3. Aufgabe: Bitmap-Datei

1 Einleitung

In dieser Programmieraufgabe soll eine Grafik erstellt und gespeichert werden. Es wurde das Bitmap-Format von Microsoft gewählt, da es recht einfach erstellt und von den meisten Grafikprogrammen gelesen werden kann. Diese Aufgabe beschränkt sich dabei auf eine Farbtiefe von 24 Bit, genannt *true color*, mit jeweils 8 Bit für die Farben rot, grün und blau (RGB), und versucht nicht allgemein das Bitmap-Grafikformat mit allen Optionen zu implementieren.

2 Grundlagen

Eine Bitmap-Datei ist wie folgt aufgebaut:

file header (14 Byte)	info header (40 Byte)	[look up table, LUT]	pixel data

- Im *file header* stehen Informationen zur Datei im Allgemeinen
- Im *info header* stehen Angaben zum Bild wie Bildgröße, Art der Farben u.s.w.
- Die *look up table* (LUT) ist eine optionale Farbtabelle, die in diesem Praktikum nicht verwendet wird
- Im Block *pixel data* stehen die Pixeldaten des Bildes

Im Folgenden werden zunächst einige elementare Datentypen eingeführt, danach werden die vier Blöcke näher erläutert. Schließlich befassen wir uns mit der Speicherbelegung für die benötigten Strukturen.

2.1 Einige elementare Datentypen

Damit das Dateiformat unabhängig von Rechnerplattform und Software ist, werden vier neue Datentypen eingeführt, die unter Visual Studio 2017 gemäß Tabelle 1 definiert werden.

Typenname	in VS 2015	Beschreibung
BYTE	unsigned char	1 Byte mit Wertebereich 0 bis $2^8 - 1$
WORD	unsigned short	2 Byte mit Wertebereich 0 bis $2^{16} - 1$
DWORD	unsigned int	4 Byte mit Wertebereich 0 bis $2^{32} - 1$
LONG	long	4 Byte mit Wertebereich -2^{31} bis $2^{31} - 1$

Tabelle 1: Elementare Datentypen

2.2 Der file header

Der *file header* fasst die in Tabelle 2 aufgelisteten Elemente zusammen. Er hat eine Länge von 14 Byte und die Elemente werden in einer Struktur gebündelt.

Datentyp	Name	Wert	Beschreibung
WORD	type	19778	Die Buchstaben 'B' und 'M' als Kennung: 'B' + 256*'M' = 19778
DWORD	fileSize	<Dateigröße>	Größe der gesamten Datei
DWORD	reserved	0	Von Microsoft für spätere Zwecke reserviert
DWORD	offBytes	54	Position der Pixeldaten in der Datei

Tabelle 2: Die Struktur für den *file header*.

type. In die ersten zwei Bytes werden die ASCII-Codes der Buchstaben 'B' und 'M' gespeichert und in dem Element *type* zusammengefasst. (Beim Schreiben des Typs *WORD* wird das niederwertige Byte zuerst geschrieben. Mit der Formel *type*='B'+256*'M' wird 'B' in das niederwertige und 'M' in das höherwertige Byte geschrieben.)

fileSize. Die Dateigröße *fileSize* muss berechnet werden und gibt in Summe die Größe der vier Blöcke in Byte an.

reserved. Dieses dritte Element ist für spätere Versionen des Formats vorgesehen und wird bis dahin immer mit dem Wert 0 beschrieben.

offBytes. Schließlich wird als viertes Element die Position der Pixeldaten innerhalb der Datei angegeben. Für Bilder im *true color* Format beträgt dieser Wert immer 54, was der Summe der Größe der ersten zwei Blöcke entspricht.

2.3 Der info header

Der *info header* setzt sich aus den in Tabelle 3 aufgelisteten Elementen zusammen und hat eine Größe von 40 Byte.

Datentyp	Name	Wert	Beschreibung
DWORD	infoSize	40	Größe des *info headers*
LONG	width	<Breite>	Breite des Bilds in Pixeln
LONG	height	<Höhe>	Höhe des Bilds in Pixeln
WORD	planes	1	Anzahl der Bildebenen (immer 1)
WORD	bitCount	24	Bits pro Pixel (hier jeweils 8 Bit für die Farben)
DWORD	compression	0	Art der Kompression (0 für keine Kompression)
DWORD	imageSize	0	Größe der Pixeldaten in Byte (0 = automatisch)
LONG	xPixelsPerMeter	1000	Pixel/Meter in x-Richtung (hier 1 Pixel = 1 mm)
LONG	yPixelsPerMeter	1000	Pixel/Meter in y-Richtung (hier 1 Pixel = 1 mm)
DWORD	colorUsed	0	Anzahl der verwendeten Farben in LUT
DWORD	colorImportant	0	Anzahl der wichtigen Farben in LUT

Tabelle 3: Die Struktur für den info header.

In diesem Praktikum muss nur die Bildgröße angepasst werden. Alle anderen Werte bleiben unverändert. Auf die Details der anderen Elemente wird hier nicht weiter eingegangen.

2.4 Die look up table, LUT

Die LUT dient einigen Varianten des Bitmap-Formats und ist von daher optional. In diesem Praktikum wird sie nicht verwendet und wird hier nicht weiter behandelt.

2.5 Der Block pixel data

Im Block *pixel data* stehen die Pixeldaten des Bildes. Je nach gewählter Variante werden hier entweder direkt die Farben eingetragen (*true color*), oder es finden sich hier Indices, mit denen über die LUT die Farben der Pixel ermittelt werden.

In diesem Praktikum verwenden wir ausschließlich die Variante *true color*. Für die Farben erstellen wir eine weitere Struktur, siehe Tabelle 4. Die Pixel eines Bildes werden beginnend mit der untersten Zeile des Bildes von links nach rechts in die Datei eingetragen.

Datentyp	Name	Wert	Beschreibung
BYTE	blue	<blau>	blauer Farbanteil im Pixel
BYTE	green	<grün>	grüner Farbanteil im Pixel
BYTE	red	<rot>	roter Farbanteil im Pixel

Tabelle 4: Die Struktur für *true color* pixel.

Wichtiger Hinweis: Auch wenn die Bilder beliebig breit sein können, muss in der Datei die Anzahl der Bytes pro Zeile ein vielfaches von vier betragen! Um das zu erreichen, müssen am Ende einer jeden Pixelzeile zwischen null und drei Bytes angehängt werden. Diese angehängten Bytes müssen auch bei der Berechnung der Dateigröße im *file header* berücksichtigt werden. Die angehängten Bytes sind nur ein Lückenfüller und werden von Grafikprogrammen bei der Anzeige nicht ausgewertet.

Beispiel: Wenn ein Bild 19 Pixel breit sein soll, so werden im *true color* Format pro Zeile 3*19=57 Bytes für die Pixeldaten benötigt. Um ein Vielfaches von 4 zu erreichen (60), müssen beim Speichern bei jeder Zeile drei Byte angehängt werden. Siehe auch Abbildung 1 für eine Breite von fünf Pixeln.

b g r	b g r	b g r	b g r	b g r	x
b g r	b g r	b g r	b g r	b g r	x
b g r	b g r	b g r	b g r	b g r	x
b g r	b g r	b g r	b g r	b g r	x
b g r	b g r	b g r	b g r	b g r	x
b g r	b g r	b g r	b g r	b g r	x

Abbildung 1: Beispiel für angehängte Bytes: Bei einer Breite von 5 Pixeln muss an jede Zeile ein Byte angehängt werden, damit die Anzahl der Bytes ein Vielfaches von vier ergibt.

2.6 Ausrichtung der Elemente einer Struktur im Speicher

Die Elemente einer Struktur werden standardmäßig im Speicher auf 8 Byte-Blöcke ausgerichtet. Dadurch wird vermieden, dass ein Element unnötig eine 8 Byte-Grenze überschreitet. Das führt dazu, dass an einigen Stellen eine Lücke in Kauf genommen wird, um im Gegenzug die Geschwindigkeit zu erhöhen. Diese Lücken führen dazu, dass eine Struktur im Speicher mehr Platz als die Summe ihrer Elemente belegt.

Beim Schreiben oder Lesen einer Struktur als Ganzes führt das zu Problemen. Um das zu vermeiden, kann dem Compiler explizit mitgeteilt werden, dass er seine Elemente an einem anderen Raster ausrichtet. Folgende Zeile sorgt dafür, dass alle folgenden Strukturen auf ein Raster von nur 2 Byte ausgerichtet werden:

```
#pragma pack(push, 2)
```

Der Befehl für den Pre-Compiler *pragma* gibt Anweisungen an den Compiler. Der Befehl *pack* ist für die Ausrichtung von Strukturelementen im Speicher zuständig. Mit *push* wird dafür gesorgt, dass der alte Wert für die Speicher-Ausrichtung gespeichert wird. Mit der Zahl 2 werden die Elemente von Strukturen ab dieser Zeile auf zwei Byte ausgerichtet. Die folgende Zeile hebt den vorherigen Befehl wieder auf:

```
#pragma pack(pop)
```

Mit *pop* wird die zuvor mit *push* zwischengespeicherte Speicher-Ausrichtung wieder reaktiviert. Schreiben Sie diese beiden Zeilen um Ihre Struktur-Deklarationen herum, um Probleme beim Schreiben und Lesen einer Bitmap-Datei zu vermeiden.

3 Obligatorische Aufgaben

3.1 Strukturen anlegen

Deklarieren Sie in einer Headerdatei *bitmap.h* die oben angegebenen Typen und Strukturen und vergeben Sie mit *typedef* folgende Namen: BYTE, WORD, DWORD, LONG, tFileHeader, tInfoHeader und tRgb. Prüfen Sie mit *sizeof(tFileHeader)*, dass die Größe der Struktur genau 14 Byte beträgt.

Erstellen Sie als nächstes wieder in *bitmap.h* eine Struktur für die gesamte Bitmap gemäß Tabelle 5. Geben Sie der Struktur mit *typedef* den Namen *tBmp*, definieren Sie in *main()* einen Zeiger auf diesen Typ mit Namen *bmp* und initialisieren Sie ihn mit NULL.

Datentyp	Name	Beschreibung
tFileHeader*	pFile	Zeiger auf den *file header*
tInfoHeader*	pInfo	Zeiger auf den *info header*
tRgb**	pixel	Zeiger für Pixeldaten
char*	data	Zeiger zur dynamischer Speicherreservierung

Tabelle 5: Struktur für den Datentyp *tBmp*.

Ähnlich dem Datentyp *FILE* beim Arbeiten mit Dateien soll der Datentyp *tBmp* als Bindeglied zwischen allen Funktionen zum Bearbeiten der Bitmap dienen.

3.2 Strukturen initialisieren

Erstellen Sie in einer Quellcodedatei *bitmap.c* eine Funktion zum Erstellen und Initialisieren einer Bitmap:

```
tBmp *createBmp(int width, int height);
```

In einem ersten Schritt muss dynamischer Speicher reserviert und die Zeiger dafür initialisiert werden. Es werden drei Speicherblöcke benötigt:

1. Speicher für die Struktur *tBmp*.
2. Speicher für die Daten der Bitmap. In diesem Speicherblock werden hintereinander der *file header*, der *info header* und die Pixeldaten gespeichert. Dieser Block kann später als ganzes in eine Datei geschrieben werden.

3. Speicher für einen Zeigervektor, über den die Pixel als zweidimensionaler Vektor angesprochen werden können.

Für eine Bitmap mit 3×3 Pixeln ist die Verknüpfung der Speicherblöcke in Abbildung 2 schematisch dargestellt. *xxx* steht für die angehängten Bytes (hier drei Bytes), damit die Anzahl der Bytes pro Zeile durch vier teilbar ist.

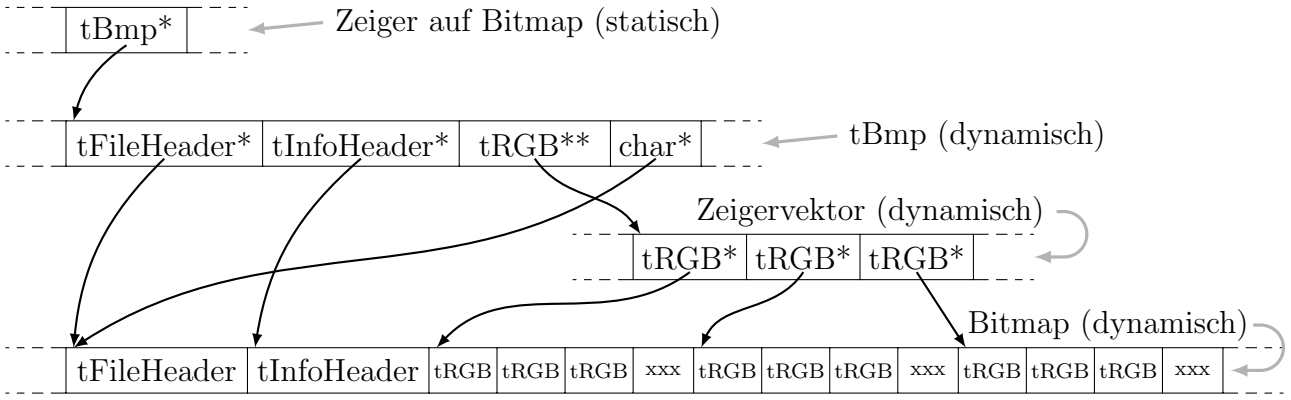

Abbildung 2: Speicherbelegung der Stukturen.

Nachdem die Datenstrukturen erstellt und verlinkt wurden, müssen danach alle Datenfelder initialisiert werden. Gehen Sie dafür die Tabellen 2 und 3 durch und füllen Sie die Strukturelemente entsprechend auf. Initialisieren Sie die Pixeldaten mit null (schwarzes Bild).

Bei einem Fehler soll die Adresse NULL zurückgeben werden.

3.3 Bitmap freigeben

Erstellen Sie in der Quellcodedatei *bitmap.c* eine Funktion, die den Speicher der Bitmap freigibt:

```
void freeBmp(tBmp *bmp);
```

Die Funktion prüft, ob die Bitmap vorhanden ist (d.h. bmp!=NULL), und gibt dann ggf. den reservierten Speicher frei.

3.4 Bitmap in eine Datei schreiben

Erstellen Sie in der Quellcodedatei *bitmap.c* eine Funktion, welche die erstellte Bitmap binär in eine Datei schreibt:

```
int writeBmp(char *filename, tBmp *bmp);
```

Die Bitmap kann mit einem einzigen *fwrite()*-Befehl in die Datei geschrieben werden.

3.5 Funktion zum Zeichnen eines Pixels

Erstellen Sie eine Funktion zum Zeichnen eines Pixels:

```
void setPixelBmp(int x, int y, BYTE red, BYTE green, BYTE blue, tBmp *bmp);
```

Die Funktion prüft, ob eine Bitmap vorhanden ist und ob sich die Koordinaten x und y innerhalb des Bildes befinden. Ist dies der Fall, trägt die Funktion die übergebenen Farben an der entsprechenden Stelle in das Bitmap ein.

3.6 Zusammenfassung in einem Programm

Fassen Sie die bisher erstellten Programmstücke in einem Hauptprogramm zusammen. Fragen Sie vom Benutzer die Größe des Bildes ab, erstellen Sie die Bitmap, zeichnen Sie einige Pixel mit unterschiedlichen Farben und schreiben Sie die Bitmap in eine Datei. Das Bild können Sie z.B. mit *Paint* von Microsoft lesen und weiter bearbeiten.

4 Optionale Aufgaben

4.1 Funktion zum Zeichnen einer Linie

Erstellen Sie eine Funktion zum Zeichnen einer Linie:

```
void drawLineBmp(int x1, int y1, int x2, int y2,
    BYTE red, BYTE green, BYTE blue, tBmp *bmp);
```

Es sollen beliebige Koordinaten möglich sein. Liegen die Koordinaten außerhalb des Bildes, so soll nur der sichtbare Teil der Linie gezeichnet werden.

4.2 Funktion zum Zeichnen eines Kreises

Erstellen Sie eine Funktion zum Zeichnen eines Kreises:

```
void drawCircleBmp(int xmid, int ymid, int radius,
    BYTE red, BYTE green, BYTE blue, tBmp *bmp);
```

4.3 Funktion zum Zeichnen eines ausgefüllten Rechtecks

Erstellen Sie eine Funktion zum Zeichnen eines ausgefüllten Rechtecks:

```
void drawBarBmp(int x1, int y1, int x2, int y2,
    BYTE red, BYTE green, BYTE blue, tBmp *bmp);
```

Viel Erfolg beim Programmieren!