Praktikum zur Lehrveranstaltung:

Microrechentechnik 1

Versuch 3: C-Programmierung

-

Aufgabe: „Animation“

-

Praktikumsgruppe 13

Datum: 12.01.2022

Ort: Dresden

Namen:		**Note:**
Martin Blümel	**4842445**	
Theo Röhl	**4845481**	
Kleon Dingeldein	**4923572**	
Justus Krenkel	**4878752**	

Inhaltsverzeichnis

1 Einführung

Im Rahmen des MRT Programmierpraktikums soll eine Animationsaufgabe in Sprache C bearbeitet werden. Aus einer vorgegebenen Konfigurationsdatei wird ein Muster ausgelesen und nach definierten Rechenregeln zu Folgebilder weiterverarbeitet und angezeigt. Um das Programm übersichtlich zu gestalten, sind die Programmteile in Module nach dem Prinzip „Divide and Conquer“ aufgeteilt. Das Ausgliedern in einzelne Teilbereiche erlaubte das parallele Arbeiten aller Projektmitglieder. Zum Verwalten und Zusammenführen aller Programmteile wurde das Source Code Management System „Git“ genutzt.

2 Projektablauf

Die Praktikumsaufgabe wurde über den Jahreswechsel 22/23 bearbeitet. Im ersten Schritt machten sich alle Teammitglieder mit der Aufgabenstellung vertraut. Anschließend wurden in einer Gruppenbesprechung am 22.12.2022 grundlegende Punkte zu Bearbeitung der Aufgabe besprochen.
Dabei wurde unter anderem die Datenstruktur „Laufzeitdaten“ festgelegt und die Interpretation des Arrays für den Puffer besprochen. Weiter wurde der grundsätzliche Ablauf des Programmes diskutiert und festgelegt.
Die Praktikumsaufgabe wurde in kleinere Arbeitspakete zerlegt und gleichmäßig im Team verteilt:

- Hauptprogramm und Benutzereingabe: Justus Krenkel
- Einlesen der Datei: Kleon Dingeldein
- Berechnung des nächsten Puffers: Martin Blümel
- Ausgabe des Bildes: Theo Röhl

In den folgenden Tagen wurden die jeweiligen Arbeitspakete von den Teammitglieder bearbeitet. Dabei fand ein regelmäßiger Austausch in der Gruppe statt, um Probleme und Schwierigkeiten bei der Programmierung gemeinsam zu besprechen.
Am 02.01.2023 fand die letzte Besprechung statt. In diesem Termin wurden die einzelnen Module zusammengesetzt und ein finaler Funktionstest durchgeführt.

3 Vorstellung des Programms

3.1 Datenstruktur – struct Laufzeitdaten

Die Definition der „Laufzeitdaten“ ist in der Datei newdata.h vorgenommen. Der Benutzerdefinierte Datentyp wurde als struct implementiert. Er strukturiert die vorgegebenen Daten aus der Konfigurationsdatei, um diese im Programm einfach verfügbar zu machen und die neuberechneten Daten zwischenzuspeichern.

Folgende Daten und Datentypen sind in den struct Laufzeitdaten vorhanden:

```
int X, Y, schritt, gesamtschritte, p;
float delay;
char* puffer;
```

Interpretation des Arrays das in den Puffer geladen und verwaltet wird:

```
// Darstellung des Puffers (Zahlen sind Elemente
// des Puffers):
//
//   |——————— X ————————|
// - 01 02 03 04 05 06 07
// | 08 09 10 11 12 13 14
// Y 15 16 17 18 19 20 21
// | 22 23 24 25 26 27 28
// - 29 30 31 32 33 34 35
```

3.2 Main

In der main.c werden alle benötigten Standardbibliotheken und Module geladen. In der gleichnamigen Funktion

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

wird der Programmablauf strukturiert.

Zuerst werden globale Programminformationen und Parameter initialisiert. Hierzu zählt das Einlesen der Settings.txt und das Initialisieren des Pause-Markers, sowie der Leinwand zur Anzeige der Berechnungsergebnisse und Benutzerinteraktion über die Tastatur.

```
Laufzeitdaten daten = einlesen("settings-1.txt");
int p = 0;
init_pic(daten);
```

Da die Berechnung zyklisch, nach vorgegebenen Rechenregeln und immer aus dem aktuellen Bild erfolgt kommt nun eine while-Schleife zum Einsatz, die mit der Prüfbedingung (1) nie ihre Gültigkeit verliert. Damit wird sichergestellte, dass beliebig viele Berechnungen durchgeführt werden können, bzw. das Programm nach dem letzten Berechnungsschritt geöffnet bleibt und nur durch ein manuelles Steuern des Benutzers geschlossen wird, solange keine Laufzeitfehler auftreten.

Der Schleifenablauf ist im Folgenden kurz skizziert und erläutert:

```
while(1) {
    p=userinput(daten.delay ,p);
    if(daten.schritt < daten.gesamtschritte) {
        daten = calculate_next_pic(daten);
        draw_pic(daten);
    }
}
```

Zu Beginn jedes Durchlaufs wird die Funktion userinput() aufgerufen und wartet eine vordefinierte Zeit auf Benutzereingaben. Die if-Schleife prüft, ob weitere Berechnungsschritte durchzuführen sind oder ob die gewünschte Anzahl bereits erreicht ist. Wenn weitere Berechnungen durchgeführt werden sollen liefert Funktion calculate_next_pic() die neuen Daten welche zum Abschluss des Schleifendurchlaufs mit draw_pic() auf die Leinwand übertragen werden. Der Schleifendurchlauf startet nun von vorn.

3.3 Einlesen der Daten

Das Einlesen einer Konfigurationsdatei ist im C-Modul config.c implementiert. Über den Funktionsaufruf *x=einlesen(“filename.txt“)* wird die Variable x vom Typ „Laufzeitdaten“ mit den eingelesenen Informationen aus der filename.txt befüllt. Nach Funktionsaufruf springt das Programm über die Headerdatei *config.h* in das gleichnamige C-Module. Dort ist der grundlegende Programmablauf in der Funktion *struct Laufzeitdaten einlesen (const char* filename)* implementiert:

1) Öffnen der Datei (ausschließlich zum Einlesen von Informationen). Der Name der Datei wird über den Funktionsaufruf im Hauptprogramm an *config.c* übergeben. Misslingt das Öffnen wird über *perror* eine Fehlermeldung angezeigt.

2) Einlesen der Informationen zu Spalten, Zeilen, Schritt, Gesamtschritte und Pausen zwischen den Animationsschritten in der Funktion *nummerneinlesen(input_file):*

   Diese Funktion bekommt den Pointer auf das geöffnete Dokument übergeben. Der Cursor befindet sich an erster Stelle. In einer while-Schleife wird ein String eingelesen und dieser mit den Schlüsselbegriffen („Zeilen:“, „Spalten:“, usw.) verglichen. Anschließend wird eine funktionsinterne Variable der Struktur *Laufzeitdaten* mit der nun eingelesenen Zahl beschrieben. Dadurch wird bei mehrfacher Angabe einer Information die letzte Angabe abgespeichert. Die Schleife wird abgebrochen, sobald die Funktion erkennt, dass als nächstes der Animations-Puffer eingelesen werden soll.

3) Einlesen des Puffers durch die Funktion *puffereinlesen(data.X, data.Y, input_file):*

   Diese Funktion liest den Puffer der Datei ein und hinterlegt ihn in einem Zwischenspeicher. Dafür werden die maximalen Maße des Bildschirmes und der Pointer auf die Einlesedatei übergeben. Dessen Cursor befindet sich zu Funktionsbeginn vor dem ersten Zeichen des Puffers.
   Der Zwischenspeicher ist von der Datenstruktur *eingelesenerpuffer*, ähnlich wie der Typ *Laufzeitdaten*. Sie umfasst jedoch nur die für diesen Schritt benötigte Zeilen- und Spaltenlänge sowie ein Array für den eingelesen Puffer. Sie ist in *config.h* deklariert, da nur *config.c* diese Datenstruktur verwendet.
   Zunächst wird in das dynamisch erzeugte Array *einlesespeicher* die erste Zeile eingelesen und dessen Länge ermittelt.
   Anschließend wird Speicherplatz für den Zwischenspeicher allokiert. Hintergrund dieser etwas umständlicheren Programmierung ist, dass der Puffer deutlich kleiner als der gegebene Bildschirm sein kann. Dadurch wird weniger Speicher zum Einlesen benötigt.
   Anschließend wird der Animations-Puffer zeilenweise in *einlesespeicher* eingelesen und im Zwischenspeicher (*zs.ptrzs*) abgespeichert.
   Der nicht mehr benötigte *einlesespeicher* wird wieder freigegeben und die Funktion gibt den eingelesenen Puffer sowie dessen Abmaße zurück.

4) Schließen der Einlesedatei.

5) Meldung, falls der eingelesen Puffer nicht zentrierbar ist:
   Aus den Maßen des eingelesenen Puffers und den Maßen des Bildschirmes wird nun geprüft, ob das Bild mittig platziert werden kann. Es wird eine Meldung ausgegeben, falls dies nicht möglich ist. Das Programm wird dennoch fortgeführt.

6) Zentrieren des Puffers in der Funktion *zentrieren(zwischenspeicher, data.X, data.Y):*
   Diese Funktion setzt den eingelesenen Puffer mittig in die Dimensionen der Leinwand. Dafür wird der benötigte Speicher allokiert. In einer doppelten for-Schleife jeder Pixel des Bildschirmes geprüft, ob es von einem Pixel des eingelesen Puffers beschrieben werden muss.
   Rückgegeben wird ein Array, das den Bildschirm mit dem zentrierten eingelesenen Puffer beschreibt. Das Array berücksichtig außerdem den zur weiteren Berechnung benötigten Rand.

7) Rückgabe der befüllten Datenstruktur an das Hauptprogram.

3.4 Berechnung des nächsten Animationsschrittes

Das Berechnen des nächsten Animationsschrittes ist im Modul engine.c implementiert. Die Funktion calculate_next_pic() realisiert eine pixelweise Neuberechnung des übergebenen Animationspuffers, das Hochzählen des Animationsschrittes und die Rückgabe der Werte als Struct vom Typ Laufzeitdaten mit den aktualisierten Parametern.

- Funktionsaufruf im Modul main.c

```
// Naechsten Animationsschritt berechnen (4.3, in engine.c):
daten = calculate_next_pic(daten);
```

- Implementierte Funktion im Modul engine.c

```
struct Laufzeitdaten calculate_next_pic(struct Laufzeitdaten caldata){
```

- Regeln zur Neuberechnung des Animationspuffers:
  1. Ein freier Pixel mit exakt 3 belegten Nachbarpixeln wird zu einem belegten Pixel.
  2. Ein belegter Pixel mit weniger als zwei belegten Nachbarpixeln wird frei.
  3. Ein belegter Pixel mit zwei oder drei belegten Nachbarpixeln bleibt belegt.
  4. Ein belegter Pixel mit mehr als drei belegten Nachbarpixeln wird frei.

Ebenfalls ist gewünscht das die Randpixel, die zu späterem Zeitpunkt nicht sichtbar sind, neu zu berechnen. Um diese Regeln umzusetzen wird in einem weiteren Laufzeitdatenstruct „betdata“, um einen Randpixel größerer Animationspuffer, allokiert, mit leeren Pixeln befüllt und anschließend mittig durch den übergebenen Animationspuffer ergänzt.

```
35          betdata.puffer = malloc(((betdata.X+2)*(betdata.Y+2))*sizeof(char));
```

Damit ist es möglich auch die acht umliegenden Pixel der Randpixel zu betrachten. Über 2 for-Schleifen wird dann der Animationspuffer durchlaufen.

```
for (int i=1; i<=(betdata.Y); i++) {
                for (int j=1; j<=(betdata.X); j++) {
```

Dabei werden die umliegenden und der zu betrachtende Pixel über if-

Anweisungen auf Freiheit geprüft. Umliegend belegte Pixel werden über einen Zähler „k“ mitgezählt.

Beispiel:

```
//Analyse der umliegenden Pixel
//Ecke oben links
if(betdata.puffer[(j+i*(betdata.X+2))-(betdata.X+3)]=='x'){
        k= k+1;
}
//oben Mitte
if(betdata.puffer[(j+i*(betdata.X+2))-(betdata.X+2)]=='x'){
        k= k+1;
```

Je nach Regel werden die geänderten Pixel oder gleich gebliebenen Pixel auf den Animationspuffer des Rückgabestructs „nextdata“ geschrieben.

Beispiel:

```
//freier Pixel mit exakt 3 belegten Nachbarpixeln wird zu belegten Pixel
if (k == 3){
        nextdata.puffer[(j-1)+(i-1)*(caldata.X+2)] = 'x';
        k = 0;
}
```

- Hochzählen des Animationschrittes

```
//hochzählen des Berechnungsschrittes vor der Rückgabe
nextdata.schritt ++;
```

- Rückgabe von nextdata über return

```
196         //Rückgabe des berechneten Schrittes an die Main-Schleife
197         return nextdata;
198 }
```

3.5 Ausgabe des Bildes

Unter der Ausgabe des Bildes fallen die Schritte, das Anzeigefenster zu initialisieren, den Hintergrund anzuzeigen und jeden einzelnen Puffer neu darzustellen. Dazu wurden im Praktikum fertige Funktionen im Modul graphic.c mitgegeben. Die Aufgabe gestaltete sich demnach darin, die einzelnen Funktionen in geeigneter Weise mit den benötigten Daten über das Modul „gfx.c“ auszurufen.
Die Funktion Init_pic() initialisiert des Fensters. Dafür benötigt es die eigelesenen Konfigurationsdaten als Typ Laufzeitdaten. Die Funktion nutzt die vorgefertigte Lösung zum Erstellen eines Fensters (grafik_init_window()), welche ein Fenster in der Größe 600x800 Pixel erstellt. Diese Dimension ist unabhängig vom eingelesenen Puffer und ist statisch im Modul „grapic.c“ definiert. Außerdem muss der Hintergrund des Fensters noch initialisiert werden. Dies passiert mit der zweiten vorgefertigten Funktion, „grafik_create_paint_area()“. Diese erhält die mit „init_pic“ erhaltenen Daten. Aus diesen Daten is die Größe des Puffers, sowie die anzuzeigende Fläche relevant. Für die Berechnung der Ränder wurde das eigentliche Feld an den Rändern um einen Pixel erweitert, sodass es nun nicht mehr „x“ Pixel breit ist, sondern „x+2“ und der anzuzeigende Bereich demnach nicht mehr von null bis „x“ läuft, sondern von eins bis „x+1“. Gleiches gilt analog für „y“. Dadurch werden Randpunkte vom anzuzeigenden Rechteck übergeben und nicht vom ganzen Feld. Länge und Breite des Puffers ist für die Auflösung wichtig. Je nach Größe des eingelesenen Puffers wird die Auflösung so angepasst, dass der Puffer das gesamte Fenster einnimmt. Würde man die Auflösung statisch festlegen, würde ein sehr kleiner Puffer das Fenster gar nicht füllen oder ein sehr großer Puffer nicht vollständig angezeigt werden. Diese Vorkehrungen sieht man bei der Initialisierung der Leinwand zwar noch nicht, da der Bildschirm noch einfarbig ist und keine Punkte gezeichnet werden, sie sind aber für die Animation im nächsten Aufgabenteil unerlässlich. Damit ist die Funktion „init_pic()“ abgeschlossen und das Programm läuft im Main Modul weiter.
Mit jedem frischen Puffer wird im Hauptprogramm die Funktion „draw_pic()“ ausgerufen, welche den aktuellen Puffer im gesamten Datenpaket erhält. Bevor dann gemalt wird, sperrt die Funktion „grafik_lock_for_painting()“ die Leinwand. Das bestehende Bild wird damit eingefroren, bis die Leinwand mit den neuen Pixeln übermalt wurde. Sonst würde sich das Bild punktweise verändern und es gäbe keinen sauberen Übergang zwischen den einzelnen darzustellenden Bildern. Der aktuelle Puffer wird nun punktweise, von links nach rechts und von oben nach unten, abgearbeitet. Ist im zu prüfenden Feld ein „x“ eingetragen, wird das entsprechend versetzte Leinwandfeld mit der Farbe „Valhalla“ bemalt, ist ein Punkt eingetragen, wird mit der Hintergrundfarbe „Twine“ bemalt.

Ist das Malen abgeschlossen, wird mit der Funktion „grafik_unlock_and_show()“ die Leinwand wieder freigegeben, sodass das neue Bild zu sehen ist. Damit ist die Funktion „draw_pic()“ abgeschlossen und wird nach Berechnung eines neuen Puffers erneut ausgerufen.

3.6 Benutzereingabe

Benutzereingaben werden mit der Funktion userinput() geprüft. Die Funktion wurde in das Modul ui.h und ui.c ausgelagert und realisiert folgende Benutzereingaben:

1. Das Programm Pausieren – Tastendruck „Leertaste“
2. Den nächsten Berechnungsschritt durchführen – Tastendruck „.“
3. Das Programm weiterlaufen lassen – Erneuter Tastendruck „Leertaste“
4. Das Programm zu jeden beliebigen Zeitpunkt Beenden – Tastendruck „q“

Des Weiteren wird durch das Warten auf Benutzerinteraktion die gewünschte verzögert des Programms zwischen zwei Berechnungsschritten realisiert.
Die Funktion userinput() nimmt zwei Variablen vom Typ float und int entgegen.

```
int userinput(float delay, int p){...}
```

Die erste Variable übermittelt die Wartezeit, die zweite den Betriebsmodus des Programms. Dabei wird zwischen Normalbetrieb (0) und Pausebetrieb (1) unterschieden.
Um die Wartezeit innerhalt der Funktion userinput() weiter benutzen zu können muss diese zuerst selbst in den Typ int umgerechnet werden, da die verwendete Funktion graphic_user_input() die Wartezeit nur in Millisekunden entgegennimmt.

```
int d = floor(delay * 1000);
```

Je nach übergebenen Betriebsmodus verarbeitet die Funktion nun folgende Benutzereingaben über switch-case-Anweisungen.

Normalbetrieb (p == 0):

- Beenden des Programms (q)

- Start des Pausebetrieb (Leertaste)
  Setzt den Wert p = 1 und übergibt diesen zurück an die Main-Funktion.

Pausebetrieb (p == 1):
Im Pausebetrieb wird innerhalb der userinput-Funktion mittels while(1) gewartet bis eine Benutzereingabe erfolgt, welche dann die Schleife abbricht.

- Beenden des Programms (q)

- Nächstes Bild berechnen (.)
  Bricht aus der while-Schleife aus und liefert den aktuellen Betriebsmodus (p=1) zurück.

- Stop des Pausebetrieb (Leertaste)
  Setz den Wert p=0 und übergibt diesen zurück an die Main-Funktion.

4 Probleme und Schwierigkeiten

- VM, Git einrichten
- (Übergang von VS zu VM)
- Umgang mit Pointer (bei Strings, Dateien,…)
- .c Dateien nicht in .h Dateien (Umgang mit c. und h. Dateien)
- Umgang mit nicht initialisierten Variablen in einem Struct
- Der Struct Laufzeitdaten verfügt über einen int, welcher den aktuellen Betriebsmodus des Programmes anzeigt. Da intern bei der Berechnung der Bilder der Struct neu erstellt wird, der Betriebsmodus aber erst nachträglich in der Definition hinzugefügt wurde, kam es zu dem Phänomen, dass der int mit zufälligen Werten befüllt, wurde uns damit nicht sinnvoll verwendet werden konnte.
- Effiziente Übergabe von Daten zwischen Funktionen und Modulen
- Allokieren und befreien von Speicher