Program documentation - Snake Game - ENG

Zip content

There are a total of 3 header files included with their own .c type files which are all included in the main file. These header files are: snake.h, Gridys.h and ui.h. These are all needed for the program to work. There is an Assets folder with all the needed textures, map layouts and also a font type. The program is working with SDL, so the included frameworks are needed for the program to run.

main.c

In this file I mainly initialize the textures and boot up SDL. I created a structure for the textures and for a font type, so it is easier to use them in functions global variables: (typedef struct contextObject) and for SDL : SDL_Window* window, SDL_Renderer* renderer

```
int main(int argc, char* argv[])
```

This is my main function, with these inputs for SDL to work properly. In the first part I create variables, the width and height are for the size of the SDL window, the rows and columns are for the Grid that divides the window into 16x16 equal parts. After that I have some variables for the Frames per Second limiting, because the program uses (SDL_PollEvent), which is very much CPU based, so it might run on 100% CPU usage. This way, with the FPS limiter we can reduce that 100% usage to about 30-50%. Then I have called the sdl_init and loadMedia functions, which initiate SDL and load in the textures and the font type. After that I have created the Grid with the Area function, this divides the whole screen into 16X16 square tiles.

The base state of the game is the main menu, so when the program starts, when user will be able to choose between 2 maps. Then comes the main game loop, here is the part where the program changes between different game states and the main actions happen. The first state is the mainMenu - (inMainMenu function). When a map is chosen by the user, this state changes to loading - (loadmap function), in this case the chosen map and the different kind of objects, for example the snake's head, the place of the apple are generated. This only takes a second, so the state then changes to inGame, where the user can see the generated structures and can play the game. This state works with SDL_PollEvent and the snake can be moved by the arrow keys and with WASD - for this the handleKeyboard function is called. The movement works with a delay variable, and with the SnakeMovement function. When the user collides with an apple, the Newcords function is called which gives a new random coordinate for the apple and with the NewSegment function, the snake gets longer. With sprintf we write the current score on the screen which is based on how many apples the user have eaten and with the textCreator function we create that text box on the screen. Then I started rendering the textures on the screen, for the background I use two for loops that go through every single one of the tiles of the grid and renders a simple grass texture or an obstacle depending on the type of location. Then I rendered the other textures with SDL_RenderCopy and SDL_RenderCopyEx and used my DrawSnakeSegment function that rendered the body parts of the snake one-by-one. After tha inGame state comes the dead - state - (deadScreen function). There is a quit state as well. By the end of the main function it is really important to free the allocated memory for the Grid - (freeArea - function), (so that there is no memory junk left) and also to close SDL - (quitSDL - function).

```
bool sdl_init(int width, int height);
```

This is the function that boots up SDL and there are some Error handling if the SDL cannot be initialized, so it writes the error codes int the console.

```
bool loadMedia(SDL_Renderer* renderer, contextObject* assets)
```

With this function, I imported the textures and the font so that they can be used in the program, and also if the textures or the font cannot be loaded it writes an error message to the console.

```
void quitSDL(contextObject* assets)
```

This function is really important, because when the program ended the textures must be destroyed, the font file must be closed, and the (SDL) window and renderer must be destroyed as well, and the SDL and TTF must quit (TTF_Quit( ), SDL_Quit( )).

snake.c / snake.h

In these files I mainly have set up the basic movement of the snake and some other functions that are in connection with the apple structure.

Firstly I have created 3 different structures them being: typedef struct Movement - in this I placed the directions, typedef struct SnakeSegments - in this there are the parts of the snake, their angles, coordinates, and the previous and next parts of the snake in a linked list, typedef struct Snake - in this is the main parts for the moving of the snake, it works with a delay mechanism, that slowly shifts the snake from one place to another, without just teleporting really fast, typedef struct Apple - this is just the coordinates of the apple.

We have 6 different functions in this file:

```
void SnakeMovement(Movement* mov, Snake* s, int rows, int columns,
Tile** Grid, state* currentState);
void DrawSnakeSegment(SDL_Renderer* renderer, SDL_Texture* tex, Snake*
segment, Tile** Grid);
void NewSegment(Snake* s);
void NewCords(Apple* a, Tile** Grid, int rows, int columns);
void handleKeyboard(int key, Movement* mov, bool* quitGame);
void freeSnake(Snake* s);
```

```
void SnakeMovement(Movement* mov, Snake* s, int rows, int columns,
Tile** Grid, state* currentState);
```

This is one of the most important functions In the whole program. The snake's whole movement is based on this function. The movement depends on previous directions as well so the snake cannot go into its self, if the snake is going in one way, it cannot go to the opposite direction because it would collide with its own head. So I had to test on every possible movement. For that I have created variables that save the current and previous directions. The movement itself is easy, we add 1 to the corresponding coordinate and use delay (in main.c - row: 118) to slowly “animate” the snake. I have also included an angle so that the texture is rotated based on the current direction of the snake. Then I added a collision detection, so if the snake’s head coordinates match with the parts of the snake or an obstacle, it is game over and it goes to the “dead” state. It is really important to free the elements of the snake linked list after it is game over so there are no remaining memory junk in the system - (freeSnake function). Then I have a few cases wether the snake is built up from only a head, or has more body parts so we know if there is collision in that coordinate. Also the body parts angle are inherited by the snake’s head angle, so they are rotated in the proper direction.

```
void DrawSnakeSegment(SDL_Renderer* renderer, SDL_Texture* tex, Snake*
segment, Tile** Grid);
```

With this function, we can render the body parts of the snake one-by-one when the snake’s head collides with an apple.

```
void NewSegment(Snake* s);
```

With this function we add new body parts to the snake (works as a linked list) when the head collides with an apple. We have do dynamically reserve space for each and one of the snake parts when the length of the snake rises.

```
void NewCords(Apple* a, Tile** Grid, int rows, int columns);
```

This function is called when we initialize the starting position of the apple on the map and also when the snake’s head collides with an apple, because after that the apple has to be placed to a new position randomly. Although it is not that simple because the apple must not spawn on an obstacle nor under the snake’s body.

```
void handleKeyboard(int key, Movement* mov, bool* quitGame);
```

This is the function that handles keyboard input, this is what sets the different directions to true or false, depending on which key the user pushed. You can use the arrow keys and also WASD to control the snake.

```
void freeSnake(Snake* s);
```

This function is used to free up the allocated memory for the snake’s body parts, so If the user has finished playing with one of the maps and then he wants to try out the other map by going back to the main menu, he wont run into bugs because of the memory junk that was left there.

GridSys.c / GridSys.h

In these files I basically divided the whole SDL window into 16x16 individual parts and also I have written a function that reads in the 2 different types of map from .txt files and then it loads in the map depending on which one the user chose.

We have 3 different functions in this file:

```
Tile** Area(SDL_Rect sqr, int rows, int columns, Tile** Grid);
void loadMap(Tile** Grid, int rows, int columns, int mapNumber);
void freeArea(Tile** Grid);
```

```
Tile** Area(SDL_Rect sqr, int rows, int columns, Tile** Grid);
```

In this function I set up a layout for the whole map where the snake can move horizontally and vertically. In this program there are 16 columns and 16 rows. For this I first needed to reserve some space for the whole Area. So basically it is a 2 dimensional array wich is 16x16 tiles big. I set up the width and height of every single little square and their x and y coordinates. And also this function has a return value Grid.

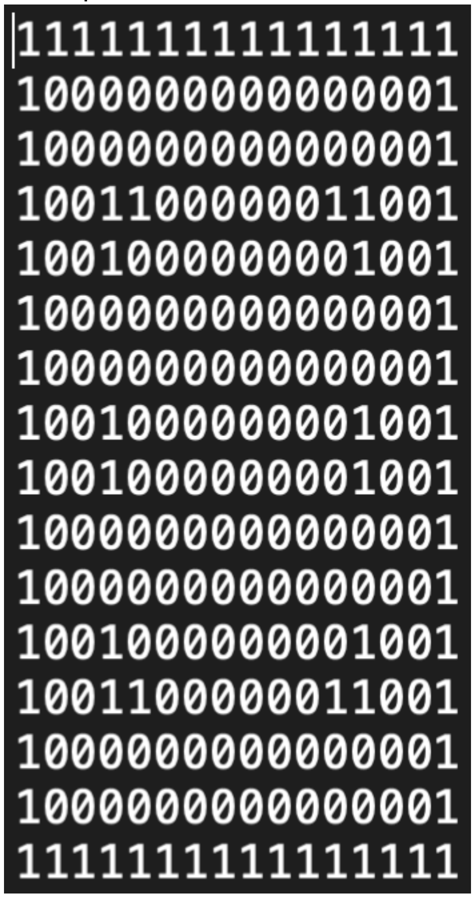
```
void freeArea(Tile** Grid);
```

This function just frees up the allocated memory for the Grid so there are no leftover memory junk. This is used in main (row: 168.), basically at the end of the program.

```
void loadMap(Tile** Grid, int rows, int columns, int mapNumber);
```

This is the function that reads in the maps from a .txt file and decides where the snake can move freely or collide with an obstacle.

A map looks like this:



```
1111111111111111
1000000000000001
1000000000000001
1001100000011001
1001000000001001
1000000000000001
1000000000000001
1001000000001001
1001000000001001
1000000000000001
1000000000000001
1001000000001001
1001100000011001
1000000000000001
1000000000000001
1111111111111111
```

There are 16 rows and 16 columns. The 1's are the obstacles, where the snake cannot go, and will end the game if they accidentally collide with and the 0's are the places where the snake can move safely. With two for loops we can go through every single element of the numbers and we can put them inside the Grid.

ui.c / ui.h

In these files I created the states for the program, so there are different types of moments in this program. Also there are functions for creating accessible buttons throughout the window that can be used by the left mouse button.

First I initialized different structures:
typedef struct textStruct - this struct has a shape, a limit of characters that can be written in it, a scale that can be used to make it bigger or smaller, and a color.
typedef struct button - this struct has a position (in the Grid), a text from the above variable, it has a bool state wether it its clicked or not, and also a color.
typedef enum state - there are 5 states in this program: mainMenu, loading, inGame, quit, dead

There are 5 functions in this file:

```
void inMainMenu(state* currentState, int* mapType, Tile** Grid, int
columns, int rows, SDL_Renderer* renderer, TTF_Font* font);
static void buttonAction(button* btn, SDL_Event* ev);
static button buttonCreator(SDL_Rect gridPosition, int width, int
height, char* text, int scaleSize);
void textRenderer(SDL_Renderer* renderer, TTF_Font* font, textStruct*
word);
void deadScreen(Tile** Grid, int columns, int rows, state*
currentState, int score, SDL_Renderer* renderer, TTF_Font* font);
textStruct textCreator(SDL_Rect box, char self[16], int scale,
SDL_Color color);
```

```
void inMainMenu(state* currentState, int* mapType, Tile** Grid, int
columns, int rows, SDL_Renderer* renderer, TTF_Font* font);
```

This is the function that is responsible for the game to start in the Main Menu, where the user has 3 options, to start playing on either one of the maps or to quit the program. Here I basically created 3 buttons (with the buttonCreator function) and placed them respectively inside the grid. Then created an event (with SDL_WaitEvent) and started rendering the texts (with textRenderer) on the screen. Then I had to make these buttons work so that when they are pressed, they do something. By using the function buttonAction I added a meaning to these buttons and they were usable. There is a quit button that ends the program (goes to state quit), a button for the classic snake map (it has a MapType = 0)(goes to state loading -> inGame, see at main.c row: 94) and also a button for the harder map (it has a mapType = 1)(goes to state loading -> inGame, see at main.c row: 94).

```
static void buttonAction(button* btn, SDL_Event* ev);
```

This function gives meanings to the buttons that are created. For this I needed a little help so I use this link to get these buttons to work properly: https://stackoverflow.com/questions/39133873/how-to-set-a-gui-button-in-the-win32-window-using-sdl-c . If the coordinates of the mouse match the created text's location in the grid and it is clicked only then can the program go to the next state. Else as it is only hovering on the button it changes colors.

```
static button buttonCreator(SDL_Rect gridPosition, int width, int
height, char* text, int scaleSize);
```

This function creates a button in the Grid that is clickable and can get you to the next state in the program in this case choose a map or quit the program. It gets a position inside the grid, its width and height, a text that has to be rendered to the screen and a scalar multiplier. In this function the textCreator function is called which is explained below.

```
textStruct textCreator(SDL_Rect box, char self[16], int scale,
SDL_Color color);
```

This function creates a text, it uses strcpy to copy the string that the function got to the textStruct tempText which is then returned in the end. It is really easy to use and to create little texts with it.

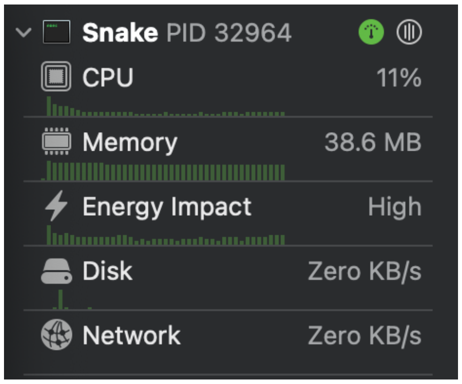
```
void textRenderer(SDL_Renderer* renderer, TTF_Font* font, textStruct*
word);
```

This function puts the texture that we want to see on the screen using TTF and the given font from the Assets folder.

```
void deadScreen(Tile** Grid, int columns, int rows, state*
currentState, int score, SDL_Renderer* renderer, TTF_Font* font);
```

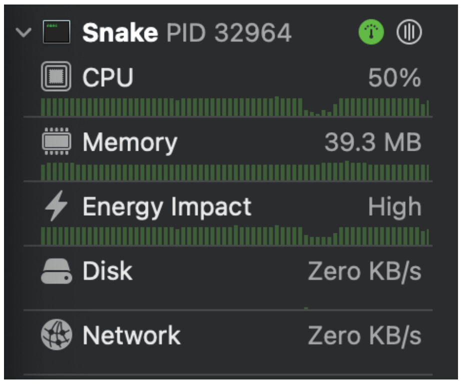
This is the state where the user gets after colliding with an other object or itself. I have added another button here so that if the user wants to quit the game he can do it here as well so he does not have to go back to the main menu. Other than that, the user can press any button to go back to the main menu. In this state I have included the user's final score with the use of sprintf and rendered it on the screen.

Tests - Benchmark on 2019 MacBook Pro - 1.4 GHz Intel Core i5 and 8 GB DDR3 ram



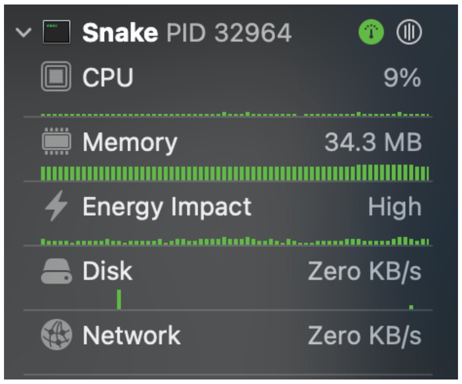
Snake PID 32964	
CPU	11%
Memory	38.6 MB
Energy Impact	High
Disk	Zero KB/s
Network	Zero KB/s

The CPU usage in the Main Menu is very little thanks to SDL_WaitEvent. And memory usage is only 38.6MB (almost constant).



Snake PID 32964	
CPU	50%
Memory	39.3 MB
Energy Impact	High
Disk	Zero KB/s
Network	Zero KB/s

The CPU usage in-game is pretty high, because of SDL_PollEvent. This is why I use FPS caping, which is set to 60, but can be change if the user wants it to. Without the FPS limiter my cpu usage was over 100%.



Snake PID 32964	
CPU	9%
Memory	34.3 MB
Energy Impact	High
Disk	Zero KB/s
Network	Zero KB/s

The CPU usage on the Game Over screen is the lowest.