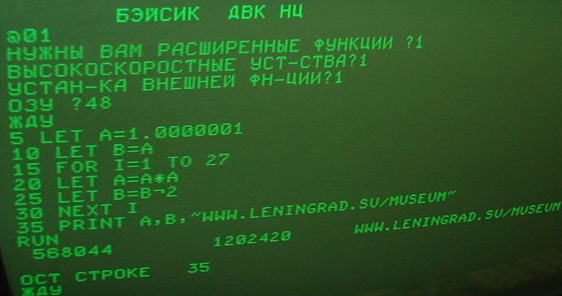


C++ Workshop

Addendum zum 3. Block: Bitmap-Framework, 18.05.2012

Robert Schneider, Markus Jung | 23. Mai 2012



```
БЭЙСИК  ДВК  НЧ
001
НУЖНЫ ВАМ РАСШИРЕННЫЕ ФУНКЦИИ ?1
ВЫСОКОСКОРОСТНЫЕ УСТ-СТВА?1
УСТАН-КА ВНЕШНЕЙ ФН-ЦИИ?1
03У ?48
ЖДУ
5 LET A=1.0000001
10 LET B=A
15 FOR I=1 TO 27
20 LET A=A*A
25 LET B=B-2
30 NEXT I
35 PRINT A,B,"WWW.LENINGRAD.SU/MUSEUM"
RUN
568044      1202420      WWW.LENINGRAD.SU/MUSEUM
ОСТ СТРОКЕ      35
ЖДУ
```

Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

- 1 Bitmap24 und Algorithmen
- 2 BatchBitmap24 und Zeichner-Instanzen

1 Bitmap24 und Algorithmen

Markus hat eine Doxygen-Dokumentation zum Bitmap-Framework geschrieben, diese findet ihr unter:

`https://github.com/downloads/kit-cpp-workshop/
workshop-ss12-03/bitmap-framework-doc-html.zip`

In dieser zip-File liegt die Dokumentation als html-Variante, Ausgangspunkt ist `index.html`.

Die Klasse `Bitmap24` ist eine minimale Klasse zum Abspeichern von Bitmaps als `.bmp`-Dateien. Sie enthält dafür die member function `save`, der am einfachsten ein String-Literal als Parameter übergeben wird, etwa `save("filename.bmp")`.

Die Klasse `Bitmap24` ist eine minimale Klasse zum Abspeichern von Bitmaps als `.bmp`-Dateien. Sie enthält dafür die member function `save`, der am einfachsten ein String-Literal als Parameter übergeben wird, etwa `save("filename.bmp")`.

Wie speichere ich eine Bitmap?

Die Bilddaten (Pixel) müssen im Puffer von `Bitmap24` liegen, sie können dorthin mit `setPixel` geschrieben werden. Anschließend speichert ein Aufruf der member function `save` die Bitmap als Datei. Geläufige C++ Implementierungen (gcc, MSVC) werden diese Datei in das Arbeitsverzeichnis legen, etwa im Ordner der ausführbaren Datei. Ebenfalls bei den geläufigen Implementierungen kann man dem Dateinamen noch einen Pfad voranstellen.

Objektorientierung im Bitmap-Framework

Dieser Workshop soll die Interaktion von Objekten (Instanzen) näherbringen. Zum Zeichnen einer Line soll daher eine Linienzeichner-Klasse existieren. Eine Linienzeichner-Instanz soll dann mit einer Instanz einer „Leinwand“-Klasse interagieren.

Objektorientierung im Bitmap-Framework

Dieser Workshop soll die Interaktion von Objekten (Instanzen) näherbringen. Zum Zeichnen einer Line soll daher eine Linienzeichner-Klasse existieren. Eine Linienzeichner-Instanz soll dann mit einer Instanz einer „Leinwand“-Klasse interagieren.

Damit die beiden Instanzen (Linienzeichner-Instanz, Leinwand-Instanz) miteinander interagieren können, brauchen sie sozusagen eine gemeinsame Sprache. Hier basiert diese auf der Verwendung von Pixeln: Die Linie weiß, wie sie in Pixeln aussieht, und die Leinwand stellt eine Möglichkeit zur Verfügung, pixelweise auf sie zu zeichnen.

Wir werden später sehen, dass es eigentlich vorteilhaft ist, die gemeinsame Sprache zwischen Zeichnern (Linie, Kreis usw.) und Leinwand abstrakt in Form eines Interface festzuhalten. Um das Programm einfach zu halten, wird zunächst auf diese Abstraktion verzichtet.

Wir werden später sehen, dass es eigentlich vorteilhaft ist, die gemeinsame Sprache zwischen Zeichnern (Linie, Kreis usw.) und Leinwand abstrakt in Form eines Interface festzuhalten. Um das Programm einfach zu halten, wird zunächst auf diese Abstraktion verzichtet.

Die `Bitmap24`-Klasse, die eigentlich zum Abspeichern einer Bitmap-Datei dient, reicht aus, um als Leinwand-Klasse verwendet werden zu können: Man kann pixelweise auf das hinterlegte Bitmap zugreifen, dies entspricht einem pixelweisen Zeichnen (`setPixel`).

Bevor man damit beginnt, eine Zeichner-Klasse zu schreiben, sollte man eine Idee haben, wie das zu Zeichnende in Pixeln aussieht. Man benötigt also eine Vorgehensweise, eine Menge von Pixeln zu „setzen“ (den Farbwert zu setzen) \implies ein Algorithmus.

Bevor man damit beginnt, eine Zeichner-Klasse zu schreiben, sollte man eine Idee haben, wie das zu Zeichnende in Pixeln aussieht. Man benötigt also eine Vorgehensweise, eine Menge von Pixeln zu „setzen“ (den Farbwert zu setzen) \implies ein Algorithmus.

Der einfachste Fall hierfür ist der Algorithmus zum Füllen der gesamten Leinwand mit einer Farbe. Es muss hierbei lediglich jedem Pixel diese Farbe zugewiesen werden, der Algorithmus benötigt also als Eingabe eine Farbe sowie die Leinwand, auf der er agieren soll.

Implementierung eines Zeichen-Algorithmus (1)

In unserem einfachen Programm können wir nun eine Funktion schreiben:

```
void fillCanvas (Bitmap24& targetCanvas , Color fillColor );
```

Beachte: Wir müssen die Leinwand als Referenz übergeben, da der Algorithmus ja auf dem übergebenen »Ding« arbeiten soll, und nicht auf einer Kopie (= nicht auf einem neuen »Ding«).

Implementierung eines Zeichen-Algorithmus (2)

Es stehen nun innerhalb der Funktion `fillCanvas` die Eigenschaften und Fähigkeiten der Instanz von `Bitmap24` zur Verfügung, man kann also pixelweise die Bitmap zeichnen und die Höhe sowie Breite abfragen. Da die Pixel bei $(0, 0)$ beginnen und die Leinwand bei $(\text{width} - 1, \text{height} - 1)$ zu Ende ist, sind alle nötigen Informationen für das Füllen der Leinwand vorhanden.

2 BatchBitmap24 und Zeichner-Instanzen

Man könnte basierend auf dem Algorithmus, der oben beispielhaft als Funktion designed wurde, bereits ein eine Klasse bauen. So könnten Instanzen dieser Klasse bspw. die Farbe als data member speichern und eine member function enthalten, die als Parameter nur noch die Leinwand annehmen muss. Was wäre der Vorteil davon?

Man könnte basierend auf dem Algorithmus, der oben beispielhaft als Funktion designed wurde, bereits ein eine Klasse bauen. So könnten Instanzen dieser Klasse bspw. die Farbe als data member speichern und eine member function enthalten, die als Parameter nur noch die Leinwand annehmen muss. Was wäre der Vorteil davon?

Z.B. könnte man eine solche Instanz im Voraus erzeugen, und anschließend auf eine Reihe von Leinwänden anwenden. So könnte man etwa eine FüllDieLeinwand-Instanz pro Grundfarbe anlegen. Aber richtig groß ist der Vorteil noch nicht, das ändert sich aber mit ein paar mehr Schritten.

Wenn ich die Menge von Zeichner-Klassen betrachte, so fallen mir zwei Punkte auf:

- Die Zeichner-Instanzen entsprechen ungefähr Werkzeugen in der Realität (betrachte auch den Werkzeugkasten im Zeichenprogramm deiner Wahl, etwa m\$PAIN).
- Die Zeichner-Instanzen sind alle Zeichner-Instanzen (triviale Aussage? nein!)

Wenn ich die Menge von Zeichner-Klassen betrachte, so fallen mir zwei Punkte auf:

- Die Zeichner-Instanzen entsprechen ungefähr Werkzeugen in der Realität (betrachte auch den Werkzeugkasten im Zeichenprogramm deiner Wahl, etwa m\$PAIN).
- Die Zeichner-Instanzen sind alle Zeichner-Instanzen (triviale Aussage? nein!)

Die Zeichner-Klassen sind unabhängig voneinander, sie sind aber alle insofern ähnlich, dass ihre Instanzen mit einer Leinwand interagieren (= sie sind alle Zeichner-Instanzen). Man kann also die Menge von Klassen (für Linien, Kreise, Rechtecke usw.) kategorisieren bzw. klassifizieren. Dafür verwenden wir eine gemeinsame Basis-Klasse – alle Zeichner-Klassen sind dann von derselben Basisklasse abgeleitet.

Da die Zeichner-Klassen voneinander unabhängig sind (z.B. um einen Kreis zu zeichnen hilft es normalerweise nicht weiter, zu wissen, wie man eine gerade Linie zeichnet), ist ihre einzige Gemeinsamkeit die Anwendung ihrer Instanzen auf eine Leinwand. Dies kann man zum Ausdruck bringen, indem man der gemeinsamen Basisklasse nur eine einzige pure virtual member function verpasst (und sonst keine member).

Da die Zeichner-Klassen voneinander unabhängig sind (z.B. um einen Kreis zu zeichnen hilft es normalerweise nicht weiter, zu wissen, wie man eine gerade Linie zeichnet), ist ihre einzige Gemeinsamkeit die Anwendung ihrer Instanzen auf eine Leinwand. Dies kann man zum Ausdruck bringen, indem man der gemeinsamen Basisklasse nur eine einzige pure virtual member function verpasst (und sonst keine member).

Diese Funktion dient dann dazu, eine Zeichner-Instanz auf eine Leinwand anzuwenden, könnte also `applyTo` genannt werden und ein `Bitmap24&` als Parameter annehmen.

In einem nächsten Schritt (Workshop) soll von der Kommandozeile aus gezeichnet werden können. Es vereinfacht dabei die Benutzung, wenn man in einer Schrittabfolge arbeiten kann. So wählt man einen Pinsel, dann eine Position, und schließlich zeichnet man von dieser Position aus etwa eine Linie bis zu einem Endpunkt (3 Schritte). Eine zweite Linie zeichnet man dann in einem vierten Schritt durch die Angabe eines weiteren Endpunktes, die Linie geht dann vom Endpunkt der vorigen Linie bis zu dem neuen Endpunkt.

In einem nächsten Schritt (Workshop) soll von der Kommandozeile aus gezeichnet werden können. Es vereinfacht dabei die Benutzung, wenn man in einer Schrittabfolge arbeiten kann. So wählt man einen Pinsel, dann eine Position, und schließlich zeichnet man von dieser Position aus etwa eine Linie bis zu einem Endpunkt (3 Schritte). Eine zweite Linie zeichnet man dann in einem vierten Schritt durch die Angabe eines weiteren Endpunktes, die Linie geht dann vom Endpunkt der vorigen Linie bis zu dem neuen Endpunkt.

Diese aktuelle Position sowie eine aktuelle Farbe (der Pinsel) müssen dafür noch irgendwo gespeichert werden. Eine Möglichkeit hierfür ist, die Leinwand-Klasse zu erweitern. Man kann sich darunter dann statt einer Leinwand einen Tintenstrahldrucker oder Plotter vorstellen.

Die Zeichner-Instanzen, die also theoretisch auf einer Instanz eines `Bitmap24` arbeiten könnten, sollten – um die Benutzereingabe auf der Kommandozeile zu vereinfachen – nur so wenig Parameter wie nötig verwenden und den Rest aus vorhandenen Daten ziehen.

Die Zeichner-Instanzen, die also theoretisch auf einer Instanz eines `Bitmap24` arbeiten könnten, sollten – um die Benutzereingabe auf der Kommandozeile zu vereinfachen – nur so wenig Parameter wie nötig verwenden und den Rest aus vorhandenen Daten ziehen.

Die Klasse `BatchBitmap24` ist eine kleine Erweiterung von `Bitmap24`, welche die besprochene Farbe und eine Position speichert. Daher sollen die Zeichner-Instanzen zur Interaktion eine Instanz von `BatchBitmap24` verwenden statt eine Instanz von `Bitmap24`.

Die unsäglichen Koordinaten-Klassen

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativCoordinate` geschaffen.

Die unsäglichen Koordinaten-Klassen

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativCoordinate` geschaffen.

AbsoluteCoordinate

Fasst einfach den x- und y-Wert einer Koordinate zusammen in eine Datenstruktur. Der Beschützerinstinkt drückt sich dabei so aus, dass eine Instanz von `AbsoluteCoordinate` immer einer Leinwand zugeordnet ist. Das verhindert (soll verhindern), dass die Koordinate auf eine ungültige Position verweist (Lokalität von Fehlermeldungen).

Die unsäglichen Koordinaten-Klassen

Mit etwas zu viel Beschützerinstinkt habe ich wohl die beiden Klassen `AbsoluteCoordinate` und `RelativeCoordinate` geschaffen.

AbsoluteCoordinate

Fasst einfach den x- und y-Wert einer Koordinate zusammen in eine Datenstruktur. Der Beschützerinstinkt drückt sich dabei so aus, dass eine Instanz von `AbsoluteCoordinate` immer einer Leinwand zugeordnet ist. Das verhindert (soll verhindern), dass die Koordinate auf eine ungültige Position verweist (Lokalität von Fehlermeldungen).

RelativeCoordinate

Die `RelativeCoordinate` ist dann sozusagen die „freie Variante einer sicheren Koordinate“. Sie speichert eine Koordinate als Vielfaches von Breite bzw. Höhe, ihre Komponenten liegen also in $[0, 1]$ (dass die 1 eingeschlossen werden sollte, werde ich vielleicht einmal an anderer Stelle erläutern). Dadurch, dass sie unabhängig ist von einer konkreten Leinwand (auch logisch, da sie sich eben *nicht* auf eine feste Breite oder Höhe bezieht) eignet sie sich in meinen Augen wesentlich besser etwa um in einem Linien-Zeichenobjekt eine Endkoordinate zu speichern.

Dieses Interface dient nun als besprochene gemeinsame Basisklasse aller Zeichner-Klassen. Es enthält folgende pure virtual member function:

```
virtual bool applyTo( BatchBitmap24& ) = 0;
```

Der Rückgabewert soll hierbei true im Falle des erfolgreichen Zeichnens sein.

Dieses Interface dient nun als besprochene gemeinsame Basisklasse aller Zeichner-Klassen. Es enthält folgende pure virtual member function:

```
virtual bool applyTo( BatchBitmap24& ) = 0;
```

Der Rückgabewert soll hierbei true im Falle des erfolgreichen Zeichnens sein.

Die Zeichner-Klassen sind also von IBatchDrawable abgeleitet, und überschreiben die pure virtual member function. In der Überschreibung wird der Zeichen-Algorithmus aufgerufen und somit auf die Leinwand gezeichnet.

- Man kann über einen `IBatchDrawable*`-Pointer die Zeichen-Instanzen auf eine Instanz von `BatchBitmap24` anwenden, ohne Näheres über eine konkrete Zeichner-Klasse zu wissen. Mittels `dynamic storage duration` / `new` lassen sich somit zur Laufzeit verschiedene Zeichner-Instanzen erstellen (bspw. auf Anfrage des Benutzers!) und allgemein auf eine Leinwand anwenden.
- Die Klassifizierung aller Zeichner-Instanzen als „is-a“
`IBatchDrawable` erlaubt es, eine Menge von Zeichner-Instanzen gemeinsam zu „speichern“ – gemeint ist etwa ein Array von `IBatchDrawable*`-Pointern, oder der bekannte Ringpuffer mit `IBatchDrawable*` statt `double`.