Programmdokumentation - Snake Game - DE

Zip-Inhalt

Es gibt insgesamt 3 Header-Dateien mit ihren eigenen Dateien vom Typ .c enthalten, die alle in main.c enthalten sind. Diese Header-Dateien sind: snake.h, Gridys.h und ui.h. Diese werden alle benötigt, damit das Programm funktioniert. Es gibt einen Assets-Ordner mit allen erforderlichen Texturen, Kartenlayouts und einer Schriftart. Das Programm arbeitet mit SDL, daher werden die enthaltenen Frameworks benötigt, damit das Programm ausgeführt werden kann.

main.c

In dieser Datei initialisiere ich hauptsächlich die Texturen und starte SDL. Ich habe eine Struktur für die Texturen und für einen Schrifttyp erstellt, damit es einfacher ist, sie in Funktionen zu verwenden. Globale Variablen: (typedef struct contextObject) und für SDL : SDL_Window* window, SDL_Renderer* renderer

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

Dies ist meine Hauptfunktion, mit diesen Eingängen, damit SDL richtig funktioniert. Im ersten Teil erstelle ich Variablen, die Breite und Höhe beziehen sich auf die Größe des SDL-Fensters, die Zeilen und Spalten sind für das Grid, das das Fenster in 16 x 16 gleiche Teile unterteilt. Danach habe ich einige Variablen für die Begrenzung von Frames pro Sekunde, da das Programm (SDL_PollEvent) verwendet, das sehr viel CPU-basiert ist, sodass es möglicherweise mit 100% CPU-Auslastung ausgeführt wird. Auf diese Weise können wir mit dem FPS-Limiter den 100% Verbrauch auf etwa 30-50% reduzieren. Dann habe ich die Funktionen sdl_init und loadMedia aufgerufen, die SDL initiieren und die Texturen und den Schrifttyp laden. Nachdem ich das Raster mit der Area erstellt habe, wird der gesamte Bildschirm in 16 x 16 quadratische Kacheln unterteilt.

Der Grundzustand des Spiels ist das Hauptmenü. Wenn das Programm startet, kann der Benutzer zwischen zwei Karten wählen. Dann kommt das “Game loop”, hier ist der Teil, in dem das Programm zwischen verschiedenen Spielzuständen wechselt und die Hauptaktionen stattfinden. Den ersten Zustand ist das mainMenu - (inMainMenu Funktion). Wenn der Benutzer eine Karte auswählt, ändert sich dieser Status in: loading - (loadmap - Funktion), In diesem Fall werden die ausgewählte Karte und die verschiedenen Arten von Objekten, z. B. der Schlangenkopf und der Ort des Apfels, generiert. Dies dauert nur eine Sekunde, daher ändert sich der Status in inGame, wo der Benutzer die generierten Strukturen sehen und das Spiel spielen kann. Dieser Status funktioniert mit SDL_PollEvent und die Schlange kann mit den Pfeiltasten und mit WASD verschoben werden - Dazu wird die handleKeyboard-Funktion aufgerufen. Die Bewegung arbeitet mit einer Verzögerung Variablen und mit der SnakeMovement - Funktion. Wenn der Benutzer mit einem Apfel kollidiert, wird die Newcords - Funktion aufgerufen, die eine neue Zufallskoordinate für den Apfel angibt, und mit der NewSegment-Funktion wird die Schlange länger. Mit sprintf schreiben wir die aktuelle Punktzahl auf den Bildschirm, basierend darauf, wie viele Äpfel der Benutzer gegessen hat, und mit der Funktion textCreator erstellen wir dieses Textfeld auf dem Bildschirm. Dann fing ich an, die Texturen auf dem Bildschirm zu rendern. Für den Hintergrund verwende ich zwei for-Schleifen, die jede einzelne Fliese des Grid durchlaufen und je nach Art des Standorts eine einfache Grasstruktur oder ein Hindernis rendern. Dann habe ich die anderen Texturen mit

SDL_RenderCopy und SDL_RenderCopyEx gerendert und meine DrawSnakeSegment - Funktion verwendet, mit der die Körperteile der Schlange einzeln gerendert wurden. Nach dem inGame-Status kommt der dead-Status - (deadScreen - Funktion). Es gibt auch einen quit-Status. Am Ende der Hauptfunktion ist es wirklich wichtig, den zugewiesenen Speicher für die Grid freizugeben - (freeArea - Funktion)(damit kein Speichermüll mehr vorhanden ist) und auch SDL zu schließen - (quitSDL - Funktion).

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

Dies ist die Funktion, die SDL startet, und es gibt einige Fehlerbehandlungen, wenn die SDL nicht initialisiert werden kann, sodass die Fehlercodes in die Konsole geschrieben werden.

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

Mit dieser Funktion habe ich die Texturen und die Schriftart importiert, damit sie im Programm verwendet werden können. Wenn die Texturen oder die Schriftart nicht geladen werden können, wird eine Fehlermeldung in die Konsole geschrieben.

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

Diese Funktion ist wirklich wichtig, da beim Beenden des Programms die Texturen zerstört werden müssen, die Schriftartdatei geschlossen werden muss und das (SDL) -Fenster und der Renderer ebenfalls zerstört werden müssen und SDL und TTF beendet werden müssen (TTF_Quit ( ), SDL_Quit ( )).

snake.c / snake.h

In diesen Dateien habe ich hauptsächlich die Grundbewegung der Schlange und einige andere Funktionen eingerichtet, die im Zusammenhang mit der Apfelstruktur stehen.

Erstens habe ich 3 verschiedene Strukturen geschaffen: typedef struct Movement - in dieser legte ich die Richtungen, typedef struct SnakeSegments - Darin befinden sich die Teile der Schlange, ihre Winkel, Koordinaten und die vorherigen und nächsten Teile der Schlange in einer verkettete Liste, typedef struct Snake - Hierbei handelt es sich um die Hauptteile für die Bewegung der Schlange. Sie arbeitet mit einem Verzögerungsmechanismus, der die Schlange langsam von einem Ort zum anderen verschiebt, ohne sich nur sehr schnell zu teleportieren, typedef struct Apple - Dies sind nur die Koordinaten des Apfels.

Wir haben 6 verschiedene Funktionen in dieser Datei:

```
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);
```

Dies ist eine der wichtigsten Funktionen im gesamten Programm. Die gesamte Bewegung der Schlange basiert auf dieser Funktion. Die Bewegung hängt auch von vorherigen Richtungen ab, so dass die Schlange nicht in sich selbst eindringen kann. Wenn sich die Schlange in eine Richtung bewegt, kann sie nicht gleich danach in die entgegengesetzte Richtung gehen, da sie mit ihrem eigenen Kopf kollidieren würde. Also musste ich jede mögliche Bewegung testen. Dafür habe ich Variablen erstellt, die die aktuelle und vorherige Richtung speichern. Die Bewegung selbst ist einfach, wir addieren 1 zur entsprechenden Koordinate und verwenden die Verzögerung (in main.c - Zeile: 118), um die Schlange langsam zu „animieren“.Ich habe auch einen Winkel eingefügt, damit die Textur basierend auf der aktuellen Richtung der Schlange gedreht wird. Dann habe ich eine Kollisionserkennung hinzugefügt. Wenn also die Kopfkoordinaten der Schlange mit den Teilen der Schlange oder einem Hindernis übereinstimmen, ist das Spiel vorbei und es geht in den "dead" Status. Es ist wirklich wichtig, die Elemente der Snake - verkettete Liste nach dem Spiel freizugeben, damit kein verbleibender Speichermüll im System vorhanden ist - (freeSnake - Funktion). Dann habe ich einige Fälle, in denen die Schlange nur aus einem Kopf aufgebaut ist oder mehr Körperteile hat, damit wir wissen, ob es in dieser Koordinate eine Kollision gibt. Auch der Winkel der Körperteile wird vom Kopfwinkel der Schlange übernommen, sodass sie in die richtige Richtung gedreht werden.

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

Mit dieser Funktion können wir die Körperteile der Schlange einzeln rendern, wenn der Kopf der Schlange mit einem Apfel kollidiert.

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

Mit dieser Funktion fügen wir der Schlange neue Körperteile hinzu (funktioniert als verkettete Liste), wenn der Kopf mit einem Apfel kollidiert. Wir haben dynamisch Platz für jedes einzelne Schlangenteil reserviert, wenn die Länge der Schlange steigt.

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

Diese Funktion wird aufgerufen, wenn wir die Startposition des Apfels auf der Karte initialisieren und wenn der Kopf der Schlange mit einem Apfel kollidiert, da der Apfel danach zufällig an eine neue Position gebracht werden muss. Obwohl es nicht so einfach ist, weil der Apfel weder auf einem Hindernis noch unter dem Körper der Schlange laichen darf.

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

Dies ist die Funktion, die die Tastatureingabe verarbeitet. Dies setzt die verschiedenen Richtungen auf wahr oder falsch, je nachdem, welche Taste der Benutzer gedrückt hat. Sie können die Pfeiltasten und auch WASD verwenden, um die Schlange zu steuern.

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

Diese Funktion wird verwendet, um den zugewiesenen Speicher für die Körperteile der Schlange freizugeben. Wenn der Benutzer also mit einer der Karten fertig ist und dann die

andere Karte ausprobieren möchte, indem er zum Hauptmenü zurückkehrt, wird er nicht darauf stoßen Fehler wegen des Speichermülls, der dort zurückgelassen wurde.

GridSys.c / GridSys.h

In diesen Dateien habe ich im Grunde das gesamte SDL-Fenster in 16x16-Einzelteile unterteilt und außerdem eine Funktion geschrieben, die die 2 verschiedenen Kartentypen aus .txt - Dateien einliest und dann die Karte lädt, je nachdem, welchen der Benutzer ausgewählt hat.

Wir haben 3 verschiedene Funktionen in dieser Datei:

```
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 dieser Funktion habe ich ein Layout für die gesamte Karte erstellt, in dem sich die Schlange horizontal und vertikal bewegen kann. In diesem Programm gibt es 16 Spalten und 16 Zeilen. Dafür musste ich zuerst etwas Platz für das gesamte Grid reservieren. Im Grunde ist es also ein zweidimensionales Array, das 16x16 groß ist. Ich habe die Breite und Höhe jedes einzelnen kleinen Quadrats sowie deren x- und y-Koordinaten festgelegt. Und auch diese Funktion hat einen Rückgabewert Grid.

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

Diese Funktion gibt nur den zugewiesenen Speicher für das Grid frei, sodass kein Speichermüll übrig bleibt. Dies wird in main (Zeile: 168.) grundsätzlich am Ende des Programms verwendet.

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

Dies ist die Funktion, die die Karten aus einer .txt - Datei einliest und entscheidet, wo sich die Schlange frei bewegen oder mit einem Hindernis kollidieren kann.

Eine Karte sieht wie folgt aus:

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

Es gibt 16 Zeilen und 16 Spalten. Die Einsen sind die Hindernisse, an die die Schlange nicht gehen kann, und beenden das Spiel, wenn sie versehentlich kollidieren, und die Nullen sind die Orte, an denen sich die Schlange sicher bewegen kann. Mit zwei for-Schleifen können wir jedes einzelne Element der Zahlen durchgehen und sie in das Grid einfügen.

ui.c / ui.h

In diesen Dateien habe ich die Status für das Programm erstellt, daher gibt es in diesem Programm verschiedene Arten von Momenten. Es gibt auch Funktionen zum Erstellen zugänglicher Schaltflächen im gesamten Fenster, die mit der linken Maustaste verwendet werden können.

Zuerst habe ich verschiedene Strukturen initialisiert:

typedef struct textStruct - Diese Struktur hat eine Form, eine Begrenzung der Zeichen, die darin geschrieben werden können, eine Skala, mit der sie größer oder kleiner gemacht werden kann, und eine Farbe.
typedef struct button -Diese Struktur hat eine Position (im Grid), einen Text aus der obigen Variablen, einen Bool-Status (klick), und auch eine Farbe.
typedef enum state - Es gibt 5 Zustände in diesem Programm:: mainMenu, loading, inGame, quit, dead

Diese Datei enthält 5 Funktionen:

```
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);
```

Dies ist die Funktion, die dafür verantwortlich ist, dass das Spiel im Hauptmenü gestartet wird, in dem der Benutzer drei Optionen hat, um entweder auf einer der Karten zu spielen oder das Programm zu beenden. Hier habe ich 3 Schaltflächen erstellt (mit der Funktion buttonCreator) und diese jeweils innerhalb des Grid platziert. Dann habe ich ein Ereignis erstellt (mit SDL_WaitEvent) und die Texte (mit textRenderer) auf dem Bildschirm gerendert. Mit der Funktion buttonAction habe ich diesen Schaltflächen eine Bedeutung hinzugefügt und sie waren verwendbar. Es gibt eine Schaltfläche zum Beenden des Programms “Quit” (geht zum quit Status), eine Schaltfläche für die klassische Schlangenkarte (sie hat einen MapType = 0) (loading Status -> inGame- , siehe in main.c Zeile: 94) und auch eine Schaltfläche für die härtere Karte (sie hat einen mapType = 1) (loading Status -> inGame- , siehe in main.c Zeile: 94).

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

Diese Funktion gibt den erstellten Schaltflächen eine Bedeutung. Dafür brauchte ich ein wenig Hilfe, also benutze ich diesen Link, damit diese Schaltflächen richtig funktionieren: https://stackoverflow.com/questions/39133873/how-to-set-a-gui-button-in-the-win32-window-using-sdl-c .Wenn die Koordinaten der Maus mit der Position des erstellten Textes im Grid übereinstimmen und geklickt wird, nur darauf kann das Programm zum nächsten Status wechseln. Andernfalls ändert sich die Farbe, da es nur auf der Schaltfläche schwebt.

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

Diese Funktion erstellt eine Schaltfläche im Grid, auf die Sie klicken können, um zum nächsten Status im Programm zu gelangen. Es erhält eine Position innerhalb des Grid, seine Breite und Höhe, einen Text, der auf dem Bildschirm gerendert werden muss, und einen skalar Multiplikator. In dieser Funktion wird die Funktion textCreator aufgerufen, die im Folgenden erklärt wird.

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

Diese Funktion erstellt einen Text und verwendet strcpy, um die String, die die Funktion erhalten hat, in den textStruct tempText zu kopieren, der am Ende zurückgegeben wird. Es ist wirklich einfach zu bedienen und kleine Texte damit zu erstellen.

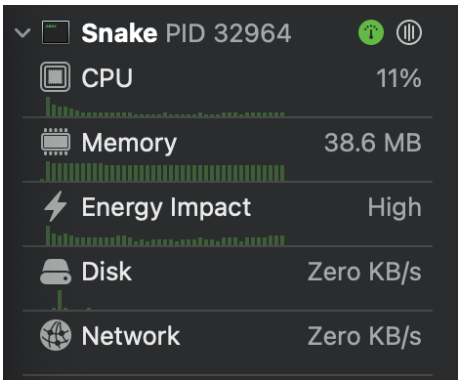
```
void textRenderer(SDL_Renderer* renderer, TTF_Font* font, textStruct*
word);
```

Diese Funktion rendert die Textur auf dem Bildschirm mit TTF und der angegebenen Schriftart aus dem Ordner “Assets".

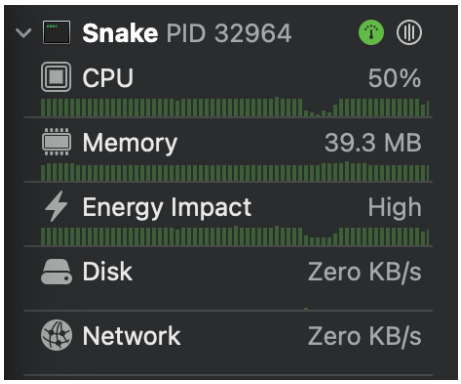
```
void deadScreen(Tile** Grid, int columns, int rows, state*
currentState, int score, SDL_Renderer* renderer, TTF_Font* font);
```

Dies ist der Status, den der Benutzer erhält, nachdem er mit einem anderen Objekt oder sich selbst kollidiert ist. Ich habe hier eine weitere Schaltfläche hinzugefügt, damit der Benutzer, das Speil beenden kann “Quit”, dies auch hier tun kann, damit er nicht zum Hauptmenü zurückkehren muss. Ansonsten kann der Benutzer eine beliebige Taste drücken, um zum Hauptmenü zurückzukehren. In diesem Zustand habe ich die endgültige Punktzahl des Benutzers mit sprintf aufgenommen und auf dem Bildschirm gerendert.

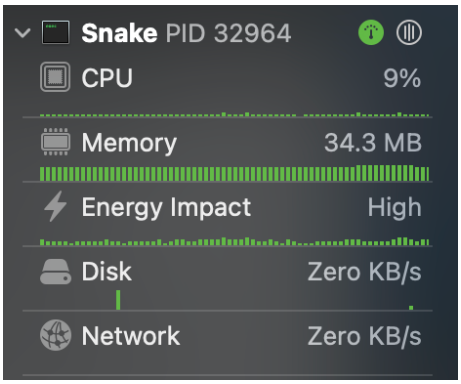
Tests - Benchmark auf 2019 MacBook Pro - 1.4 GHz Intel Core i5 and 8 GB DDR3 ram



Die CPU - Verwendung im Hauptmenü ist sehr wenig dank SDL_WaitEvent. Die Speichernutzung beträgt nur 38,6 MB (fast konstant).



Die CPU - Verwendung im Spiel ist aufgrund von SDL_PollEvent ziemlich hoch. Aus diesem Grund verwende ich die “FPS - Limiter”, die auf 60 eingestellt ist, aber geändert werden kann, wenn der Benutzer dies wünscht. Ohne die “FPS - Limiter” meine CPU -Verwendung war über 100%.



Die CPU - Verwendung auf dem Game Over-Bildschirm ist am niedrigsten.