Snake Design Document

Author: Bokai XU [bokaixu@link.cuhk.edu.cn](mailto:bokaixu@link.cuhk.edu.cn)

* Describe your thoughts and overall approach, including the breakdown of your logic into sub components and how these sub components are related/connected (basically the structure of your program)

We can divide the program into six components, including a screen, which is the game area. A snake and a monster, which is movable in the game area. We also need food items, which is static. To control the state of snake, we also need a control module. Last but not least, a module of game status is also important.

So, we can divide the program in to:

* + 1. A Snake
    2. A Monster
    3. Food Items
    4. Game Status
    5. Controls
* The logic breakdown is simple: there are only two timelines.

Motion of snake

The next game status

The previous game status

Motion of monster

- Drawing and erasing

After we finish the motion module, we need to display the result of our calculation. So we also need two modules which could draw a snake and a monster on the screen.

Clear snake

Motion of snake

Draw snake

Clear monster

Motion of monster

Draw monster

The next screen

The previous screen

To realize it, we can define a graphing function for snake. We can plug a snake list into it and it will help us draw some square boxes on the screen so it would look like a snake.

We can use turtle module as facilities for our function.

Note that we also need to erase the previous snake boxes on the screen, so we can try using stamp function. Each time we run the graphing function, we want to delete all the previous stamps, so we can use clearstamps() method. And we use a loop to make *n* stamps, where every one of stamps represents a square box. So we can draw a new snake on the screen.

The same is true for monster. We first clear the previous monster stamps. Then we create a new monster stamp so we can see a new monster on the screen.

So, in theory this can be done well.

* Control Module

We only need to control the direction of snake, so we use onkey() method provided by turtle.

Keyboard Control by User

Clear Snake, Motion of Snake, Draw Snake

The next screen

The previous screen

Clear Monster, Motion of Monster, Draw Monster

Then we should consider how to create a recursion. One direct approach is to use ontimer() method provided by turtle. It can run a given function with a given time interval. So it will repeat itself automatically. By the way we can also make a timer.

Clear Snake, Motion of Snake, Draw Snake

Keyboard Control by User

The previous screen

The next screen

Clear Monster, Motion of Monster, Draw Monster

Recursion: Ontimer(fun, time)

Then we have to consider how to terminate our game. That is, the game status.

According to the convention, the game ends when the snake is eaten by monster. The game ends when the snake becomes a loop. The game ends when all the foods are consumed by the snake.

Clear Snake, Motion of Snake, Draw Snake

Judge if Win/Lose

The next screen

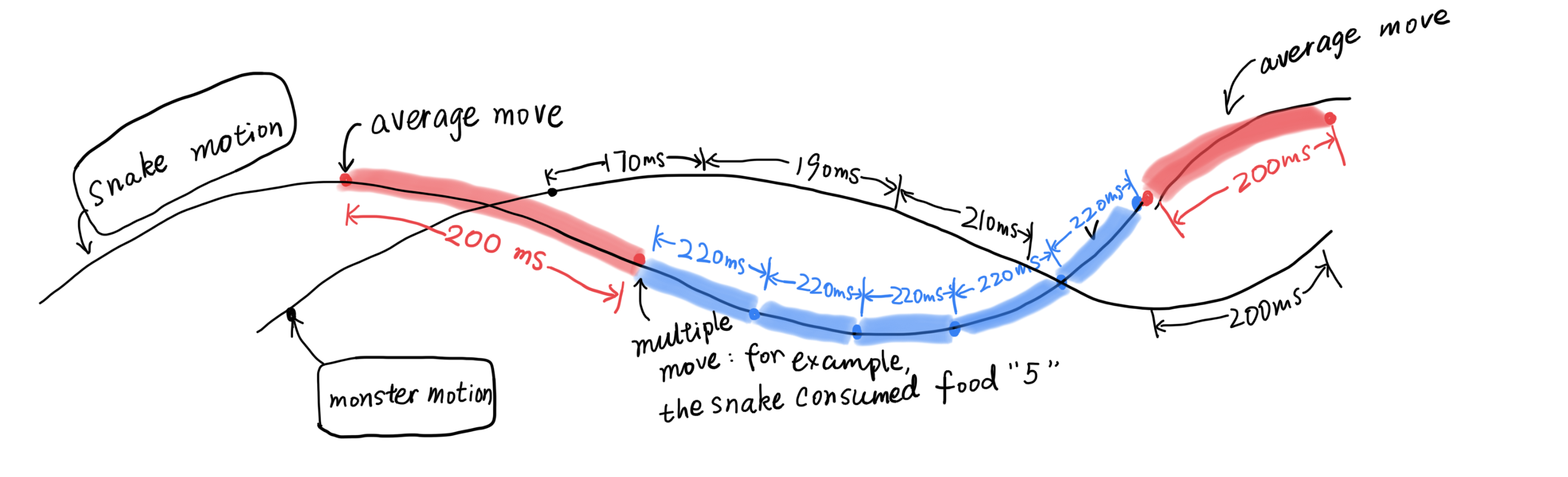
The previous screen

Clear Monster, Motion of Monster, Draw Monster

Keyboard Control by User

Recursion: Ontimer(fun, time)

- Then we consider the sequence which contains both average motion and food consuming motion of snake.

If the snake detected a food at some point on the screen, then it have to run a different procedure. The graph above is not robust enough to handle this problem, so we need to consider a new method. We can view screen as an independent object, we do not need to refresh it every second. We just refresh it manually, wherever we need a refreshment. Then we can also regard the Snake module as an independent process. So it is with Monster module.

By adopting a new approach, the switches between different modes (or series) will be much more clear. At each of our junction, we can use ontimer(fun, time\_interval) to switch freely between different modes. So, we have to make sure that there is only one function in Snake module is running and only one function of Monster module is running.

So we only need to make sure that all the functions can be assembled and run smoothly. This is an illustration, showing what happens in an time interval of 1000ms:

We may not compile those logic together in one *controlling function*, so we just let it go. We only focus on what happens within the *screen*.

Game Starts:

Two

Timelines

Sequence of Monster Motion : A E A E A F A K A A A A P A K A K K A K A A A K P P A P P P O K K K

Sequence of Snake Motion : A E B E B F B K B A B A P B K A K K A K A

*(Notice that A represents that the result of the function will be displayed on the screen, those thin texts represent that the result of the function is just for controlling, B represents user input)*

Given that we are sure that there is simultaneously one task within the sequence of Snake Motion, and simultaneously one task with the sequence of Monster Motion. This is an important logic in constructing the program.

Screen [A]

Game Starts:

Two

Timelines

Sequence of Monster Motion : A E A E A F A K A A A A P A K A K K A K A A A K P P A P P P O K K K

Sequence of Snake Motion : A E B E B F B K B A B A P B K A K K A K A

User Input [B]

Interaction Between Snake, Monster and System: E(Game Status) F(Food consumption) K P O…

Here I provide my program structure:

* Describe the data types that you use to develop your program for tracking the various game objects (snake, monster and food items)

1. For the snake object, we can make it a list.

snake\_list = [(x1, y1), (x2, y2), (x3, y3), (x4, y4), …]

snake\_dir = ‘u’ or ‘d’ or ‘r’ or ‘l’

Where the numbers in parenthesis is the coordinate of a part of snake. Note that the first one (x1, y1) is the head of snake, the second one represents that second part of snake. The last one (xn, yn) is the end of snake.

When we want to retrieve the position of snake head, we just use:

snake\_list[0]

And when we want to append a new segment to the snake, we can process the list with a function and get the return. When we want to make the snake move, we can also process the list with a function and get the return.

When drawing the snake, we just pick snake\_list[\*] and draw a number of small squares.

2. For the monster object, we consider a sequence (x, y):

mons\_pos = (x\_mons, y\_mons)

dir\_mons = ‘u’ or ‘d’ or ‘r’ or ‘l’

That’s because the monster only consists of one square.

3. For the food items, we consider using a dictionary:

food\_list = {1: (x1, y1), 2: (x2, y2), 3: (x3, y3), 4: (x4, y4), 5: (x5, y5), …}

That’s because the food is distributed within the game area entirely randomly. There does not exist a method that can make them ordered. So we may not use list, because we may delete some food items form it so that we won’t know what the *n*-th term is. List is not reliable here, so we use dictionary.

About important global variables:

4. A integer which determines how long should the nail extend, namely, food\_payable.

Every time snake consumes a food item, it would add a number which equal to the value of that food item. Every time the snake is to extend its nail, the integer will be subtract 1:

food\_payable = food\_payable - 1

When this integer becomes 0 and food items are all consumed, the user will win the game.

5. Two time variables, start\_time and end\_time:

start\_time is the absolute time when the game starts, end\_time is the absolute time when the game ends.

6. Three controlling variables which control three timelines, including snake, monster and time counter: snake\_ctr and mons\_ctr and count\_ctr:

When they become ‘p’, the thread of snake and monster will be killed automatically, so they look like stopped. When they become ‘o’, the thread of snake and monster can be started, but they still need being invoked.

7. A integer which determines how many times does the snake contact with monster: total\_numbers.

8. The minimum time interval of both snake and monster: time\_unit. The default value is 200 ms. The refresh interval of snake is 200 ms. The refresh interval of monster is. a random integer from 200 \* 2 \*0.95 to 200 \* 2 \* 1.05.

9. About four turtle objects: snake, monster, food and screen.

The first two represent the snake and monster, which will be displayed on the screen. The last one is screen object, which would display snake and monster.

snake= Turtle()

mons= Turtle()

food=Turtle()

screen= Screen()

10. What parts of the snake have intersection with monster? We use contact\_list.

Contact\_list = [1, 2]

This means that the second and third parts of snake have intersection with monster.

* Describe the motion logic for both snake and monster.
* Snake Motion Module:

We can use a series of function to convert the previous status to the next status. We can consider representing the snake by a list. Then at one time, we can read the corresponding snake list. Then the task is to convert an old list into a renewed list.

L1 = [(1, 1), (2, 1), (3, 1), (4, 1)]

Assume the direction of the snake is left, then we need a function to convert L1 into L2:

L2 = [(0, 1), (1, 1), (2, 1), (3, 1)]

This function is easy to define.

1. A function that add a new coordinate pair to the left hand side of the list according to the direction of the motion. If the snake moves to the right, we can add a new coordinate pair on the right hand side of the snake head, vice versa.

2. A function that deletes the last coordinate pair.

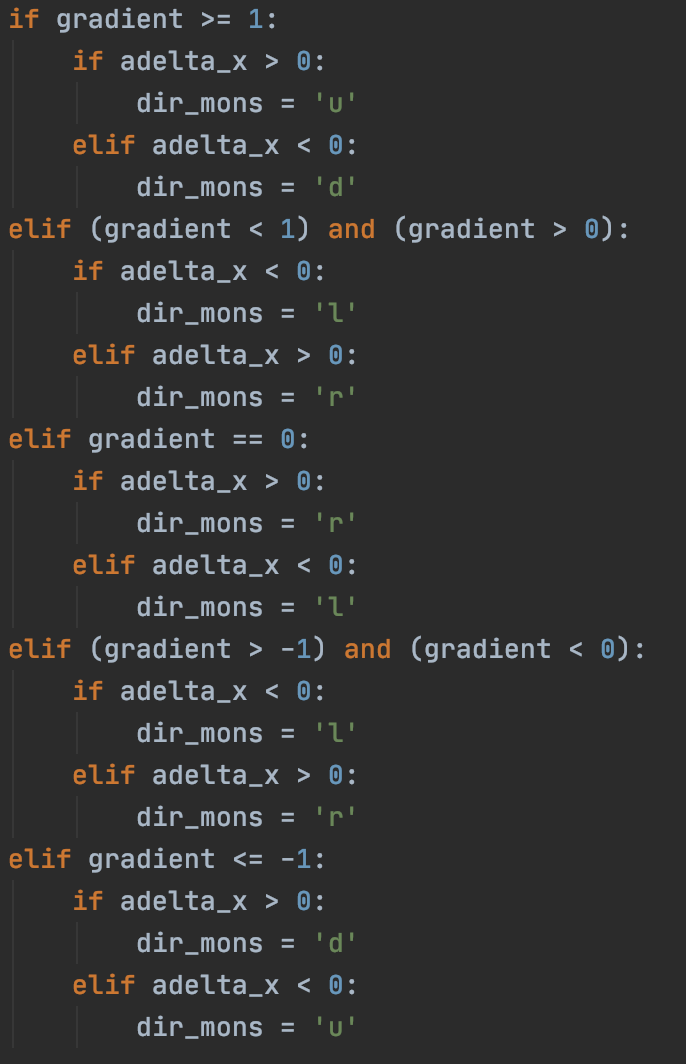
|  |  |
| --- | --- |
|  |  |
| Before | After |

This is straight forward because if we consider the snake is (x(t), y(t)), we can re-parametrize the shape of snake by t, and we consider the black square as liquid, so it will move along t, thus the last square should be removed.

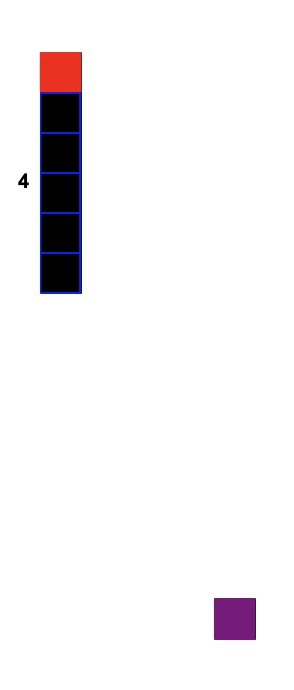
Actually the two functions above can be complied to one function, namely,

snakeNewList()

- Monster Motion Module:

The case is different from motion of snake. The motion of monster is dependent on the motion of snake. So we use an algorithm to determine the direction of snake. This process is realized by calculating the gradient of the line between two objects (snake head and monster).

We can discuss different gradient separately and give an optimized direction in order to approach the snake head. In theory, this can be done. We define a function, which contains the algorithm, namely,



monsDirection()

We can assign an optimized direction for each gradient:

G > 1 and dX > 0 : Go up

0 < G < 1 and dX > 0: Go Right

-1 < G < 0 and dX > 0: Go Right

…

We only need to give 8 different conditions.

After the monster moves to the next position, it will repeat the previous procedure.

By repeating this process, the monster will eventually reach the snake head.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

* Describe the expansion logic for the snake tail:

Assume that snake consumed a food item. At this time we have to extend the length of tail. It is the same as the average motion, but a little different:

L1 = [(1, 1), (2, 1), (3, 1), (4, 1)]

Here we don’t delete the last coordinate pair, because the tail is actually longer than the last one.

Then we have to append a coordinate pair to the left of the list:

L2 = [(0, 1), (1, 1), (2, 1), (3, 1), (4, 1)]

When the snake consumed a food item, system will add equivalent value to a global variable called food\_payable:

food\_payable = food\_payable + food\_post

Every time the tail of snake extends by 1 unit, system will subtract the variable by 1:

food\_payable = food\_payable - 1

When the variable becomes 0, the extension of tail will end.

If the snake is to consume another food item, it will just repeat this process.

This procedure is finished by function snakeMoveHub().

* Describe the body contact logic between the monster and the snake:

The basic approach is to calculate the distance between the monster and every part of the snake. If there exist more than or equal to 1 part of the snake such that distance along X axis and distance along Y axis are both smaller than 1 unit, we may conclude that the monster has contacts with snake.

We use a while loop to search any snake parts which have intersection with monster:

while card <= length:

index\_card = card - 1

cod\_xy = snake\_list[index\_card]

x\_s = cod\_xy[0]

y\_s = cod\_xy[1]

delta\_x = abs(x\_s - x\_m)

# print(delta\_x)

delta\_y = abs(y\_s - y\_m)

# print(delta\_y)

card = card + 1

if (delta\_x <= 1) and (delta\_y <= 1):

contact\_list.append(card)

This may be rational, but we should pay attention to the practice.

Every time the monster moves and every time the snake moves, we invoke the function. But this may cause the overestimation of times of contacts, the times of contacts may be counted twice or even more times (because the refresh time interval is very short). The contact is not well-defined and hard to calculate how many times the snake contact with monster.

My solution is: We define the concept contact as a process form the time when there exist a part of snake has contact with monster to the time when there is no contact between monster and snake. Then we can conclude the following process is so-called contact.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| My Definition | 0 | 1 | 1 | 1 |
| Wrong Definition | 0 | 1 | 2 | 2 |

Every time my program concludes that there is contract, we add 1 to the total number of contacts.

- Describe usage of all your newly defined functions, including details of parameter(s):

1. Snake Module:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Argument/Input/Global Variables | Output/Return | Description |
| snakeCreate() | N/A | snake\_list | At the beginning of the game, this function is to generate all the coordinates of food items randomly |
| snakeDrawInit() | snake\_list | *Screen: a snake head* | At the beginning of the game, it will draw a snake head in the middle of screen |
| snakeCovertCod() | snake\_list | cod\_list | Convert relevant coordinates into absolute value |
| snakeDrawRedSqu() | N/A | *Screen: draw a snake head* | Draw a snake head on the screen |
| snakeDrawSqu() | N/A | *Screen: draw other parts of snake* | Draw a snake without head on the screen |
| snakeDrawGeneral() | cod\_list | Control snakeDrawSqu() and snakeDrawRedSqu() | Given the coordinate list of snake, it can draw an entire snake |
| snakeNewList() | snake\_list | snake\_list | Renew the snake list so our snake can move on automatically |
| snakeNewFreq() | status = ’n’(not consuming food) or ‘e’(consuming food) | snake\_freq | Return a new frequency for the movement of snake, when the snake starts consuming food and stop consuming food |
| snakeMoveHub() | N/A | *Screen: a new snake*  *Title: a new status* | Every time the snake moves, it will refresh the title bar. |
| Make the snake move forward and the control the extension of tail |
| Judge if the game is over |
| Make the snake stop when the snake is going to crash the border |
| Invoke itself so the snake can move continuously |
|  |  |  | Judge if the food items are eaten up by the snake, if so, the user wins. |
|  |  |  |  |

1. Monster Module:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Argument/Input/Global Variables | Output/Return | Description |
| monsCreate() | N/A | mons\_pos | At the beginning of the game, this function is to generate a monster randomly |
| monsDrawInit() | mons\_pos | *Screen: a monster* | At the beginning of the game, it will draw a monster randomly within screen |
| monsDrawGene() | N/A | *Screen: a monster* | Draw a monster within the screen |
| monsDirection() | snake\_list and mons\_pos | dir\_mons | Given the position of monster and snake head, it will return a new direction for monster so the monster may catch the snake head |
|
| monsNewFreq() | N/A | mons\_freq | Return.a new frequency for the movement of monster, randomly, between 0.8x and 1.2x. |
| monsMoveHub() | N/A | Invoke statusCountHub() | Every time it is invoked, it will refresh the title bar |
| Invoke statusIfEatByMons() | Judge if the game is over |
|  | Realize the movement of monster |
|  | Invoke itself so the monster will continue moving around |

1. Food Module:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Argument/Input/Global Variables | Output/Return | Description |
| foodCreate() | N/A | food\_list | At the beginning of the game, this function is to generate all the coordinates of food items randomly |
| foodDraw() | food\_list | *Screen: 9 numbers* | Given a food list and draw all the food items on the screen |
| foodCoverSqu() | N/A | *Screen: a white square covering the given food item* | Draw a white square to hide the food item which has been consumed |

1. Game Status Module:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Argument/Input/Global Variables | Output/Return | Description |
| statusIfEatByMons() | snake\_list and mons\_pos | True or False | Return True if snake head is eaten by monster |
| statusIfLoop() | snake\_list | True or False | Return True if snake becomes a loop |
| statusIfFoodInRange | snake\_list and food\_list | True or False | Return True if food is available for snake |
| statusDrawOver() | snake\_ctr and mons\_ctr and count\_ctr | snake\_ctr and mons\_ctr and count\_ctr | Stop the motion of snake and master and stop the timer |
| *Screen: Display a text saying Game is Over* | Game is Over |
| statusIfCont() | snake\_list and mons\_pos | contact\_list | Return a global variable storing all the parts of snake which have intersection with monster. |
| 0 or 1 | When the snake has intersection with monster this time and no intersection last time, return 1. Else return 0. This is to keep track of times of contact. (See my definition of Contact) |
| statusCountHub() | startTime and endTime | *Screen Title: Display the time* | Refresh the title of screen |
| statusIfCont() | *Screen Title: Display the times of contacts* |

1. Control Module:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Argument/Input/Global Variables | Output/Return | Description |
| controlNull() | N/A | N/A | A null function |
| controlPause() | Global variables:  mons\_ctr  snake\_ctr  count\_ctr | Global variables:  mons\_ctr  snake\_ctr  count\_ctr | Pause all timelines (Stop the motion of snake, monster and time counter) |
|  | Invoke controlResume() | Prompt user to resume |
| controlResume() | Global variables:  startTime | startTime | Make correctness on the time counter |
| Global variables:  mons\_ctr  snake\_ctr  count\_ctr | Global variables:  mons\_ctr  snake\_ctr  count\_ctr | Resume the motion of snake, monster and time counter. |
| controlRight()  controlLeft()  controlUp()  controlDown() | snake\_dir | snake\_dir | When the snake is not at the border, change the direction of snake. |
| snake\_ctr | Prompt user to input a valid direction.  Invoke snakeMoveHub() | When the snake got to the border, it will prompt user to give a valid direction, otherwise it would not do anything.  After the user gives a valid input, it will invoke snakeMoveHub() so the snake would move again. |
| controlKeyUser() | *Keyboard Input* | Invoke:  controlUp()  controlDown()  controlRight()  controlLeft() | This function will be invoked automatically.  It can listen the keyboard input by user and give response. |
| controlDisplayUI() | N/A | *Screen: Display a welcome screen and prompt user to continue* | Screen: Display a welcome screen and prompt user to continue |
| controlDynam() | (x, y) | Invoke:  clearscreen() | Clear the welcome screen. |
| Invoke:  monsMoveHub()  snakeExtendSingle()  snakeMoveHub() | This function is to activate the game. |
| Invoke:  snakeExtendSingle() for 5 times | Simulate the growth of snake (the length of snake is 5) |

F. Intializer Module:

|  |  |  |  |
| --- | --- | --- | --- |
| intializer() | N/A | Global variables:  n = 0  stamp\_list = []  total\_numbers = 0  contact\_list = []  eating = False  strat\_time = time.pref\_counter() | Define some properties of game. |
| Global variables:  count\_ctr  snake\_ctr  mons\_ctr  over\_control | The status of snake, monster and time counter. |
| Global Variables:  time\_unit | Define the minimum refresh frequency of the game objects |
| Global Objects:  snake  mons  food  screen | The creation of Turtle instance |
| Invoke:  foodCreate()  snake\_create()  snakeDrawGeneralInit()  monsCreate()  monsDrawInit()  controlDisplayUI() | Draw a snake and monster on the screen. Prepare all the food items. |
| Prompt user to continue |  |

- Sample Outputs:

1. Winner:

|  |  |
| --- | --- |
| 1 |  |
| 2:  0 items consumed |  |
| 3:  3 items consumed |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8:  Winner |  |

2. Game Over:

|  |  |
| --- | --- |
| 1:  0 items consumed |  |
| 2 |  |
| 3 |  |
| 4:  3 items consumed |  |
| 5 |  |
| 6:  Game Over |  |