NUST

"Think in other Terms"

Algorithms and data structures, first semester project.

Snake game implementation

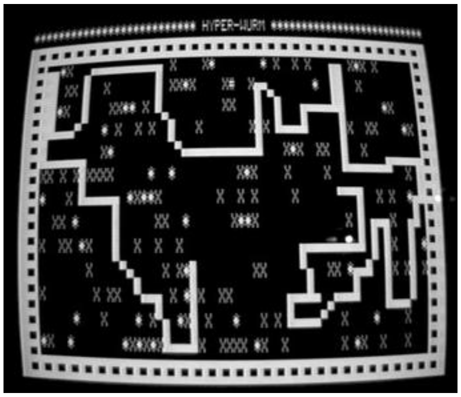
Lecturer: K. Mzelikahle
Student: Tendai D. Zhou, N0175499D

November 2018

Abstract

Snake is a classic game that was originally released in the 1976 arcade game Blockade. Many variations of the game have surfaced since then, all offering different types of challenges for example the campaign mode where the snake keeps growing until the head stumbles into the snakes tail and loses or the multi-player mode where multiple players control different snakes concurrently. I intend to implement the single player variant, where the player is supposed to avoid self collision. Below is a screen shot of the original snake released in the 1976 game, Blockade.

The classic snake game



Like the classic blockade game the player must avoid colliding with any of the walls or the tail in order avoid losing the game. The game will have five levels each with a slightly tougher set of walls to avoid than its predecessor. only the top 10 scores will be recorded on the online leader board.

I made use of the procedural development paradigm in conjunction with the agile development paradigm. I followed the evolutionary prototyping and incremental methodologies, within the bounds of these paradigms. These philosophies and methods provide a flexible and powerful development framework for reasoning and problem solving.

Contents

1 Introduction

1.1 Background

The original snake game and many games of the time were implemented on a state machine and not a general purpose computer. The circuit boards were custom build for a specific purpose for example pong, tetris, break breaker and snake, and where therefore rigid in nature. The circuitry required to implement the game was very complex and limited the possibilities for the game. Contemporary technology has improved drastically and as a result the burden has shifted from circuit design to software design and implementation. The currently available technology provides unlimited possibilities for system developer such as programmers, game designers and system analysts, with the availability of parallel processors, main memory in the order of Giga and disk space in the order of Tera.

1.2 Considerations

In order to fulfill all the requirements of the project the game will make use of the following concepts:

1. Game trees
2. Visualisation
3. Sorting algorithms
4. Searching algorithms
5. Decision making algorithms

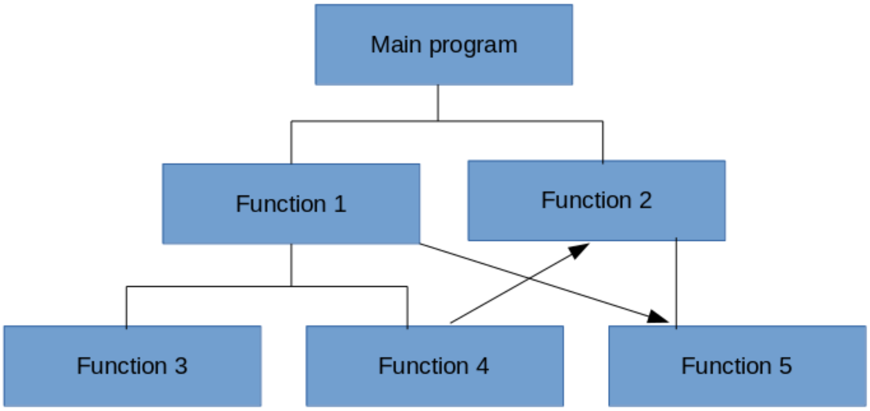
1.3 Defining the problem

I will create a 2 dimensional snake game that allows the player to manipulate the snakes path via an input device such as a key board , a touch screen or a joy stick. The objective of the game is for the player to amass as many points as possible by eating the fruit displayed on the screen while avoiding any collision with either the walls or the snake's tail. The game will also include a demo of how the game should be played.

Game development paradigms

1.4 Procedural development

In this project I followed the procedural development paradigm which focuses on the algorithms or steps necessary to solve a problem. It decomposes the problem into simpler sub-problems which can be represented by procedures, functions, or methods. It is relatively easy to follow this paradigm and it allows a smooth transition from design and analysis to implementation.



1.4.1 Advantages of procedural development

1. It is relatively simple for small projects as compared to other approaches such as OOP.

2. It is the most natural way to instruct a computer, because the micro processor's language (machine code) is procedural so the translation of a procedural high level language such as C is straight forward and efficient.

3. The ability to re-use the same code at different places in the program without copying it.

4. An easier way to keep track of program flow.

5. The ability to be strongly modular or structured.

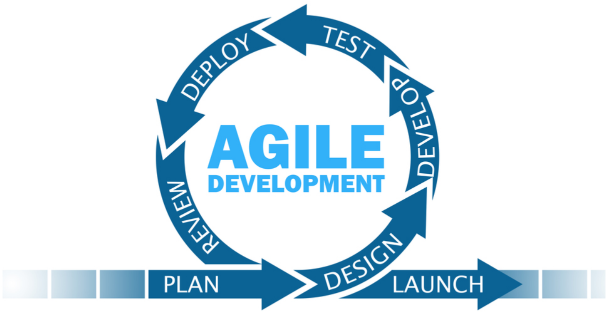
1.4.2 Disadvantages of procedural development

1. It is difficult to relate to real world objects.

2. Importance is given to the operation on data rather than the data itself.

1.5 The agile development paradigm

Agile development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self organisation cross functional teams. It advocates adaptive planning, evolutionary development, early delivery, continual improvement and it encourages rapid and flexible response to change. Below is a diagrammatic representation of the agile development philosophy.



1.5.1 Advantages of agile development

1. **High productivity**
   In agile development, testing integrated during the cycle, which means that the product is working during the development. This enables the product owner to make changes if needed and the team is aware if there are any changes.

2. **Increased project control**
   There is high transparency, both the developers and end users can easy access and contribute to the development the system easily. There is high visibility of each step of the project for both parties.

3. **Reduced risk.**
   Agile methodologies minimise the chance of complete project failure. Since it advocates for short cycles there is always a working product at any stage of the project after the first cycle. Iterative development ensures a short time between initial project investment and either, failing fast or knowing that a product or approach work will work.

4. **Faster ROI**
   The fact that agile development is iterative means that the features are delivered incrementally, therefore benefits are realised early while the product is under development. A functional, product which is ready for market it developed after just a few iterations.

1.5.2 Disadvantages of agile development

1. **Documentation might be insufficient**
   It places less focus on documentation and focuses on the product development itself and as a result the documentation of the system might be insufficient to fully understand the system.

1.6 The prototyping methodology

Uses system prototypes as deliverables

1.7 The incremental methodology

Focus on features that should be delivered on each iteration.

1.8 Game play

The player controls the snake either using the arrow keys or the W, A, S and D keys to turn in the desired direction. The objective of the game is to earn as many points as possible by eating the food which in turn results in the growing of the snake. Each time the snake eats the food it relocates to a new random location. While trying to accumulate as many points as possible the player also has to avoid colliding with any present obstacles such as the walls or tail of the snake. Colliding with anything other than the food, results in the death of the snake, and therefore the end of the game.

2 Analysis

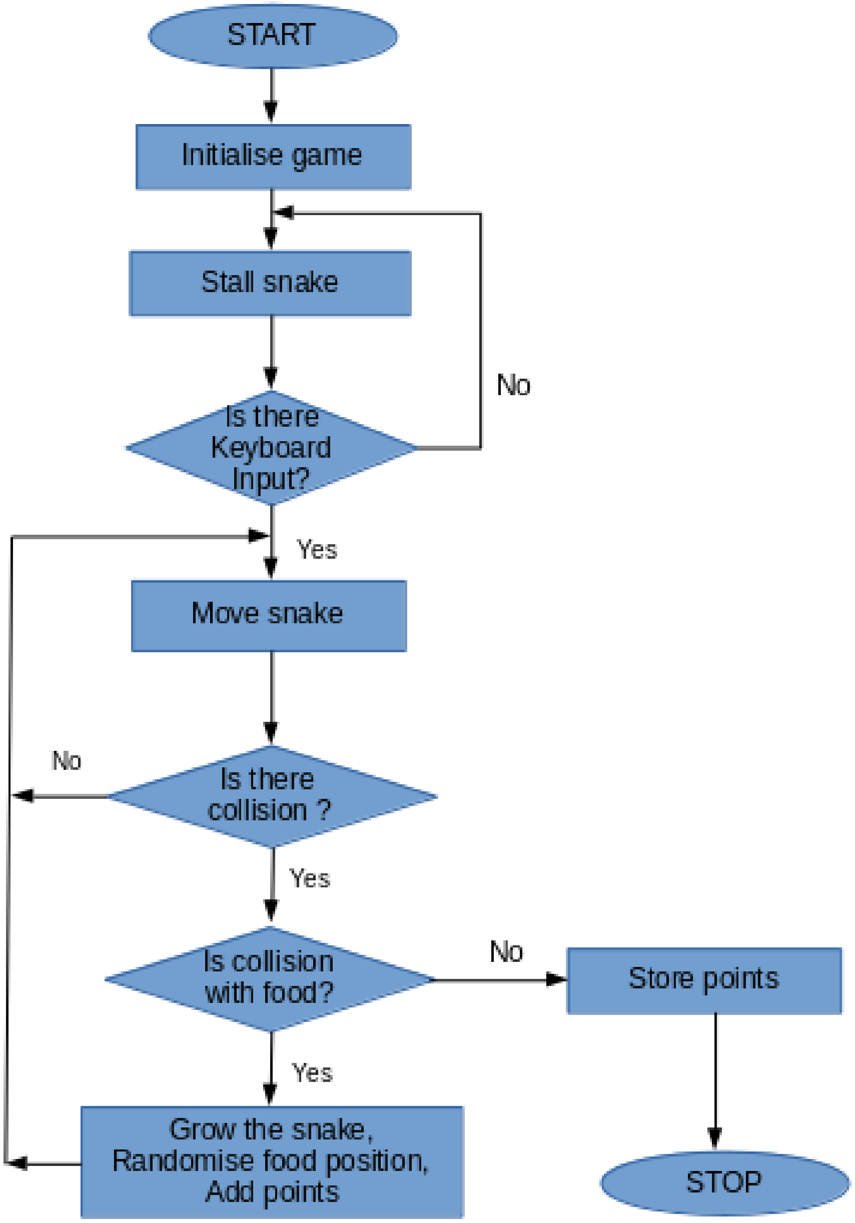
2.1 Decomposing the problem statement

The following functional requirements where deduced from the problem statement stated earlier.

2.2 Functional requirements (Design specification)

1. The snake must be able to move around the screen.
2. The snake must turn in response to the user's input.
3. The snake's length increases each time it swallows its food.
4. The food randomly relocates to another position on the screen each time its eaten.
5. The snake dies if it collides with its tail or any of the walls.
6. The snake must never stop moving if the game is in progress.
7. The player cannot make the snake go in reverse.
8. The game should include a demo mode, to teach the player how it is played.

2.3 Overall system flow chart



The snake will comprise of multiple segments of the same type. Each segment will have its graphics change to reflect its current state (position and orientation). Each node will store (encapsulate) information about its current state internally.

2.4 Space Complexity

The snake will be dynamically allocated more memory by the operating system each time it grows. The snake will be limited to a total number 12 segments at any given level. Each node will store an SDL_Rect data type to hold the nodes position on the screen, a char data type to store its current orientation and a next variable which will store the child of this current segment. The linear search algorithm which is used for pattern recognition when rendering the snake has a space complexity of O(1).

2.5 Time Complexity

A simple linear search algorithm is used for pattern recognition when rendering the snake. This has a O(N) because each element is compared only once.

3 Design

Since the analysis phase focuses on defining the requirements of the system that are independent of any implementation details, the non-functional requirements could not be included. The non-functional requirements are as follows.

3.1 Non functional requirements

1. The game will be implemented using the C programming language.

2. The Simple direct media layer version 2 library will be used for detecting user input, rendering the graphics and playing sounds.

3. Web technologies such as HTML, Javascript, CSS and PHP for an online game score board.

4. An online mysql database will be used to store the high scores.

5. Unit testing is achieved by making use of the Check unit testing framework.

3.2 Game manager (Singleton pattern)

The game manager is a globally defined game object which has the role of housing all the persistent information in the whole game. There should only be one copy of this object during execution of the program for the intended behavior to occur.
The structure of the singleton pattern is implemented as follows:

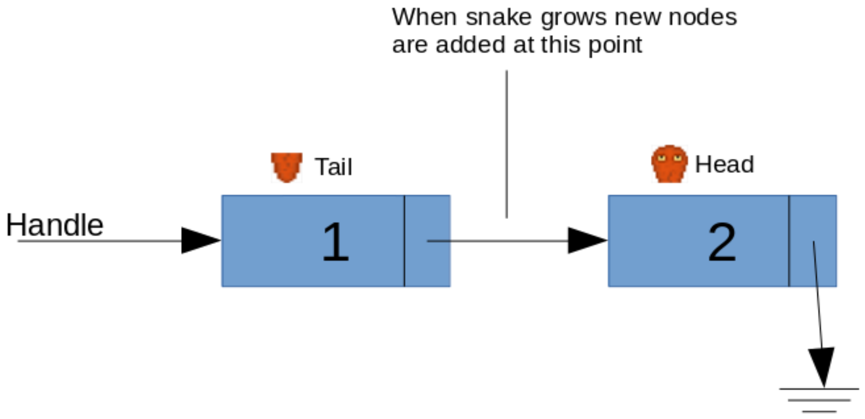
```
void game_init(void);
void game_quit(void);

static struct {
// define attributes
SDL_bool running;

// define methods
void (*init)(void);
void (*quit)(void);

} Game = {

SDL_FALSE,
game_init ,
game_quit

};
```

The game manager has both attributes (properties) as well as functions (methods or behaviors) as shown above.

The responsibilities of the game manager are as follows:

1. Initialise and start the game.
2. Keep track of the player's current score.
3. Allow the player to pause, restart or quit the game.
4. Keep track of the location of the snake.
5. Keep track of the location of the food.
6. Keep track of the current game state.

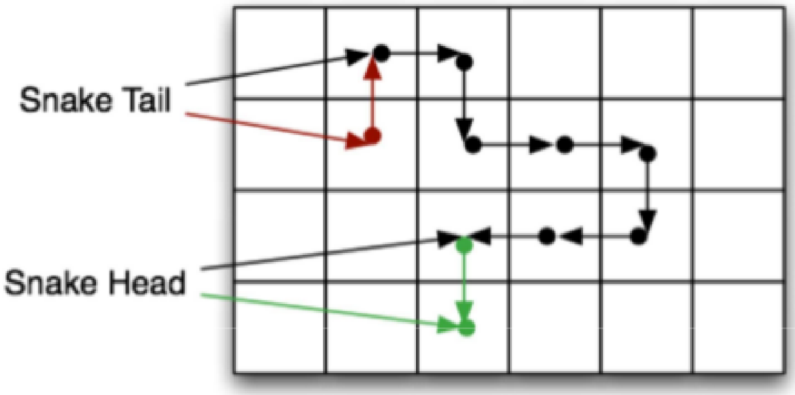
3.3 The snake (Singly linked list)



The snake is implemented as a singly linked list, where the handle points to the tail of the snake. The tail is the point where new nodes are added each time the snake eats a fruit. The snake data structure is a stack which only allows new nodes to be pushed on it and prohibits popping of the list. The diagram below shows how the snake segments are used to model the snake. Each segment has a next pointer which points to the segment whose position it will occupy when the snake moves forward.The direction variable which is the char data type, of each segment points to the direction of the next node relative to itself. These next pointers allow the head and tail of the snake to be advanced easily.
Each segment of the snake is defined by the following struct:

```
typedef struct head {

    SDL_Rect headRect;  //Position of this node
    char dir;   //Direction this node is facing
    struct head *next;  //Parent of this node

}         Head;
```

The snake data structure is diagrammatically illustrated below.



Red nodes and pointers represent the state of the snake before the tail moves. Green nodes and pointers represent the new state of the snake after the head moves.

3.4 Displaying graphics to the screen

In the initial procedure used to generate and display graphics, a collection of textures where stored in video memory and used by the renderer to draw onto screen. However in terms of memory usage this is an inefficient approach, as it wastes memory. I opted for an atlas texture (meta-sprites) because it conserves memory and has a lesser memory overhead compared to multiple textures.

Initial code before memory optimisation is as follows:

```
#include <SDL2/SDL.h>
#include <stdio.h>

int main(int argc, char** argv)
{
 SDL_Texture tex1 = SDL_CreateTextureFromSurface(Game.screen.renderer,
  surface1, -1);
 SDL_Texture tex2 = SDL_CreateTextureFromSurface(Game.screen.renderer,
  surface2, -1);
 SDL_Texture tex3 = SDL_CreateTextureFromSurface(Game.screen.renderer,
  surface3, -1);
 SDL_Texture tex4 = SDL_CreateTextureFromSurface(Game.screen.renderer,
  surface4, -1);
}
```

A snip of the code showing the implementation of the atlas texture is shown below.

```
{
...

int i, j;
        SDL_Rect rect = {0, 0, 8, 8};
        for(j=0; j<h; j++) {
                for(i=0; i<w; i++) {
                        rect.x = i*8;
                        rect.y = j*8;
                        SDL_BlitSurface(
                        Game.gfx.spritesheet[grid[j][i]],
                        NULL,
                        surface,
                        &rect);
                }
        }

SDL_Texture* tex =
SDL_CreateTextureFromSurface(
Game.screen.renderer,
        surface);
}
```

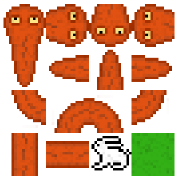
As show in the snippet of code above only a single texture is used to store all the images.

3.5 Forward movement of the snake

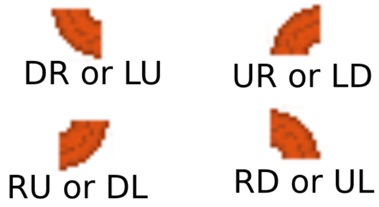
One of the functional requirements of the game is that the snake must always move when the game is running. This is achieved by the use of a loop whose condition is the game status variable. The number of iterations of this loop is limited by two variable which make sure that the game runs at 30 frames per second. These variables determine the frame rate by making use of the clock ticks function (SDL_GetTicks()) found in the SDL2 library. Limiting the frame rate makes sure that the screen is not updated at a rate which is greater then the screen refresh rate because this would result in the wastage of system resources. Within this loop the snake head is move one step per unit time in the direction the head is facing. Since the handle references the location of the tail, a loop is used to iterate to the head node of the snake. Once the head node is reached, the head is moved one step in the direction it is facing. Every other node whose next pointer is not pointing to NULL is moved to the position of the node its next pointer is referencing. In this way the snake is kept moving continuously.

3.6 Pattern recognition and displaying the snake

In order to render each segment of the snake, both its position and orientation are considered. The position of a segment is stored in a SDL_Rect data type, which stores the x and y position of the node on the screen. The orientation is stored using a char data type, with "U" representing upwards direction , "D" downwards, "R" representing the right and "L" representing left. Using this encapsulated information each node can be displayed on the screen, after variables of the following type have been created an initialised: SDL_Window, SDL_Renderer, SDL_Surface and the SDL_Texture.
The SDL_Window pointer is used to store a reference to the window in which the graphics will be displayed. The SDLRenderer pointer stores a render associated with the window. The SDL_Surface variable is used to store a texture in main memory (RAM) before it is used. The SDL_Texture is used to store an image in the graphics hardware (Video RAM) before it is finally displayed. After the display environment has been setup each node can be rendered on the screen. This is achieved by making use of a while loop to traverse the entire linked list starting from the tail. The snake segments are rendered using the sprite sheet shown below.



The snake head can be rendered using any one of the four possible images depending on the direction it is facing. A node which in the middle thus a node which is neither the head nor the tail has can take any one of the six possible images at any time depending its predecessor. If its predecessor was facing the same direction it is facing a straight segment is used and if not an appropriate turning graphic is used to display it. When the while loop iterates the snake list it looks for patterns which correspond to a particular image. Bellow is a table of patterns and their corresponding image graphic. Where the letters D, L, U and R stand for down, left, up and right respectively.



3.7 Input detection

Input is detected by the use of the SDL_Event data type which can handle all the possible events that SDL can handle. I make use of a while loop as show below, in order to poll for the input events as show below.

```
SDL_Event event;

 /* Use SDL_PollEvent() function which returns 0 when there are no
 more events on the event queue, as the while loop condition to poll
 events. When SDL_PollEvent() returns 0, the while loop is exited. */


while( SDL_PollEvent( &event ) )
{
  //I am only conserned about SDLKEYDOWN events
  switch(event.type )
  {
    case SDLKEYDOWN:
      //Key handling code goes here.
      break;

    default:{}
  }
}
```

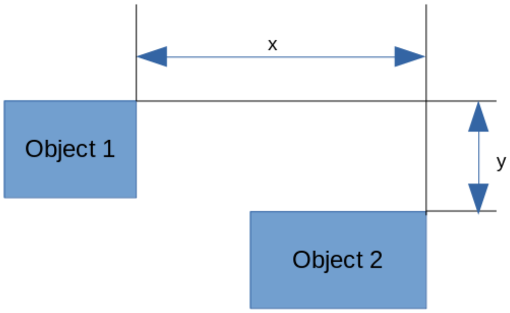
Pressing the left arrow key causes the snake to turn left if it is legal turn and so do the other arrows in their respective directions. If the snake is moving either upward or downward the only legal turn it can make is only left or right and conversely if it is moving left or right it can only turn up or down. The function which checks if a requested turn is permissible or not is as follows:

```
SDL_bool requestTurn(char dir, Head *tail)
{
        Head *head;
            while(tail != NULL)
            {
                head=tail;
                tail = tail->next;
            }
    if(head->dir == 'U' || head->dir == 'D')
    {
        if(dir == 'L' || dir == 'R')
        {
            head->dir = dir;
            return SDL_TRUE;
        }
    }
    else if(head->dir == 'L' || head->dir == 'R')
    {
        if(dir == 'U' || dir == 'D')
        {
            head->dir = dir;
            return SDL_TRUE;
        }
    }
    return SDL_FALSE;
}
```

This function returns a boolean value of the type SDL_bool, which can take the value SDL_TRUE or SDL_FALSE. When the requestTurn() function is called and it returns the value SDL_TRUE it means that the requested turn is legal otherwise if SDL_FALSE is returned it is illegal.

3.8 Collision detection

Collision detection is handled by tracking the edges of the objects as shown in the diagram below:



If the distance between the right edges or the top edges of the objects is less than the length or height of the objects it means that a collision has occurred.

The x and y distances are calculated as follows:

$$x <= |object1RightEdgePos - object2RightEdgePos|$$

$$y <= |object1TopEdgePos - object1TopEdgePos|$$

The collision detection method is implemented as followes:

```
SDL_bool detectCollision(SDL_Rect rectFoodPos, Head *tail)
{
        Head *snakeHead;
    while(tail != NULL)
    {
        snakeHead=tail;
        tail = tail->next;
    }

        SDL_Rect a = rectFoodPos;
        SDL_Rect b = snakeHead->headRect;
        //The sides of the rectangles
        int rightA, rightB;
        int topA, topB;
        //Calculate the sides of rect A
        rightA = a.x + a.w;
        topA = a.y;
        //Calculate the sides of rect B
        rightB = b.x + b.w;
        topB = b.y;

        if((abs(rightA-rightB)<=16) && (abs(topA-topB)<=16))
        return SDL_TRUE;
        else
                return SDL_FALSE;
}
```

3.9 Fruit Randomisation

Randomisation of the food's position is necessary in order to keep the game interesting an unpredictable. The fruit position is randomised by the use of the srand() function found in the standard library of the C language. The srand() functions takes a seed value as its argument and it must be unique in order to generate random numbers.The clock tick value obtained by calling the SDL_Ticks() method is passed in as the argument into srand() function because it is guaranteed to be different on each call. In order to limit the value generated by the srand function the modulo operator.

This ensures that the food objects never goes off screen. The modulo operator is used used as follows:

$$x = random\%width$$

$$y = random\%height$$

3.10 Game levels

This game has five unique levels, which differ in the amount of difficulty. The higher the level, the more challenging it becomes. In order to get to the next level, the snake must eat the fruit twelve times. When all five levels have been completed the levels wrap around the snake's speed increases. This continues until the snake collides with object.

3.11 Restarting the Game and Storing the high score

Restarting of the game is handled by the game manager. This is done by terminating the current game session and re-initialising the game state. The current score is part of the players persistent data, and has to be considered for storage in the leader score board. This is handled by the game manger object which encapsulates the player score variable. If the player score qualifies him/her to be in the top ten the uploadScore() function is called with the score as argument. The uploadScore function makes use of the insertion sort algorithm because the list of scores is already sorted and it merely needs to add the current score in its correct place. The function also takes advantage of the jsm library in order to use the JSON format standard for simplicity when storing or retrieving data from the high score server.

Storing the high score
The current score of the player is compared to the lowest entry on the score board in order to determine if it qualifies the player to be on the score board. If the qualifies,his/her score is inserted into the list by making use of the insertion sort algorithm. The pseudo code of the insertion sort algorithm is as follows (Geeks for geeks, insertion sort) :

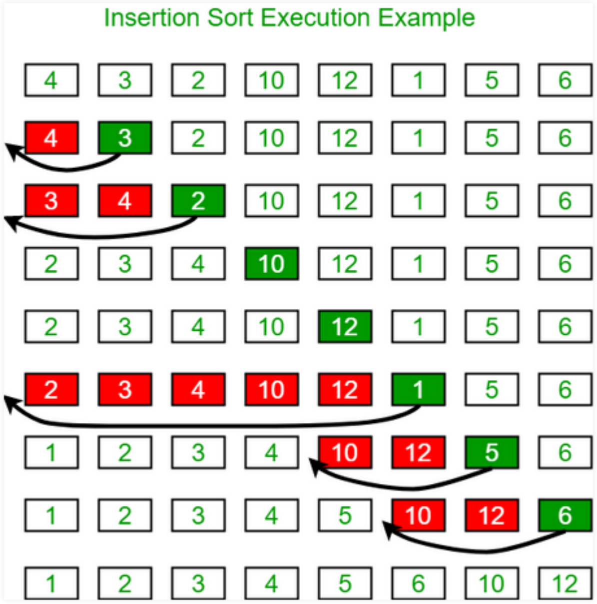
```
// Sort an array of size n
insertionSort(array, n)
Loop from i = 1 to n-1.
Pick element arr[i] and insert it into sorted sequence array[0 to i - 1]
```

The insertion sort algorithm is implemented as follows:

```
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
         key = arr[i];
         j = i-1;

         /* Move elements of arr[0..i-1], that are
             greater than key, to one position ahead
             of their current position */
         while (j >= 0 && arr[j] > key)
         {
              arr[j+1] = arr[j];
              j = j-1;
         }
         arr[j+1] = key;
    }
}
```

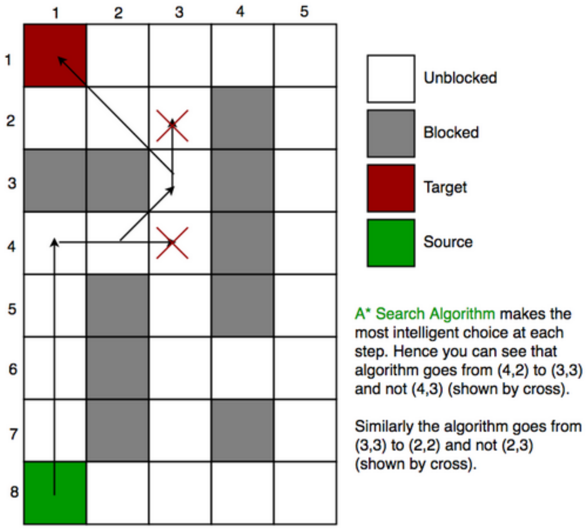
This ensures that the current new high score is placed in the correct position in order to avoid unnecessary sorting in the future which would use more system resources.

The following is an illustration of the execution of the insertion sort algorithm.



3.12 Computer AI (Demo mode)

The purpose of the computer AI is to operate the game while it is in demo mode, to show the player how the game should be players. The computer AI is based on a recursive search algorithm which finds the closest food item and follows a path to it. At the core of this algorithm is a function which determines the distance to the nearest food item. This function is like a recursive maze search algorithm, where each recursive step is to search all of the cells adjacent to a given cell. In order to determine distance to food, however, this algorithm must also mark entries in a distance map with the distance traveled thus far. Also, in order to avoid wasteful searching, the algorithm must also consider any cell with a distance less than or equal to the distance traveled to be a barrier, because searching it would result in the same or worse results. This search algorithm is know as the A* search algorithm.



A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

The A* search algorithm is one of the best and most popular techniques used in path-finding and graph traversals problems.

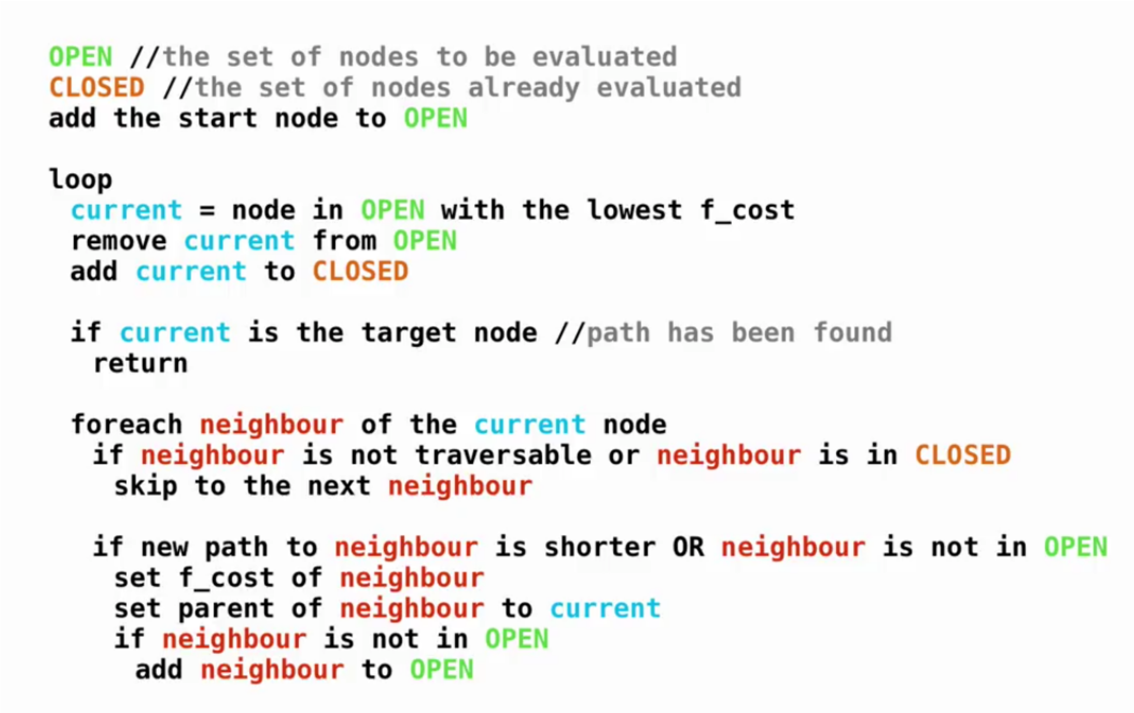
Explanation of the A* search algorithm

Considering a square grid having many obstacles, given a starting point cell and a target or destination cell. The objective is to reach the target cell from the current cell as quickly as possible. A* selects the quickest path by considering various heuristics, namely the F cost, G cost and the H cost.
G cost is the distance from the starting point to the current cell.
H cost is the distance from the current cell to the target cell.
F cost is the sum of the G cost and the H cost.

Limitations of the A* search algorithm

Although it is the fastest algorithm, it does not always find the shortest path to a given point, as it relies heavily on heuristics or approximations to calculate h. The A* search algorithm pseudo code is as follows:

Pseudo code for A* search algorithm



```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

4 Implementation

4.1 The Makefile.

I made use of a make file to make simplify the compilation process and make it as convenient as possible. The game is implemented in modules and as a result of this there are multiple source files. Therefore using a make file is the most convenient and least tedious approach possible. The make file is implemented as follows:

```
default:
        cc -c ‘sdl2-config --cflags‘ main.c -o main.o
        cc -c snake.c -o snake.o
        cc -c walls.c -o walls.o
        cc -c ‘sdl2-config --cflags‘ gui.c -o gui.o
        cc -c ‘sdl2-config --cflags‘ bonusTimer.c -o bonusTimer.o
        cc -c ai.c -o ai.o

link_them:main.o snake.o walls.o gui.o bonusTimer.o ai.o
        cc main.o snake.o walls.o gui.o bonusTimer.o ai.o
        -I/usr/include/SDL2 ‘sdl2-config --cflags --libs‘
        -lSDL2_image -lSDL2_ttf -Wall -g -o snake2d

clean:main.o snake.o walls.o gui.o bonusTimer.o ai.o
        rm main.o snake.o walls.o gui.o bonusTimer.o ai.o
```

4.2 The game manager file, main.c

```
#include <stdio.h>
#include <stdlib.h>
#define SDL_MAIN_HANDLED
#include <SDL2/SDL.h>
#include <SDL2/SDL_timer.h>
#include <SDL2/SDL_image.h>
#include <math.h>
#include ”head.h”
```

```
#include "snake.h"
#include "walls.h"
#include "gui.h"
#include "scoreBoard.h"
#include "ai.h"

#define SCREEN_w 640
#define SCREEN_h 480
#define SCREEN_SCALE 1
#define SCREEN_NAME "Snake2d"
#define LIMIT_X_H 640
#define LIMIT_X_L 0
#define LIMIT_Y_H 480
#define LIMIT_Y_L 0

SDL_Rect rectFoodPos;

void displaySnake();
void displayFood();
void setupSpriteSheet();
void randomiseFood(int min, int max);
void eatFood();
void nextLevel();
void updateScore();
void resetSnake();
void createSnake(Head * snakeHead);

void game_init(void); //game object initialisation
void game_quit(void); //game object clean up

//This is the main game object, and it is a singleton
static struct {
  //define "attributes"
  SDL_bool running;
  Head * tail;

  struct {
    unsigned int w;
    unsigned int h;
    const char * name;
    SDL_Window * window;
    SDL_Renderer * renderer;
  }
  screen;

  struct {
    unsigned int n;
    SDL_Surface ** spritesheet;
    SDL_Surface * surfaceBack;
  }
  gfx;

  //define ”methods”
  void( * init)(void);
  void( * quit)(void);

}
Game = {
  SDL_FALSE,
  NULL,
  {
    SCREEN_SCALE * SCREEN_w,
    SCREEN_SCALE * SCREEN_h,
    SCREEN_NAME,
    NULL,
    NULL
  },
  {
    0,
    NULL,
    NULL
  },
  game_init,
  game_quit
};
```

```
unsigned int lastTime = 0, currentTime;
unsigned int lastSnakeMoveTime = 0, currentSnakeMoveTime;
SDL_bool nextFrame = SDL_FALSE;
int frameWait = 200;
SDL_bool start = SDL_FALSE;
int level = 1, levelChange = 0;
int fruitsEaten = 0;
int points = 0;
char score[8];
SDL_Texture * texureGUI;
SDL_Texture * textureBack;


int main(int argc, char ** argv) {
  Game.init();
  SDL_Event event;

  updateScore();

  rectFoodPos.w = 16;
  rectFoodPos.h = 16;

  textureBack = SDL_CreateTextureFromSurface(
    Game.screen.renderer, Game.gfx.surfaceBack);

  texureGUI = SDL_CreateTextureFromSurface(
    Game.screen.renderer, displayGUI(score));

  SDL_Rect guiRect;
  guiRect.x = 600;
  guiRect.y = 20;
  guiRect.w = 20;
  guiRect.h = 20;

  if (textureBack == NULL) {
    printf("Error creating texture: %s", SDL_GetError());
    return EXIT_FAILURE;
  }
```

```
SDL_Rect rectBack = {
  0,
  0,
  640,
  480
};

Head * snakeHead = (Head * ) malloc(sizeof(Head));
createSnake(snakeHead);
resetSnake();
randomiseFood(SCREEN_h - 16, SCREEN_w - 16);

while (Game.running) {
  if (nextFrame == SDL_TRUE) {

    if (detectCollision(rectFoodPos, Game.tail) == SDL_TRUE) {
      srand(SDL_GetTicks());
      eatFood();
      if (levelChange != level) {
        textureBack = SDL_CreateTextureFromSurface(
          Game.screen.renderer, Game.gfx.surfaceBack);
        levelChange = level;
      }

      randomiseFood(SCREEN_h - 16, SCREEN_w - 16);

    }
    while (SDL_PollEvent( & event)) {
      switch (event.type) {
      case SDL_QUIT:
        Game.running = SDL_FALSE;
        break;
      case SDL_KEYDOWN:
        switch (event.key.keysym.scancode) {
        case SDL_SCANCODE_0:
          start = SDL_FALSE;
          break;
  case SDL_SCANCODE_W:
  case SDL_SCANCODE_UP:
    if (start == SDL_FALSE)
      start = SDL_TRUE;
    if (requestTurn('U', Game.tail)) {
      makeTurn('U', Game.tail);
    }
    break;
  case SDL_SCANCODE_A:
  case SDL_SCANCODE_LEFT:
    if (start == SDL_FALSE)
      start = SDL_TRUE;
    if (requestTurn('L', Game.tail)) {
      makeTurn('L', Game.tail);
    }
    break;
  case SDL_SCANCODE_S:
  case SDL_SCANCODE_DOWN:
    if (start == SDL_FALSE)
      start = SDL_TRUE;
    if (requestTurn('D', Game.tail)) {
      makeTurn('D', Game.tail);
    }
    break;
  case SDL_SCANCODE_D:
  case SDL_SCANCODE_RIGHT:
    if (start == SDL_FALSE)
      start = SDL_TRUE;
    if (requestTurn('R', Game.tail)) {
      makeTurn('R', Game.tail);
    }
    break;
  default:
    {}
  }
  break;
default:
  {}
```

```
    }
  }
    SDL_RenderClear(Game.screen.renderer);

    SDL_RenderCopy(Game.screen.renderer,
    textureBack, NULL, &rectBack);


    displayFood();

    displaySnake();

    SDL_RenderCopy(Game.screen.renderer,
    texureGUI, NULL, &guiRect);


    SDL_RenderPresent(Game.screen.renderer);
}

currentTime = SDL_GetTicks();

currentSnakeMoveTime = SDL_GetTicks();

if (currentTime > lastTime + 200) {
  lastTime = currentTime;
  nextFrame = SDL_TRUE;
} else
  nextFrame = SDL_FALSE;

if (currentSnakeMoveTime > lastSnakeMoveTime + frameWait) {

  lastSnakeMoveTime = currentSnakeMoveTime;

  if (start == SDL_TRUE) {
    moveForward(snakeHead -> dir, Game.tail);
    if (checkWallCollision(level,
    & snakeHead -> headRect) == SDL_TRUE) {
            //game over you collided with the wall
```

```
                printf(”Game␣over\n”);
                resetSnake();
                points = 0;
                start = 0;
                updateScore();
                level=1;
                nextLevel();
        textureBack = SDL_CreateTextureFromSurface(
        Game.screen.renderer, Game.gfx.surfaceBack);
        }
      }
    }
  }

  SDL_DestroyTexture(textureBack);
  Game.quit();
  return EXIT_SUCCESS;
}

void resetSnake() {
  while (Game.tail -> next -> next != NULL) {
    free(Game.tail);
    Game.tail = Game.tail -> next;
  }
  Game.tail -> headRect.x = SCREEN_w / 2;
  Game.tail -> headRect.y = SCREEN_h / 2;

  Game.tail -> dir = ’U’;

  Game.tail -> next -> headRect.x = SCREEN_w / 2;
  Game.tail -> next -> headRect.y = SCREEN_h / 2 - 16;
  Game.tail -> next -> dir = ’U’;

  initialiseGrid(level);
}

void createSnake(Head * snakeHead) {
  if (!Game.tail)
    Game.tail = (Head * ) malloc(sizeof(Head));

  Game.tail -> headRect.x = SCREEN_w / 2;
  Game.tail -> headRect.y = SCREEN_h / 2;
  Game.tail -> headRect.w = 16;
  Game.tail -> headRect.h = 16;

  Game.tail -> dir = 'U';

  Game.tail -> next = snakeHead;
  snakeHead -> next = NULL;

  snakeHead -> headRect.x = SCREEN_w / 2;
  snakeHead -> headRect.y = SCREEN_h / 2 - 16;
  snakeHead -> headRect.w = 16;
  snakeHead -> headRect.h = 16;
  snakeHead -> dir = 'U';
}

void updateScore() {
  sprintf(score, "%d", points);
  texureGUI = SDL_CreateTextureFromSurface(
    Game.screen.renderer, displayGUI(score));
}

void nextLevel() {
  switch (level) {
  case 1:

    Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level1.png");

    break;
  case 2:

    Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level2.png");
```

```
    break;
  case 3:

    Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level3.png");

    break;
  case 4:

    Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level4.png");

    break;
  case 5:
    Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level5.png");

    break;
  default:
    {}
  }
  resetSnake();
}

void randomiseFood(int min, int max) {
  while(1){
        srand(SDL_GetTicks());
        rectFoodPos.x = rand() % max;
        rectFoodPos.y = rand() % min;
        if (checkWallCollision(level,&rectFoodPos) == SDL_FALSE)
        {
              Head *head = Game.tail;
              while(head->next != NULL)
                      head = head->next;

              SDL_Rect temp;
              temp.x = 0;
```

```
                temp.y = 0;


            //if(findPath(rectFoodPos,head->headRect) == SDL_TRUE)
            //  printf("Path found");
        /*
            Cell c1 = cellFromPos(head->headRect);
            Cell c2 = cellFromPos(rectFoodPos);
            printf("Snake to food distance: %d\n",getDistance(&c1, &c2)
            */


            printf("list␣has␣%d␣cells\n",openSetCount());

                break;
          }
  }

}

void eatFood() {
  points++; //Add points
  Head * node = (Head * ) malloc(sizeof(Head));
  updateScore();

  switch (Game.tail -> dir) {
  case 'D':
    node -> headRect.x = Game.tail -> headRect.x;
    node -> headRect.y = Game.tail -> headRect.y - STEP;
    break;
  case 'U':
    node -> headRect.x = Game.tail -> headRect.x;
    node -> headRect.y = Game.tail -> headRect.y + STEP;
    break;
  case 'L':
    node -> headRect.x = Game.tail -> headRect.x + STEP;
    node -> headRect.y = Game.tail -> headRect.y;
    break;
  case 'R':
    node -> headRect.x = Game.tail -> headRect.x - STEP;
    node -> headRect.y = Game.tail -> headRect.y;
    break;
  default:
    {}
  }
  node -> dir = Game.tail -> dir;
  node -> next = Game.tail;
  Game.tail = node;

  fruitsEaten++;
  if (fruitsEaten >= 12) {
    level++;
    nextLevel();
    fruitsEaten = 0;
  }
  /*wrap around and go back to
   first level with greater speed*/
  if (level > 5) {
    //increase speed
    level = 0;
  }
}

void moveForward(char dir, Head * tail) {
  Head * snakeHead = tail;
  while (snakeHead -> next != NULL) {
    snakeHead = snakeHead -> next;
  }

  Head * node = tail;
  while (node -> next != NULL) {
    node -> headRect = node -> next -> headRect;
    node -> dir = node -> next -> dir;
    node = node -> next;
  }
  switch (dir) {
  case 'R':
    snakeHead  -> headRect.x += STEP;
    //wrap snake around the screen
    if (snakeHead -> headRect.x > LIMIT_X_H - 8)
      snakeHead -> headRect.x = LIMIT_X_L;
    break;
  case 'L':
    snakeHead -> headRect.x -= STEP;
    //wrap snake around the screen
    if (snakeHead -> headRect.x < LIMIT_X_L)
      snakeHead -> headRect.x = LIMIT_X_H - 16;
    break;
  case 'U':
    snakeHead -> headRect.y -= STEP;
    //wrap snake around the screen
    if (snakeHead -> headRect.y < LIMIT_Y_L)
      snakeHead -> headRect.y = LIMIT_Y_H - 16;
    break;
  case 'D':
    snakeHead -> headRect.y += STEP;
    //wrap snake around the screen
    if (snakeHead -> headRect.y > LIMIT_Y_H - 8)
      snakeHead -> headRect.y = LIMIT_Y_L;
    break;
  default:
    {}
  }
}

void displayFood() {
  SDL_Texture * texture = SDL_CreateTextureFromSurface(
Game.screen.renderer, *(Game.gfx.spritesheet + 15));

  SDL_RenderCopy(Game.screen.renderer, texture, NULL,
  &rectFoodPos);
  SDL_DestroyTexture(texture);
}

void setupSpriteSheet() {
  SDL_Surface * surface = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/Snake.png");

  if (surface == NULL) {
    printf("Error␣while␣openning␣spritesheet:\n%s\n", SDL_GetError());
    exit(EXIT_FAILURE);
  }

  int n = ((surface -> w / 16) * (surface -> h / 16) + 1);

  Game.gfx.n = n;
  Game.gfx.spritesheet =
(SDL_Surface ** ) malloc(sizeof(SDL_Surface * ) * n);

  Game.gfx.surfaceBack = (SDL_Surface * ) malloc(sizeof(SDL_Surface));

  Game.gfx.surfaceBack = IMG_Load(
"/home/steamrolle/Documents/SOURCE_CODE/C/Snake2D/level1.png");

  if (Game.gfx.surfaceBack == NULL) {
    printf("Error␣openning␣image:␣%s\n", SDL_GetError());
    exit(EXIT_FAILURE);
  }

  int i, x, y;
  SDL_Rect rect = {
    0,
    0,
    16,
    16
  };
  for (i = 0; i < n; i++) {
    Game.gfx.spritesheet[i] =
SDL_CreateRGBSurface(0, 16, 16, 32, 0x00, 0x00, 0x00, 0x00);
```

```
    SDL_SetColorKey(Game.gfx.spritesheet[i], 1, 0xFF00FF);

    SDL_FillRect(Game.gfx.spritesheet[i], 0, 0xFF00FF);

    if (i != 0) {
      x = (i - 1) % (surface -> w / 16);
      y = (i - x) / (surface -> w / 16);
      rect.x = x * 16;
      rect.y = y * 16;
      SDL_BlitSurface(surface, &rect, Game.gfx.spritesheet[i],
NULL);

    }
  }
  SDL_FreeSurface(surface);
}

void game_init(void) {
  if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
    printf("Error␣initialisation␣SDL␣%s\n", SDL_GetError());
    exit(EXIT_FAILURE);
  }
  unsigned int w = Game.screen.w;
  unsigned int h = Game.screen.h;
  const char * name = Game.screen.name;

  Game.screen.window = SDL_CreateWindow(
    name,
    SDL_WINDOWPOS_CENTERED,
    SDL_WINDOWPOS_CENTERED,
    w, h, 0
  );

  if (Game.screen.window == NULL) {
    printf("Couldn't␣create␣window:␣%s\n", SDL_GetError());
    exit(EXIT_FAILURE);
  }
  Game.screen.renderer = SDL_CreateRenderer(
    Game.screen.window, -1,
    SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC
  );

  if (Game.screen.renderer == NULL) {
    printf("Couldn't create renderer: %s\n", SDL_GetError());
    exit(EXIT_FAILURE);
  }

  setupSpriteSheet();

  Game.running = SDL_TRUE;
}

void game_quit(void) {
  int i;
  for (i = 0; i < Game.gfx.n; i++)
    SDL_FreeSurface(Game.gfx.spritesheet[i]);

  SDL_FreeSurface(Game.gfx.surfaceBack);

  free(Game.gfx.spritesheet);
  Game.gfx.spritesheet = NULL;

  free(Game.tail);

  SDL_DestroyRenderer(Game.screen.renderer);
  Game.screen.renderer = NULL;

  SDL_DestroyWindow(Game.screen.window);
  Game.screen.window = NULL;

  SDL_Quit();
  Game.running = SDL_FALSE;
}
```

```
void displaySnake() {
  Head * current = Game.tail;

  SDL_Texture * texture;

  char * prevChar = NULL;
  while (current != NULL) {
    switch (current -> dir) {
    case 'U':
      if (!prevChar) { //this is the tail
        prevChar = (char * ) malloc(sizeof(char));
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 5));
        if (texture == NULL) {
          printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
          exit(EXIT_FAILURE);
        }
      } else {
        if (!current  -> next) { //this is the head
          texture = SDL_CreateTextureFromSurface(
            Game.screen.renderer, *(Game.gfx.spritesheet + 1));
          if (texture == NULL) {
            printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          }
        } else { //not the head! so perform pattern recognition
          if ( * prevChar == 'L')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 9));
          if (texture == NULL) {
            printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          } else if ( * prevChar == 'R')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 12));
          if (texture == NULL) {
            printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          } else if ( * prevChar == 'U')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 13));
          if (texture == NULL) {
            printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          }
        }

      }
      * prevChar = 'U';
      break;
    case 'D':
      if (!prevChar) { //this is the tail
        prevChar = (char * ) malloc(sizeof(char));
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 7));
        if (texture == NULL) {
          printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
          exit(EXIT_FAILURE);
        }
      } else {

        if (!current -> next) { //this is the head
          texture = SDL_CreateTextureFromSurface(
            Game.screen.renderer, *(Game.gfx.spritesheet + 3));
          if (texture == NULL) {
            printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          }
        } else { //not the head! so perform pattern recognition
          if ( * prevChar == 'L')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 10));
          else if ( * prevChar == 'R')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 11));
          else if ( * prevChar == 'D')
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 13));
    }
  }
  *prevChar = 'D';
  break;
case 'L':
  if (!prevChar) { //this is the tail
    prevChar = (char *) malloc(sizeof(char));
    texture = SDL_CreateTextureFromSurface(
      Game.screen.renderer, *(Game.gfx.spritesheet + 8));
    if (texture == NULL) {
      printf("Error creating texture: %s\n", SDL_GetError());
      exit(EXIT_FAILURE);
    }
  } else {

    if (!current -> next) { //this is the head
      texture = SDL_CreateTextureFromSurface(
        Game.screen.renderer, *(Game.gfx.spritesheet + 4));
      if (texture == NULL) {
        printf("Error creating texture: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      }
    } else { //not the head! so perform pattern recognition
      if ( *prevChar == 'L')
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 14));
      if (texture == NULL) {
        printf("Error creating texture: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      } else if ( *prevChar == 'D')
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 12));
      if (texture == NULL) {
        printf("Error creating texture: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      } else if ( *prevChar == 'U')
```

```
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 11));
      if (texture == NULL) {
        printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      }
    }
  }
  * prevChar = 'L';
  break;
case 'R':
  if (!prevChar) { //this is the tail
    prevChar = (char * ) malloc(sizeof(char));
    texture = SDL_CreateTextureFromSurface(
      Game.screen.renderer, *(Game.gfx.spritesheet + 6));
    if (texture == NULL) {
      printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
      exit(EXIT_FAILURE);
    }
  } else {

    if (!current -> next) { //this is the head
      texture = SDL_CreateTextureFromSurface(
        Game.screen.renderer, *(Game.gfx.spritesheet + 2));
      if (texture == NULL) {
        printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      }
    } else { //not the head! so perform pattern recognition
      if ( * prevChar == 'R')
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 14));
      if (texture == NULL) {
        printf("Error␣creating␣texture:␣%s\n", SDL_GetError());
        exit(EXIT_FAILURE);
      } else if ( * prevChar == 'D')
        texture = SDL_CreateTextureFromSurface(
          Game.screen.renderer, *(Game.gfx.spritesheet + 9));
          if (texture == NULL) {
            printf("Error creating texture: %s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          } else if (*prevChar == 'U')
            texture = SDL_CreateTextureFromSurface(
              Game.screen.renderer, *(Game.gfx.spritesheet + 10));
          if (texture == NULL) {
            printf("Error creating texture: %s\n", SDL_GetError());
            exit(EXIT_FAILURE);
          }
        }
      }
      *prevChar = 'R';
      break;
    default:
      {}
    }

    SDL_RenderCopy(Game.screen.renderer, texture,
    NULL, &current -> headRect);

    current = current -> next;
  }

  SDL_DestroyTexture(texture);

}
```

4.3 The head.h header file

```
#ifndef HEAD_H
#define HEAD_H

#ifdef __cplusplus
extern "C" {
#endif
#include <SDL2/SDL.h>

typedef struct head {
    SDL_Rect headRect;
    char dir;//L = left, R = right, U = up, D = down
    struct head *next;//parent of this node
} Head;


#ifdef __cplusplus
}
#endif

#endif /* HEAD_H */
```

4.4 The snake.c source file

```
#include <SDL2/SDL.h>
#include "snake.h"
#include "head.h"

SDL_bool requestTurn(char dir, Head *tail)
{
        Head *head;
            while(tail != NULL)
            {
                head=tail;
                tail = tail->next;
            }
    if(head->dir == 'U' || head->dir == 'D')
    {
        if(dir == 'L' || dir == 'R')
        {
            head->dir = dir;
            return SDL_TRUE;
        }
    }
    else if(head->dir == 'L' || head->dir == 'R')
    {
        if(dir == 'U' || dir == 'D')
        {
            head->dir = dir;
            return SDL_TRUE;
        }
    }
    return SDL_FALSE;
}

void makeTurn(char dir, Head* tail)
{
        Head *head;
            while(tail != NULL)
            {
                head=tail;
                tail = tail->next;
            }
    switch(dir)
    {
        case 'U':
            head->dir = 'U';
            break;
        case 'R':
            head->dir = 'R';
            break;
        case 'D':
            head->dir = 'D';
            break;
        case 'L':
            head->dir = 'L';
            break;
        default:{}
    }
}

SDL_bool detectCollision(SDL_Rect rectFoodPos, Head *tail)
{
        Head *snakeHead;
        while(tail != NULL)
        {
            snakeHead=tail;
            tail = tail->next;
        }
        SDL_Rect a = rectFoodPos;
        SDL_Rect b = snakeHead->headRect;

        //The sides of the rectangles
        int rightA, rightB;
        int topA, topB;
        //Calculate the sides of rect A
        rightA = a.x + a.w;
        topA = a.y;
        //Calculate the sides of rect B
        rightB = b.x + b.w;
        topB = b.y;

        if((abs(rightA-rightB)<=16) && (abs(topA-topB)<=16))
                return SDL_TRUE;
        else
                return SDL_FALSE;
}
```

4.5 The snake.h header file.

```
#ifndef SNAKE_H
#define SNAKE_H
#include "head.h"

#define STEP 14

void makeTurn(char dir, Head *tail);

//used by both head and body elements
void moveForward(char dir, Head *tail);

//head requests to turn
SDL_bool requestTurn(char dir, Head *tail);

//stop game if snake collides
SDL_bool detectCollision(SDL_Rect rectFoodPos, Head *tail);

void randomiseFood(int min, int max) ;
#endif /* SNAKE_H */
```

4.6 The walls.h header file.

```
#include <stdio.h>
#include <stdlib.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>


SDL_bool checkWallCollision(int level, SDL_Rect *pos);
```

4.7 The walls.c source file.

```
#include <stdio.h>
#include <stdlib.h>
#include <SDL2/SDL.h>
```

```
#include <SDL2/SDL_image.h>
#include "walls.h"

SDL_bool checkWallCollision(int level, SDL_Rect *pos)
{
        switch(level)
        {
                case 1://There are no walls on level 1
                        return SDL_FALSE;
                break;
                case 2://Box
                if(pos->x <= 0 || pos->x >= 624 ||
                pos->y <= 0 || pos->y >= 462)
                {
                        return SDL_TRUE;
                }else
                {
                        return SDL_FALSE;
                }
                break;
                case 3://cross
                if(pos->x == 320 || pos->y == 240)
                {
                        return SDL_TRUE;
                }else
                {
                        return SDL_FALSE;
                }
                break;
                case 4://Pool table

                break;
                case 5://Apartment

                break;
                default:{}
        }
}
```

4.8 The ai.h header file.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include ”walls.h”
#include <stdio.h>

typedef struct cell {
        SDL_bool walkable;
        SDL_Rect worldPosition;
        int g_cost,h_cost;

        struct cell *next;
        struct cell *parent;//used for backtracking
} Cell;

int openSetCount();
void initialiseGrid(int level);
void statistics();
SDL_bool findPath(SDL_Rect start, SDL_Rect dest);
int getDistance(Cell *c1, Cell *c2);
Cell *getNeighbours(SDL_Rect pos);
Cell cellFromPos(SDL_Rect pos);
int fCost(Cell *c);
void addOpenSet(Cell *c);
void removeOpenSet(Cell *c);
void addClosedSet(Cell *c);
int setCount(Cell *c);
SDL_bool contains(Cell *list, Cell *c);
```

4.9 The ai.c source file.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <mysql/mysql.h>
#include ”ai.h”
#include ”walls.h”
#include <stdio.h>
```

```
#include <math.h>

Cell grid[40][30];//grid that stores the state of the level

/*openSet list is implemented as a queue*/
Cell *openSet;//the set of nodes to be evaluated
Cell *closedSet;//the set of nodes already evaluated

void initialiseGrid(int level)
{
        int x, y;
        for(x=0;x<40;x++)
        for(y=0;y<30;y++)
        {
                grid[x][y].worldPosition.x = x*16;
                grid[x][y].worldPosition.y = y*16;
                //length of each cell
                grid[x][y].worldPosition.w = 16;
                grid[x][y].worldPosition.h = 16;

                grid[x][y].next = NULL;
                grid[x][y].parent = NULL;

                if(checkWallCollision(level, &grid[x][y].worldPosition)
                {
                        grid[x][y].walkable = SDL_TRUE;
                }
                else
                {
                        grid[x][y].walkable = SDL_FALSE;
                }
        }
}

//temporary function
int openSetCount(){

        SDL_Rect dummy1,dummy2,dummy3;
        dummy1.x = 600;
        dummy1.y = 205;
        dummy2.x = 60;
        dummy2.y = 15;
        dummy3.x = 400;
        dummy3.y = 24;

        Cell c1 = cellFromPos(dummy1);
        Cell c2 = cellFromPos(dummy2);
        Cell c3 = cellFromPos(dummy3);

        Cell *l1 = &c1;
        c1.next = &c2;
        c2.next = &c3;


        addOpenSet(&c1);
        addOpenSet(&c3);
        addOpenSet(&c2);
        removeOpenSet(&c3);
        return setCount(openSet);
}

/*statistics function prints a summary of the total
number of free cells and the total number of occupied cells*/
void statistics()
{
        int x, y, free = 0, occupied = 0;
        for(x=0;x<40;x++)
        for(y=0;y<30;y++){
                if(grid[x][y].walkable == SDL_TRUE)
                        free++;
                else
                        occupied++;
        }

        printf("free␣cells␣:␣%d\n", free);
        printf("occupied␣cells␣:␣%d\n", occupied);
```

```
}

//A* search algorithm
SDL_bool findPath(SDL_Rect start, SDL_Rect dest)
{
        Cell startCell = cellFromPos(start);
        Cell destCell = cellFromPos(dest);

        //clear the openSet and closedSet
        openSet = NULL;
        closedSet = NULL;

        addOpenSet(&startCell);

        while(setCount(openSet) > 0)
        {
                Cell *current = openSet;//this is the starting node

                int i;
                for(i=1;i<setCount(openSet);i++)
                {
                        if(fCost(openSet+i) < fCost(current) || fCost(o
                        {
                                current = openSet+i;
                        }
                }

                removeOpenSet(current);
                addClosedSet(current);

                //destination has been reached
                if(current->worldPosition.x == destCell.worldPosition.x
                current->worldPosition.y == destCell.worldPosition.y){
                return SDL_TRUE;
                }

                Cell *neighbours = getNeighbours(current->worldPosition
                for(i=0; i<setCount(neighbours); i++)
                {
                        if(grid[(neighbours+i)->worldPosition.x][(neighb
                        {
                                continue;
                        }

                        int newMovementCostToNeighbours = current->g_cos

                        if(newMovementCostToNeighbours < (neighbours+i)-
                        {
                                (neighbours+i)->g_cost = newMovementCos
                                (neighbours+i)->h_cost = getDistance((n
                                (neighbours+i)->parent = current;

                                if(contains(openSet,neighbours+i) == SD
                                {
                                        addOpenSet(neighbours+i);
                                }
                        }
                }
        }
}

//calculates the distance between two cells
int getDistance(Cell *c1, Cell *c2)
{
        int x = abs(c1->worldPosition.x - c2->worldPosition.x);
        int y = abs(c1->worldPosition.y - c2->worldPosition.y);

        return (x+y);
}

Cell *getNeighbours(SDL_Rect pos)
{
        Cell *neighbours;//list of neighbour cells
        Cell u, d, l, r;
        if(pos.y-16>=0){
          u.worldPosition.x = pos.x;
          u.worldPosition.y = pos.y+16;
          neighbours = &u;
          if(pos.y+16<=480){
                    d.worldPosition.x = pos.x;
                    d.worldPosition.y = pos.y-16;
                    u.next = &d;
          }
          if(pos.x-16>=0){
                    l.worldPosition.x = pos.x-16;
                    l.worldPosition.y = pos.y;
                    d.next = &l;
          }
          if(pos.x+16<=640){
                    r.worldPosition.x = pos.x+16;
                    r.worldPosition.y = pos.y;
                    l.next = &r;
          }
          }
else if(pos.y+16<=480){
                    d.worldPosition.x = pos.x;
                    d.worldPosition.y = pos.y-16;
                    neighbours = &d;
          if(pos.x-16>=0){
                    l.worldPosition.x = pos.x-16;
                    l.worldPosition.y = pos.y;
                    d.next = &l;
          }
          if(pos.x+16<=640){
                    r.worldPosition.x = pos.x+16;
                    r.worldPosition.y = pos.y;
                    l.next = &r;
          }
          if(pos.y-16>=0){
                    u.worldPosition.x = pos.x;
                    u.worldPosition.y = pos.y+16;
                    r.next = &u;
          }
```

```
}
else if(pos.x-16>=0){
                l.worldPosition.x = pos.x-16;
                l.worldPosition.y = pos.y;
                neighbours = &l;
                if(pos.x+16<=640){
                        r.worldPosition.x = pos.x+16;
                        r.worldPosition.y = pos.y;
                        l.next = &r;
                }
                if(pos.y-16>=0){
                        u.worldPosition.x = pos.x;
                        u.worldPosition.y = pos.y+16;
                        r.next = &u;
                }
                if(pos.y+16>=0){
                        d.worldPosition.x = pos.x;
                        d.worldPosition.y = pos.y-16;
                        u.next = &d;
                }
}
else if(pos.x+16<=640){
        r.worldPosition.x = pos.x+16;
        r.worldPosition.y = pos.y;
        neighbours = &r;
                if(pos.y-16>=0){
                u.worldPosition.x = pos.x;
                u.worldPosition.y = pos.y+16;
                r.next = &u;
                }
                if(pos.x-16>=0){
                        l.worldPosition.x = pos.x-16;
                        l.worldPosition.y = pos.y;
                        u.next = &l;
                }
                if(pos.y+16<=480){
                        d.worldPosition.x = pos.x;
                        d.worldPosition.y = pos.y-16;
```

```
                                l.next = &d;
                        }
        }
        return neighbours;
}

//creates a Cell given a SDL_Rect
Cell cellFromPos(SDL_Rect pos)
{
        Cell myCell;
        myCell.worldPosition = pos;
        myCell.walkable = SDL_TRUE;
        myCell.next = NULL;
        myCell.parent = NULL;
        return myCell;
}

int fCost(Cell *c)
{
return c->g_cost + c->h_cost;
}

//add an element to open set
void addOpenSet(Cell *c)
{
        if(openSet == NULL)//list is empty
                openSet = c;
        else{//enqueue new cell onto the list
                Cell *temp = openSet ;
                openSet = c;
                c->next = temp;
        }

}
//remove element from the open list
void removeOpenSet(Cell *c)
{
        if(openSet == NULL || c == NULL)
                return;

        Cell *temp = openSet;
        int i;
        Cell* prev;
        printf("OPEN␣has␣%d␣cells",setCount(temp));

        for(i=0;i<setCount(temp);i++)
        {
                if(temp->worldPosition.x == c->worldPosition.x &&
                temp->worldPosition.y == c->worldPosition.y)
                {
                        //remove this cell
                        if(prev == NULL)
                                openSet = openSet->next;
                        else{
                                prev->next = temp->next;
                                temp->next = NULL;
                        }
                        break;
                }
                prev = temp;
                temp = temp->next;
        }

}

//add cell to closed set
void addClosedSet(Cell *c)
{
        if(closedSet == NULL)//list is empty
                closedSet = c;
        else{//enqueue new cell onto the list
                Cell *temp = closedSet ;
                closedSet = c;
                c->next = temp;
        }
```

```
}

int setCount(Cell *c)
{
        Cell* temp = c;
        int count = 0;
        while(temp != NULL)
        {
                count++;
                temp=temp->next;
        }
        return count;
}

//returns true if list contains the cell c
SDL_bool contains(Cell *list, Cell *c)
{
        if(list == NULL || c == NULL)
                return SDL_FALSE;

        Cell *temp = list;
        while(temp != NULL)
        {
                if(temp->worldPosition.x == c->worldPosition.x &&
                temp->worldPosition.y == c->worldPosition.y)
                {
                        return SDL_TRUE;
                }
                temp=temp->next;
        }
        return SDL_FALSE;
}
```

4.10 The user interface header file.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
```

```
SDL_Surface* displayGUI(const char *text);
```

4.11 The user interface source file.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include "gui.h"

TTF_Font * font;

SDL_Surface* displayGUI(const char *text)
{
        TTF_Init();
        if (!font)
                font = TTF_OpenFont("/home/steamrolle/Documents/SOURCE_C
        SDL_Color color={255,255,255};
        SDL_Surface * scoreText = TTF_RenderText_Solid(font,text, color
        return scoreText;
}
```

4.12 The audio player header file.

4.13 The audio player source file.

4.14 The score board manager header file.

```
#include <SDL2/SDL.h>
#include <mysql/mysql.h>

SDL_bool updateBoard();
SDL_bool checkScore(int score);
int insertionSort(int score);
SDL_bool uploadScore(int position, int score, char *name);
```

4.15 The score board manager source file.

```
#include <SDL2/SDL.h>
#include <mysql/mysql.h>
#include "scoreBoard.h"


SDL_bool updateBoard()
{
    MYSQL *conn;

    conn = mysql_init(NULL);

    if (conn == NULL) {
        printf("Error␣%u␣%s\n", mysql_errno(conn), mysql_error(conn));
        exit(SDL_FALSE);
    }

    if (mysql_real_connect(conn, "localhost", "root", "root", NULL, 8889
        printf("Error␣%u:␣%s\n", mysql_errno(conn), mysql_error(conn));
        exit(SDL_FALSE);
    }


    if (mysql_query(conn, "create␣database␣testdb")) {
        printf("Error␣%u:␣%s",mysql_errno(conn), mysql_error(conn));
        exit(SDL_FALSE);
    }

    mysql_close(conn);

    return SDL_TRUE;
}

/*
checkScore returns SDL_FALSE is score isn't good enough for top 10,
else returns SDL_TRUE
*/
```

```
SDL_bool checkScore(int score)
{
        char *query = "SELECT␣*␣FROM␣score_board";

}

//returns the position that the passed score must take
int insertionSort(int score)
{

}

//uploads the passed score into the position specified
SDL_bool uploadScore(int position, int score, char *name)
{

}
```

5 Testing

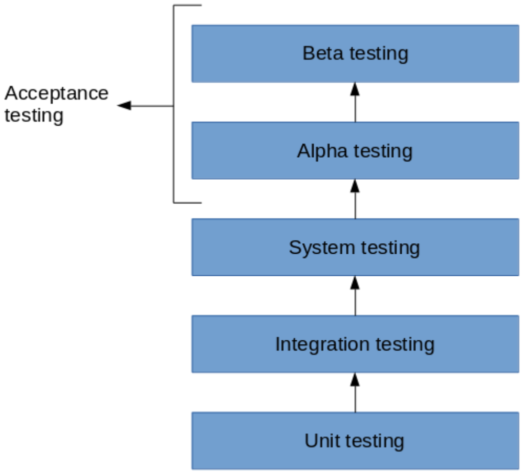
5.1 Importance of testing.

Testing is very vital because it allows the developer to discover bugs in the code before in the early stages before releasing it. This help minimise unintended behavior as much as possible and improve the quality of the final product. Testing helps developers discover and reduce unnecessary coupling between units and also improve the coherence of each. Testing is an investigative process that proves that the developed system satisfies the following:

1. It meets all the requirements.
2. It responds predictably to all kinds of possible input.
3. It performs all tasks within an acceptable amount of time.
4. Useful software is realised in shorter span of time as the developers are more focused on meeting the test requirements from the start.

There are various types of testing that can be conducted on a software to verify and validate its functionality at different levels of abstraction.

Different types of testing



5.2 Unit testing

Unit testing is where individual units or components of tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of a software. It usually accepts a single or multiple inputs and produces a single output. In procedural programming, the smallest unit is a function. Unit testing frameworks drivers, stubs and mock objects are tools used to assist in unit testing.

Benefits of unit testing

Unit testing allows developers to follow the incremental code and test approach which has three main benefits:

> 1.) Because the unit test uses the interface to the unit being tested it allows the developer to think about how the interface should be designed for usage early in the coding process.
>
> 2.) They help the developer avoid spaghetti code and aberrant cases. This increases the chances of producing code that conforms to standard practices.
>
> 3.) By providing a documented level of correctness, they allow the developer to refactor aggressively.

The check unit testing framework

I made use of the Check unit testing framework which was designed for the C programming language. The check framework is a simple framework which features a simple interface for defining unit tests. Tests are run in a separate address space, so both assertion failures and code errors that cause segmentation faults or other signals can be caught.

5.2.1 The game manager function.

Unit testing the main module.

5.2.2 The snake function.

```
#include <check.h>
#include <stdlib.h>
#include <stdio.h>
#include ”head.h”

void makeTurn(char dir, Head *tail);

//used by both head and body elements
void moveForward(char dir, Head *tail);

//head requests to turn
SDL_bool requestTurn(char dir, Head *tail);

//stop game if snake collides
SDL_bool detectCollision(SDL_Rect rectFoodPos, Head *tail);

void randomiseFood(int min, int max) ;

START_TEST(test_collision)
{
        Head tail;
        tail.headRect.x=0;
        tail.headRect.y=0;
        tail.dir = ’U’;
        tail.next = NULL;

        SDL_Rect pos;

        pos.x = 0;
        pos.y = 0;

        ck_assert_msg (detectCollision(pos, &tail) == SDL_TRUE, ”Collisi

}
END_TEST
```

```
START_TEST(test_request_turn)
{
/*
This tests the request turn function which governs that the direction w
*/
        Head head;
        head.headRect.x=0;
        head.headRect.y=0;
        head.next = NULL;

        head.dir = 'U'; //facing up
        //requesting to turn left
        ck_assert_msg (requestTurn('L', &head) == SDL_TRUE,
"Facing␣up,␣and␣failed␣to␣turn␣left");
        //requesting to turn right
        ck_assert_msg (requestTurn('R', &head) == SDL_TRUE,
"Facing␣up,␣and␣failed␣to␣turn␣right");
        //requesting to turn up
        ck_assert_msg (requestTurn('U', &head) == SDL_TRUE,
"Facing␣up,␣and␣failed␣to␣turn␣up");
        //requesting to turn down
        ck_assert_msg (requestTurn('D', &head) == SDL_TRUE,
"Facing␣up,␣and␣failed␣to␣turn␣down");

        head.dir = 'R'; //facing right
        //requesting to turn up
        ck_assert_msg (requestTurn('U', &head) == SDL_TRUE,
"Facing␣right,␣and␣failed␣to␣turn␣up");
        //requesting to turn right
        ck_assert_msg (requestTurn('R', &head) == SDL_TRUE,
"Facing␣right,␣and␣failed␣to␣turn␣right");
        //requesting to turn down
        ck_assert_msg (requestTurn('D', &head) == SDL_TRUE,
"Facing␣right,␣and␣failed␣to␣turn␣down");
        //requesting to turn left
        ck_assert_msg (requestTurn('L', &head) == SDL_TRUE,
"Facing␣right,␣and␣failed␣to␣turn␣left");
```

```
        head.dir = 'D';//facing down
        //requesting to to up
        ck_assert_msg (requestTurn('U', &head) == SDL_TRUE,
"Facing␣down,␣and␣failed␣to␣turn␣up");
        //requesting to turn right
        ck_assert_msg (requestTurn('R', &head) == SDL_TRUE,
"Facing␣down,␣and␣failed␣to␣turn␣right");
        //requesting to turn down
        ck_assert_msg (requestTurn('D', &head) == SDL_TRUE,
"Facing␣down,␣and␣failed␣to␣turn␣down");
        //requesting to turn left
        ck_assert_msg (requestTurn('L', &head) == SDL_TRUE,
"Facing␣down,␣and␣failed␣to␣turn␣left");

        head.dir ='L';//facing left
        //requesting to to up
        ck_assert_msg (requestTurn('U', &head) == SDL_TRUE,
"Facing␣left,␣and␣failed␣to␣turn␣up");
        //requesting to turn right
        ck_assert_msg (requestTurn('R', &head) == SDL_TRUE,
"Facing␣left,␣and␣failed␣to␣turn␣right");
        //requesting to turn down
        ck_assert_msg (requestTurn('D', &head) == SDL_TRUE,
"Facing␣left,␣and␣failed␣to␣turn␣down");
        //requesting to turn left
        ck_assert_msg (requestTurn('L', &head) == SDL_TRUE,
"Facing␣left,␣and␣failed␣to␣turn␣left");
}
END_TEST


Suite *test_snake_suite(void)
{
        Suite *suite;
        TCase *core;

        suite = suite_create("Snake");
        core = tcase_create(”Core”);

        tcase_add_test(core, test_collision);
        tcase_add_test(core, test_request_turn);

        suite_add_tcase(suite, core);

        return suite;
}

int main(void)
{
        int failed;
        Suite *suite;
        SRunner *runner;

        suite = test_snake_suite();
        runner = srunner_create(suite);

        srunner_run_all(runner, CK_VERBOSE);
        failed = srunner_ntests_failed(runner);
        srunner_free(runner);

        return (failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

5.2.3 The wall generation function.

```
#include <check.h>
#include <stdlib.h>
#include <stdio.h>
#include ”walls.h”


//function prototypes to be tested


START_TEST(test_walls)
```

```
{
        int level; //level being tested
        SDL_Rect pos; //position being evalueted

        level = 1;
        //Checking positions without walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_FALSE,
"Warning␣collision␣occured");

        //Checking positions with walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_TRUE,
"Warning␣collision␣did␣not␣occured");

        level = 2;
        //Checking positions without walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_FALSE,
"Warning␣collision␣occured");

        //Checking positions with walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_TRUE,
"Warning␣collision␣did␣not␣occured");

        level = 3;
        //Checking positions without walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_FALSE,
"Warning␣collision␣occured");

        //Checking positions with walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_TRUE,
"Warning␣collision␣did␣not␣occured");

        level = 4;
        //Checking positions without walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_FALSE,
"Warning␣collision␣occured");

        //Checking positions with walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_TRUE,
"Warning␣collision␣did␣not␣occured");

        level = 5;
        //Checking positions without walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_FALSE,
"Warning␣collision␣occured");

        //Checking positions with walls
        ck_assert_msg (checkWallCollision(level, pos) == SDL_TRUE,
"Warning␣collision␣did␣not␣occured");
}
END_TEST

Suite * test_walls_suite()
{
        Suite *suite;
        TCase *core;

        suite = suite_create("Snake");

        core = tcase_create("Core");

        tcase_add_test(core, test_walls);

        suite_add_tcase(suite, core);

        return suite;
}


int main(void)
{
        int failed;
        Suite *suite;
        SRunner *runner;

        suite = test_walls_suite();
        runner = srunner_create(suite);
        srunner_run_all(runner, CK_VERBOSE);
        failed = srunner_ntests_failed(runner);
        srunner_free(runner);

        return (failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

5.2.4 The food position generation function.

```
#include <check.h>
#include <stdlib.h>
#include <stdio.h>
#include ”walls.h”


//function prototypes to be tested


START_TEST(test_collision)
{

}
END_TEST

Suite* test_food_suite()
{

}

int main(void)
{
        int failed;
        Suite *suite;
        SRunner *runner;

        suite = test_food_suite();
        runner = srunner_create(suite);
        srunner_run_all(runner, CK_VERBOSE);
        failed = srunner_ntests_failed(runner);
        srunner_free(runner);

        return (failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

5.2.5 The high score update function.

```
#include <check.h>
#include <stdlib.h>
#include <stdio.h>
#include "walls.h"


//function prototypes to be tested


START_TEST(test_collision)
{

}
END_TEST

Suite* test_sb_suite()
{

}

int main(void)
{
        int failed;
        Suite *suite;
        SRunner *runner;

        suite = test_sb_suite();
        runner = srunner_create(suite);
        srunner_run_all(runner, CK_VERBOSE);
        failed = srunner_ntests_failed(runner);
        srunner_free(runner);

        return (failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

5.2.6 Errors discovered

Various bugs were rooted out during this testing process, namely :
Snake wrapping around issue. Snake was going of screen because the wrap around function parameters where not properly set.

5.3 Integration testing

Integration testing is the second level of testing performed after unit testing. At this point individual units are combined and tested as a group. The purpose of this of this level of testing is to expose defects faults in the interaction between integrated units. Test drivers and test stubs are used to assist in integration testing. This type of testing exposes defects in the interfaces and interaction between integrated components. At this point I put all the modules together and make sure that they behave as expected.
There are various approaches to performing integration testing, namely: big bang, top down, bottom up and hybrid.

5.3.1 The big bang approach

In this approach to integration testing all or most of the units are combined together and tested in one go. Integration testing only test the interactions between the various units and does not test the entire system, which is perform at the system testing phase.

5.3.2 The top down approach

In this types of testing the highest level units are tested first and following level follows in-turn. This continues recursively until to the lowest level units are reached. This approach it taken when a top down development paradigm is followed. Test stubs are are used to simulate lower level units if necessary because they may not be available during the initial phases.

5.3.3 The bottom up approach

This approach to integration testing is followed when the bottom up development methodology is used when designing the system. The lowest level units are tested first and upper level units inline after that until the highest level units are reached. Test drivers are used to simulate higher level units which may not be available during the initial phases.

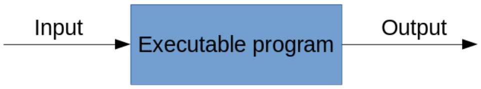
5.3.4 The sandwich or hybrid approach

This is a combination of the top down and bottom up approaches.

5.4 The system testing phase

This is the third level of system testing performed after integration testing and before the first alpha release. The entire system is tested at this phase. The purpose of this test is to evaluate the system's compliance with the specified requirements.In order to pass this test the system must meet all the requirements. The black box method is usually used when performing system testing. A detailed explanation of the black boxing method is given below.

5.5 The black boxing method



This is also know as behavioral testing. In this method the tester need not have any knowledge of the following inner working of the system or unit in question:

1. Internal structure
2. Design
3. Implementation

In this method the internal workings of the system are not visible to the user. The user is only able to interact with the system.

Black boxing attempts to find and correct the following

1. Incorrect or missing functions.
2. Interface errors.
3. Errors in data structures or external database access.
4. Behavior or performance errors.

5. Initialisation or termination errors.

Black boxing is applicable to the integration and system testing levels. The higher the level the more complex and bigger the box.

There are various methods used to design black box tests. The following is a brief description of each.

1. **Equivalence partitions:**
   This is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

2. **Boundary value analysis**
   In this test design technique the determination of boundaries for input values and the selection of values that are at the boundaries and just inside or outside of the boundaries as test data

3. **Cause effect graphing:**
   It is a software test design technique that involves identifying the causes (input conditions) and effects (output conditions), producing a cause effect graph , and generating test cases accordingly.

Advantages of black boxing

1. Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.

2. Testers need not know know any programming languages or how the software has been implemented.

3. Tests can be conducted by a body independent from the developers , allowing for an objective perspective and the avoidance of developer bias.

4. Test cases can be designed as soon as the specifications are completed.

Disadvantages of black boxing

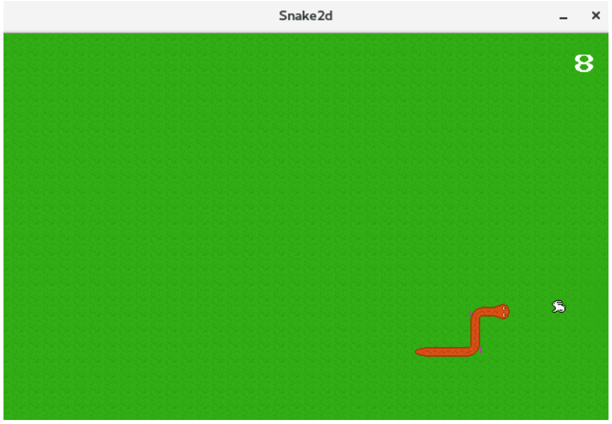
1. Only a small number of possible inputs can be tested and many program paths will be left untested.

2. Without clear specifications test cases are difficult to design.

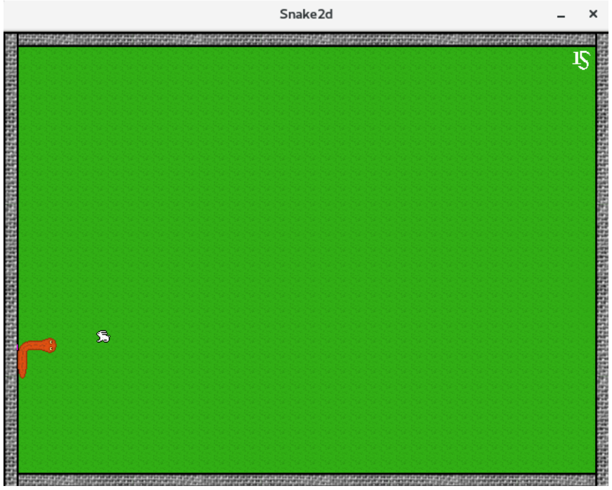
3. Tests can be redundant if the software designer or developer has already run a test case.

5.6 Screen shots

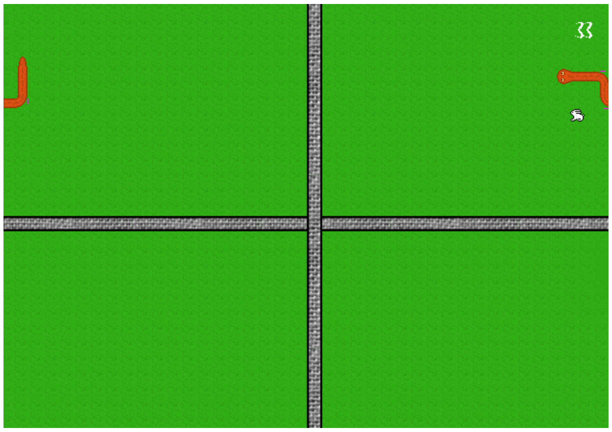
5.6.1 The first level



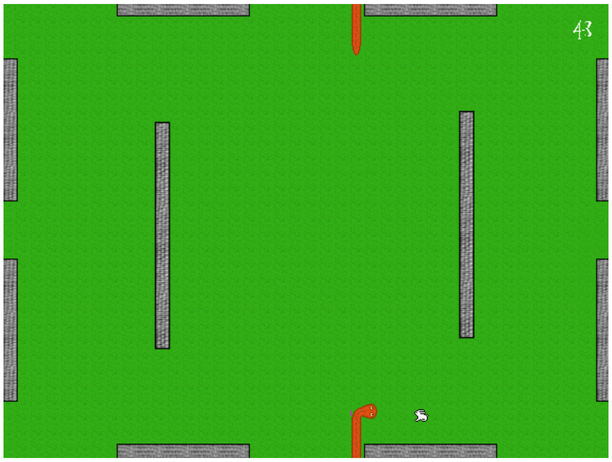
5.6.2 The second level



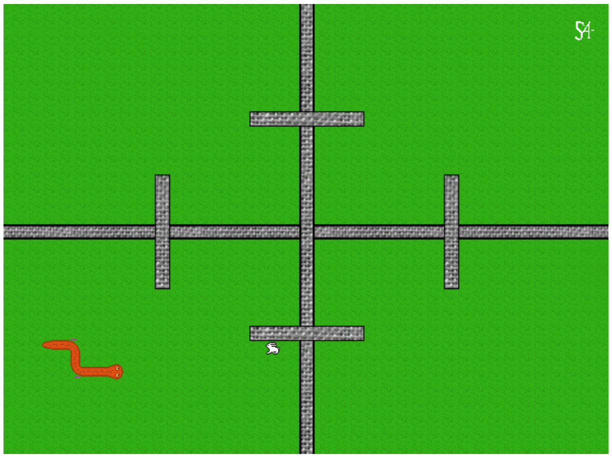
5.6.3 The third level



5.6.4 The fourth level



5.6.5 The fifth level



5.7 The acceptance test

The alpha release and the beta are known together as the acceptance test.

5.7.1 The alpha release

Alpha testing is a type of acceptance testing performed to unveil hidden bugs that where missed by prior tests. This is done before releasing the product to the public. The main focus of the test is to simulate the end users by using black-box or white-box techniques. The aim is for the alpha testers to carry out the tasks that a typical end user might perform and discover any unhandled bugs. This test is usually performed in a closed environment such as a laboratory by test engineers, employees, friends and family.

Advantages of alpha testing

1. Provides better view about the reliability of the software at an early stage.

2. Helps simulate real time user behavior and environment.

3. Detect many serious errors and bugs.

4. Ability to provide early detection of errors with respect to design and functionality.

Disadvantages of alpha testing

1. In depth functionality cannot be tested as software is still under development stage. Sometimes the developers and testers are dissatisfied with the results of alpha testing.

5.7.2 The beta release

Best testing of a product is performed by the end users of the software application in a real environment and can be considered as a form of external user acceptance. The main purpose of this test is to obtain feedback on the product quality. The beta version of the product is only released to a limited number of users.

Advantages of beta testing

1. Reduces product failure risk via customer validation.

2. Beta testing allows a company to test post-launch infrastructure.

3. Improves product quality via customer feedback.

4. Much more most effective compared to similar data gathering methods.

5. Create goodwill with customers and increases customer satisfaction.

Disadvantages of beta testing

1. Since beta test occurs outside the organisation it has unique issues that are not faced in a internal test performed within the organisation in controlled environment. It is performed out in the real world were there is a limited amount of control.

2. Finding the right beta users and maintaining their participation could be a challenge.

5.8 The final release

All the bugs discovered in the various testing stages where handled accordingly. After considering and applying some of the improvements suggested by alpha and beta testers the project has been a success. All the requirements (software specification) have been met.

6 Future improvements

The amount of parallel auxiliary memory space used during run time can be reduced by optimising the use of memory. For example instead of loading multiple SDL_Textures into memory, one meta sprite could be used to represent all the images. Using a single meta sprite also improves the space complexity as it eliminates the overhead encountered when switching textures in the graphics hardware. The game can be improved and therefore making it more exciting and challenging to players by adding more levels and more engaging appealing graphics such as an engaging typeface for example "JuiceBold" and cartoon art assets.

The game can also be improved by including a multi-player mode over a network (online). This would allow multiple players in different locations to compete directly on

Implementing different approaches to the non-player-character (NPC) might yield interesting results for the players, for example the use of neural networks that use supervised learning, a convolutional neural network and other searching algorithms such as linear search, binary search and interpolation search.

7 Conclusion

The software eventually passed all the tests it was subjected to. All the bugs uncovered during the various testing phases were resolved. Although this software has room for more improvement, this project was a complete success.

8 References

1. Normand E, Programming Paradigms and the Procedural Paradox, accessed 14 October 2018,
<https://lispcast.com/procedural-paradox/>

2. KaeTheDev, The Importance of Prototyping , accessed 05 October 2018,
<https://gamejolt.com/@KaeTheDev/post/the-game-creation-process-part-3-the-importance-of-prototyping-t-jx8echxy>

3. Geeks for geeks, Linked lists, accessed 28 October 2018,
<https://www.geeksforgeeks.org/data-structures/linked-list/>

4. Geeks for geeks, Insertion sort, accessed 03 November 2018,
<https://www.geeksforgeeks.org/insertion-sort/>

5. Millington I, Funge J 2009, *Artificial intelligence for games*, edn 2, Morgan Kaufmann publishers, USA.

6. Pazera E, 2003, *Focus on SDL*, Premier press, Cincinnati, Ohio, USA

7. Teixeira de Sousa B M, 2002, *Game programming all in one*, Premier press, USA

8. Sobell M G, 2010, *A practical guide to linux commands, editors, and shell programming*, edn 2, Prentice hall, USA

9. Tremblay C, 2004, *Mathematics for game developers*, Premier press, USA

10. Software Testing, Unit Testing, accessed 06 November 2018,
<http://softwaretestingfundamentals.com/unit-testing/>