
VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
DEPARTMENT OF INFORMATION TECHNOLOGY
SUBJECT: **PROGRAMMING TECHNIQUES**

# GUIDELINE

# HUNTING SNAKE

HCM city, 10th January 2022

# CONTENTS

# 1 Introduction

In this part, we use some class techniques and basic data structure to build a simple game of snake. To complete this project, we need to know some basic knowledge: file processing, one/two-dimensional array… This guideline only helps students to build the basic game. You should research many problems and do your best.

# 2 Screenplay

Firstly, the game shows the frame with a snake slowly moving, and user controls the keys 'W', 'A', 'S', 'D' to move it.

When my snake touch another side of the frame, the program asks the players if they want to continue. If the players choose "y", the program will restart. Otherwise, we exit the game.

When a snake eats "**enough food**", its speed will be increased (meaning that it moves to the next level). Keeping the length of snake unchanged is up to you. When our snake moves to "**enough level**", the game restarts again.

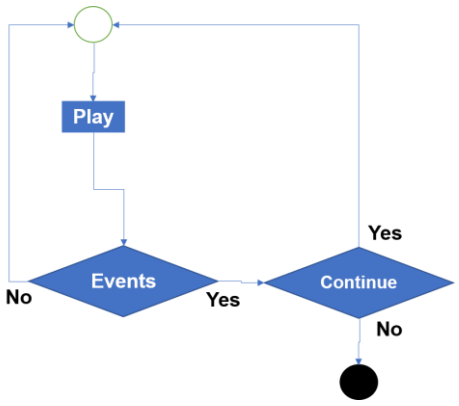



Figure 1: Screen-play diagram

# 3 Some steps to build the game

In this part, we go through all the steps to build the basic game. Note that this is only the guideline, group of students can make your own design.

Step 1: In this step, we use the function "fixConsoleWindow" to fix the screen with reasonable size. This helps to prevent the players from resizing the screen, and make our computation be incorrect.

| Lines | |
|---|---|
| 1 | `void FixConsoleWindow() {` |
| 2 | `    HWND consoleWindow = GetConsoleWindow();` |
| 3 | `    LONG style = GetWindowLong(consoleWindow, GWL_STYLE);` |
| 4 | `    style = style & ~(WS_MAXIMIZEBOX) & ~(WS_THICKFRAME);` |
| 5 | `    SetWindowLong(consoleWindow, GWL_STYLE, style);` |
| 6 | `}` |

In above code, HWND is a "special handle" (**HWND is not a pointer-type**) to Console window. To work with these graphical object, we need such that type. Flag GWL_STYLE is a mark for function "GetWindowLong" to get the features of Console window. Returned-type of "GetWindowLong" is a long, and we edit at line 4. The goal is to disable the button "maximize" and ask the players not to resize the window. Next, we use "SetWindowLong" to assign the edited-result back to.

Step 2: In board, there are many positions we want to print there, so we need to move to all the positions of console window

| Lines | |
|---|---|
| 1 | `void GotoXY(int x, int y) {` |
| 2 | `    COORD coord;` |
| 3 | `    coord.X = x;` |
| 4 | `    coord.Y = y;` |
| 5 | `    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);` |
| 6 | `}` |

In this code, we use struct _COORD (COORD). This is a structure reserved for console coordination processing. We assign x-value and y-value to coord variable, then set the position with "SetConsoleCursorPosition". Note: this function needs a main object which is console window, so we also need a handle to this window. We have it by calling "GetStdHandle" with the flag parameter called STD_OUTPUT_HANDLE.

Step 3: We need data for our game. For simple, we use global variables. Their roles are presented in below table.

| Lines | |
|---|---|
| 1 | `//Constants` |
| 2 | `#define MAX_SIZE_SNAKE 10` |
| 3 | `#define MAX_SIZE_FOOD 4` |
| 4 | `#define MAX_SPEED 3` |
| 5 | `//Global variables` |

```
6  POINT snake[10]; //snake
7  POINT food[4]; // food
8  int CHAR_LOCK;//used to determine the direction my snake cannot move (At a moment, there is one direction my snake cannot move to)
9  int MOVING;//used to determine the direction my snake moves (At a moment, there are three directions my snake can move)
10 int SPEED;// Standing for level, the higher the level, the quicker the speed
11 int HEIGH_CONSOLE, WIDTH_CONSOLE;// Width and height of console-screen
12 int FOOD_INDEX; // current food-index
13 int SIZE_SNAKE; // size of snake, initially maybe 6 units and maximum size may be 10
14 int STATE; // State of snake: dead or alive
```

Step 4: We build "ResetData" function, and its goal is to reset all data. This function calls "GenerateFood" function, initializing the positions of food (coordinates of food array must be different from snake's). So, the function "GenerateFood" calls "IsValid" to verify their coordinates.

```
Lines
1  bool IsValid(int x, int y) {
2     for (int i = 0; i < SIZE_SNAKE; i++) {
3        if (snake[i].x == x && snake[i].y == y) {
4           return false;
5     return true;
6  }
7
8  void GenerateFood() {
9     int x, y;
10    srand(time(NULL));
11    for (int i = 0; i < MAX_SIZE_FOOD; i++) {
12       do {
13          x = rand() % (WIDTH_CONSOLE -1) + 1;
14          y = rand() % (HEIGH_CONSOLE -1) + 1;
15       } while (IsValid(x, y) == false);
16       food[i] = { x,y };
17    }
18 }
19
20 void ResetData() {
21    //Initialize the global values
22    CHAR_LOCK = 'A', MOVING = 'D', SPEED = 1; FOOD_INDEX = 0, WIDTH_CONSOLE = 70, HEIGH_CONSOLE = 20, SIZE_SNAKE = 6;
23 // Initialize default values for snake
24    snake[0] = { 10, 5 }; snake[1] = { 11, 5 };
```

```
25    snake[2] = { 12, 5 }; snake[3] = { 13, 5 };
26    snake[4] = { 14, 5 }; snake[5] = { 15, 5 };
27    GenerateFood();//Create food array
28  }
```

The function “IsValid” checks if the input-coordinates are the same as anything of snake-array or not. If this hold, it will return false to notify of the invalidity. Otherwise, it returns true.

The function “GenerateFood” randomly creates food-array. Here, we use the technique of randomly generation of number. The function “srand(time(NULL))” (Note: we can leave it in “main()” function) to create a random-seed, the function “time(NULL)” returns the milli-second (long datatype) counted from 1/1/1970. In loop “do while”, we create all elements of the food-array (using POINT datatype for each member). If the coordinate is valid, we assign it food[i] = {x, y}

Finally, the function “ResetData()” will assign the initial values for the game.


Figure 2: Function-calling diagram started from “ResetData()”

Step 5: Next, we build the function “StartGame()”. This function consists of all tasks needed to do before coming into game

```
Lines
1   void StartGame() {
2     system("cls");
3     ResetData(); // Intialize original data
4     DrawBoard(0, 0, WIDTH_CONSOLE, HEIGH_CONSOLE); // Draw game
5     STATE = 1;//Start running Thread
6   }
7
8   void DrawBoard(int x, int y, int width, int height, int curPosX = 0, int curPosY
    = 0){
9     GotoXY(x, y);cout << 'X';
10    for (int i = 1; i < width; i++)cout << 'X';
11    cout << 'X';
12    GotoXY(x, height + y);cout << 'X';
13    for (int i = 1; i < width; i++)cout << 'X';
14    cout << 'X';
15    for (int i = y + 1; i < height + y; i++){
```

```
16        GotoXY(x, i);cout << 'X';
17        GotoXY(x + width, i);cout << 'X';
18     }
19     GotoXY(curPosX, curPosY);
20  }
```

The first line is used to clean the screen. The second line is used to call the function "ResetData()" to reset the data to original values. Next, we call the function "DrawBoard()" to draw the surrounded rectangle. Final line is important, when STATE = 1, then the "snake" is drawn on screen.

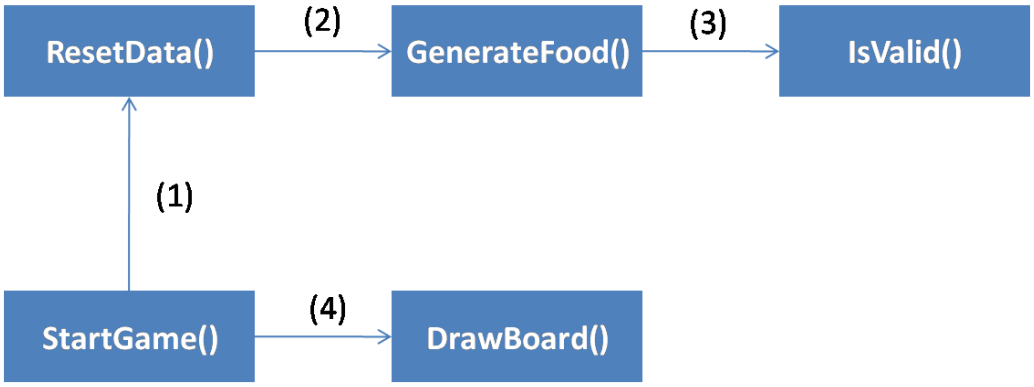



Figure 3: Function-calling diagram started from "StartGame()"

Step 6: In addition to "StartGame()", we build two functions "ExitGame()" and "PauseGame()" to exit and pause game.

```
1  //Function exit game
2  void ExitGame(HANDLE t) {
3     system("cls");
4     TerminateThread(t, 0);
6  }
7
8  //Function pause game
9  void PauseGame(HANDLE t) {
10    SuspendThread(t);
11 }
```

In "ExitGame()", we clean the screen and terminate running-thread, and we need the HANDLE of that thread. Note: for the sake of illustration, we directly terminate the thread. If that thread has pointers, this will leak memories. In "PauseGame()", we pause the game by using "SuspendThread()"

Step 7: Next, we build the function to process the case that my "snake" eats "food". The length of snake will increase by one, and the next element of food-array will be rendered. Because the **food-array is initialized before, there is a possibility that their coordinates may be the same as the elements of snake-array**.

| Lines | |
|---|---|
| 1 | `//Function to update global data` |
| 2 | `void Eat() {` |
| 3 | `   snake[SIZE_SNAKE] = food[FOOD_INDEX];` |
| 4 | `    if (FOOD_INDEX == MAX_SIZE_FOOD - 1)` |
| 5 | `    {` |
| 6 | `        FOOD_INDEX = 0;` |
| 7 | `        SIZE_SNAKE = 6;` |
| 8 | `        if (SPEED == MAX_SPEED) SPEED = 1;` |
| 9 | `        else SPEED++;` |
| 10 | `        GenerateFood();` |
| 11 | `    }` |
| 12 | `    else {` |
| 13 | `        FOOD_INDEX++;` |
| 14 | `        SIZE_SNAKE++;` |
| 15 | `    }` |
| 16 | `}` |

If all elements of food-array are eaten, we set the global variables to increase the speed of snake. Otherwise, we render the next element of food-array, and increase the size of snake.


Figure 4: Function-calling diagram started from "Eat()"

Step 8: This function processes the case that my snake touches the "wall". We need to change its STATE = 0 to stop the loop drawing snake, and print the string to ask if the player wants to continue ('y' to continue) or not (anykey)

| Lines | |
|---|---|
| 1 | `//Function to process the dead of snake` |
| 2 | `void ProcessDead() {` |
| 3 | `   STATE = 0;` |
| 4 | `   GotoXY(0, HEIGH_CONSOLE + 2);` |
| 5 | `   printf("Dead, type y to continue or anykey to exit");` |

| | |
|---|---|
| 6 | } |

Step 9: The function “DrawSnakeAndFood()” to draw “food” and “snake” continously. This function scans the snake-array to print its characters.

| Lines | |
|---|---|
| 1 | `//Function to draw` |
| 2 | `void DrawSnakeAndFood(char* str){` |
| 3 | `   GotoXY(food[FOOD_INDEX].x, food[FOOD_INDEX].y);` |
| 4 | `   printf(str);` |
| 5 | `   for (int i = 0; i < SIZE_SNAKE; i++){` |
| 6 | `      GotoXY(snake[i].x, snake[i].y);` |
| 7 | `      printf(str);` |
| 8 | `   }` |
| 9 | `}` |

Step 10: Next, we build the function “Up”, “Down”, “Left”, “Right” to process the moving direction of snake. When moving, we check if my snake touches the “wall” or not (the 1st If statement). If my snake touches the “wall”, we call “ProcessDead()” and exit because there is no need to update snake-array. However, if there is no any impact and “snake” eat “food” (the 1st if in else statement), we call “Eat()”. Then, we increase all the elements of snake-array by one position (actually, we assign the values of the back-element to the front-element).

| Lines | |
|---|---|
| 1 | `void MoveRight() {` |
| 2 | `   if (snake[SIZE_SNAKE - 1].x + 1 == WIDTH_CONSOLE) {` |
| 3 | `      ProcessDead();` |
| 4 | `   }` |
| 5 | `   else {` |
| 6 | `      if (snake[SIZE_SNAKE - 1].x + 1 == food[FOOD_INDEX].x && snake[SIZE_SNAKE - 1].y == food[FOOD_INDEX].y) {` |
| 7 | `         Eat();` |
| 8 | `      }` |
| 9 | `      for (int i = 0; i < SIZE_SNAKE - 1; i++) {` |
| 10 | `         snake[i].x = snake[i + 1].x;` |
| 11 | `         snake[i].y = snake[i + 1].y;` |
| 12 | `      }` |
| 13 | `      snake[SIZE_SNAKE - 1].x++;` |
| 14 | `   }` |
| 15 | `}` |
| 16 | |
| 17 | `void MoveLeft() {` |
| 18 | `   if (snake[SIZE_SNAKE - 1].x - 1 == 0) {` |

```
19        ProcessDead();
20     }
21     else {
22        if (snake[SIZE_SNAKE - 1].x - 1 == food[FOOD_INDEX].x && snake[SIZE_SNAKE
   - 1].y == food[FOOD_INDEX].y) {
23           Eat();
24        }
25        for (int i = 0; i < SIZE_SNAKE - 1; i++) {
26           snake[i].x = snake[i + 1].x;
27           snake[i].y = snake[i + 1].y;
28        }
29        snake[SIZE_SNAKE - 1].x--;
30     }
31  }
32
33  void MoveDown() {
34     if (snake[SIZE_SNAKE - 1].y + 1 == HEIGH_CONSOLE) {
35        ProcessDead();
36     }
37     else {
38        if (snake[SIZE_SNAKE - 1].x == food[FOOD_INDEX].x && snake[SIZE_SNAKE -
   1].y + 1 == food[FOOD_INDEX].y) {
39           Eat();
40        }
41        for (int i = 0; i < SIZE_SNAKE - 1; i++) {
42           snake[i].x = snake[i + 1].x;
43           snake[i].y = snake[i + 1].y;
44        }
45        snake[SIZE_SNAKE - 1].y++;
46     }
47  }
48
49  void MoveUp() {
50     if (snake[SIZE_SNAKE - 1].y - 1 == 0) {
51        ProcessDead();
52     }
53     else {
54        if (snake[SIZE_SNAKE - 1].x == food[FOOD_INDEX].x && snake[SIZE_SNAKE -
   1].y - 1 == food[FOOD_INDEX].y) {
55           Eat();
56        }
57        for (int i = 0; i < SIZE_SNAKE - 1; i++) {
58           snake[i].x = snake[i + 1].x;
```

```
59          snake[i].y = snake[i + 1].y;
60      }
61      snake[SIZE_SNAKE - 1].y--;
62   }
63  }
```

Below is function-calling diagram

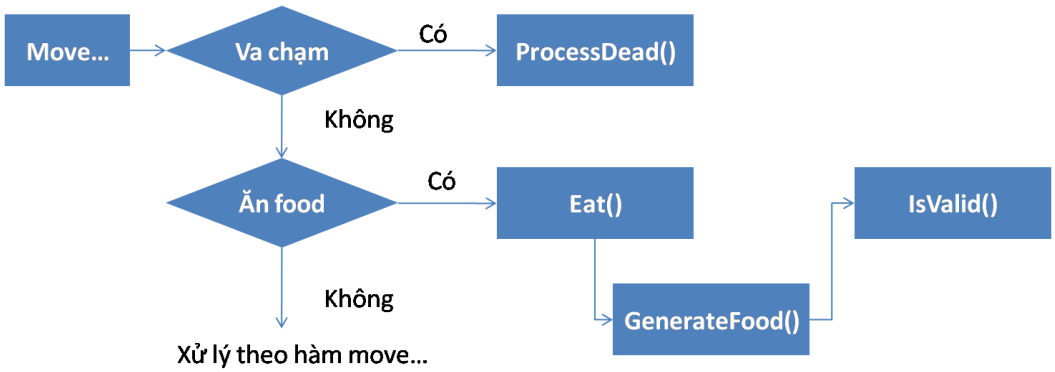



Figure 5: Function-calling diagram started from "Move…()"

Step 11: Next, we create a function continuously run when we start the thread in main().

```
1  //Subfunction for thread
2  void ThreadFunc() {
3     while(1) {
4        if (STATE == 1) {//If my snake is alive
5           DrawSnakeAndFood(" ");
6           switch (MOVING){
7              case 'A':
8                 MoveLeft();
9                 break;
10             case 'D':
11                MoveRight();
12                break;
13             case 'W':
14                MoveUp();
15                break;
16             case 'S':
17                MoveDown();
18                break;
19          }
20          DrawSnakeAndFood("O");
```

```
21          Sleep(1000 / SPEED);//Sleep function with SPEEED variable
22      }
23   }
24  }
```

In this function, we have an infinity loop. If STATE == 1, the snake is alive, and we check the variable MOVING processing moving. Otherwise STATE == 0, the snake is dead, and the loop does nothing (The screen keeps unchanged). The function "DrawSnakeAndFood(" ")" cleans the old-position of snake to let the function "DrawSnakeAndFood("0")" re-draw the snake at the new position. Then, we let the thread sleep (1000/SPEED). The bigger the SPEED, the lesser the thread sleeps and the faster the snake moves. In this function, we use "switch" to check the user's key-pressed.

Step 12: Finally, we build main function to connect all the above functions.

```
1  // main function
2  void main(){
3     int temp;
4     FixConsoleWindow();
5     StartGame();
6     thread t1(ThreadFunc); //Create thread for snake
7     HANDLE handle_t1 = t1.native_handle(); //Take handle of thread
8     while(1){
9        temp = toupper(getch());
10       if (STATE == 1) {
11          if (temp == 'P'){
12             PauseGame(handle_t1);
13          }
14          else if (temp == 27) {
15             ExitGame(handle_t1);
16             return;
17          }
18          else{
19             ResumeThread(handle_t1);
20             if ((temp != CHAR_LOCK) && (temp == 'D' || temp == 'A' || temp == 'W' || temp == 'S'))
21             {
22                if (temp == 'D') CHAR_LOCK = 'A';
23                else if (temp == 'W') CHAR_LOCK = 'S';
24                else if (temp == 'S') CHAR_LOCK = 'W';
25                else CHAR_LOCK = 'D';
26                MOVING = temp;
```

```
27              }
28          }
29      }
30      else {
31          if (temp == 'Y') StartGame();
32          else {
33              ExitGame(handle_t1);
34              return;
35          }
36      }
37  }
38  }
```

The first task of main function is to fix the size and disable the "maximum button" of console-window. Then, it calls "StartGame()" to prepare basic things, such as initial data or position. Next, "main()" creates a parallel thread. The function of this thread is "ThreadFunc()" built in step 9. Because we need to stop and destroy the thread, we must have its "HANDLE" used in "PauseGame()" and "ExitGame()".

Firstly, we allow the player to press a key. Then, if the STATE = 1 or STATE = 0, we will process correspondingly. If the snake is dead, we will ask if the player wants to continue or not. If he/she chooses 'y', we call "StartGame()" to start again. Otherwise, we call "ExitGame()" to quit the game. If the snake is alive, we check if the player pressed 'P', 'ESC' or not. If this holds, we call "PauseGame()" or ExitGame() correspondingly. If the player presses the "moving-key", we must check the direction of head of snake to lock the opposite direction in variable CHAR_LOCK (At a moment, there is one direction my snake cannot move).

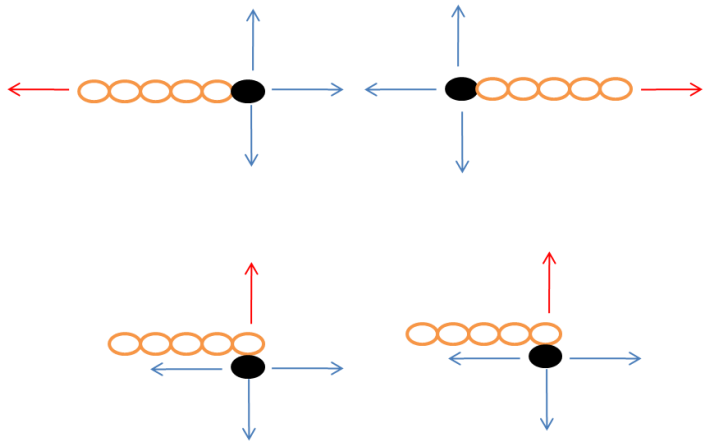


Figure 6: Moving directions of snake

To watch the function “main()”, we consider below diagram

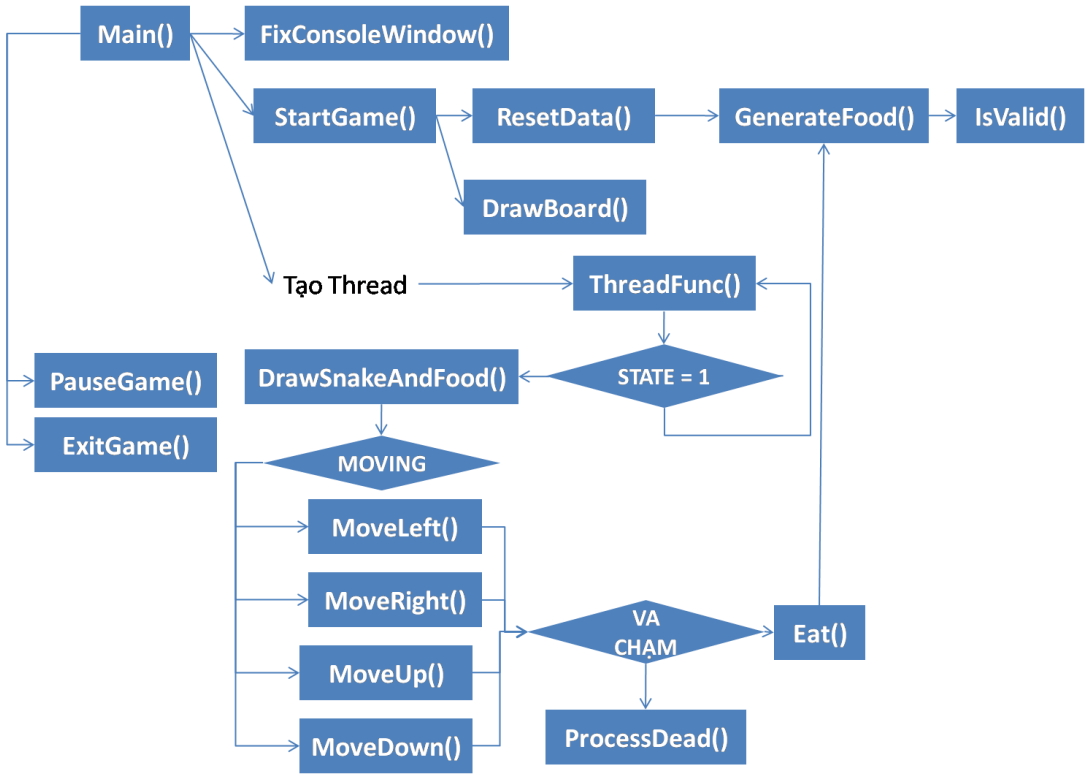



Figure 7: Function-calling diagram started from “main()”

# 4 REQUIREMENTS

In this guideline, we lack some basic features

## 4.1 Process the case the head of snake touches the its body

In the guideline, there is no solution of the case that the head of snake touches its body. So, we need to stop the game and ask if the player wants to continue or not?

## 4.2 Save/load

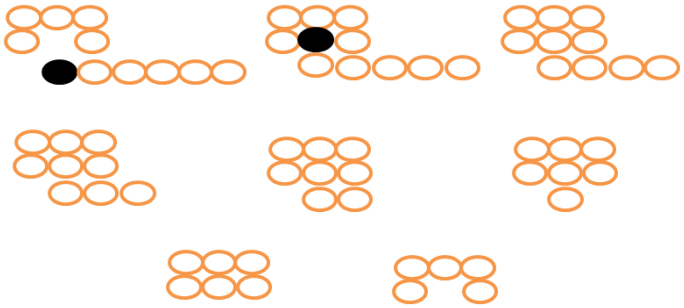
In this guideline, we cannot save and load the game. We need two features. When the players hit ‘L’, we show the line requesting the players provide the filename to save. When the players hit ‘T’, we show the line requesting the players provide the file to load.

## 4.3 Keeping the length unchanged

In the guideline, when moving the next level (SPEED++), the length of snake will reset (default is 6). We need to keep this length unchanged when moving to next level. Only when moving to MAX_SPEED, it will be reset.

## 4.4 Process the gate

In the guideline, when a snake eats 4 foods (just for illustration), the snake will be moved to next level automatically. However, we need to provide the gate for my snake come through. When my snake comes thought this gate completely, it will be moved to next level.



## 4.5 Provide animation

When snake touches the "wall", the its body, or final-gate, we create some lively animation for these events.

## 4.6 Provide main menu

When coming to boardgame, we print the menu game, for example, "New Game", "Load Game", "Settings", … So, this helps the players to easily choose actions they want.

Note: The body of snake consists of all student-IDs of the team. For example, the team has two students with two IDs: 1312918 and 1312920, then the initial body of snake is: "1312918", when snake eats "one food", **the 1st digit of the 2nd student id** will be added, for example "13129181". If the length of snake is "13129181312920", **the 1st digit of the 1st student id will be repeated**, for example: "131291813129201" …