slow to determine the shortest path. Path finding algorithms (Blackmamba and Viper) implemented in this game have their own computation limitations which has been describe above.

Full stress test of the application has not been done yet. Hence, the response of game server in unpredictable situations cannot be handled properly.

* + 1. **Future Enhancements**
  + Port classic Snake to cell phone platform and One Laptop Per Child – OLPC (which uses Sugar Desktop environment). The presence of several connectivity options(Bluetooth, WIFI, GPRS, CDMA) in cell phones makes it a very attractive platform for a multiplayer game like classic. Local WIFI network formed by kids using OLPC laptops can be used as a platform for classic snake deployment.
  + As classic snake game server communicates with remote playing using a well defined and very simple protocol classic Snake clients programmed in other programming platform like Flash, Pycharm GUI, etc can be developed.

**8.0** **Conclusion**

* I was successful in creating a multiplayer version of traditional snake game. The computer controlled intelligent opponents have been successfully tested in the game is a unique feature of classic Snake.
* We learned several project management techniques used by professionals to develop large scale project. The experience of working in team and integration of modules developed independently, with just requirement specifications, is a very important achievement for the classic Snake team.

**References**

* + - * Information about Python, including tutorials:[​](http://littlebits.cc/projects)[​http://python.org](http://python.org/)
      * Information on the pygame library: [​http://pygame.org](http://pygame.org/)
      * [www.csermoocs.adelaide.edu.au](http://www.csermoocs.adelaide.edu.au)
      * [​http://pygame.org/wiki/GettingStarted](http://pygame.org/wiki/GettingStarted)​