Dr. U. Köthe, A. Alpay *Heidelberg, 25. Januar 2016*

Übungsblatt 12

Aufgabe 12.1 Transformation zwischen Koordinatensystemen [7 Punkte]

In Bildverarbeitung und Computergraphik muss man häufig zwischen den ganzzahligen Koordinaten der Pixel (dem *Pixelgitter*) und den reellwertigen (mathematischen) Koordinaten des zu lösenden Problems unterscheiden. In dieser Aufgabe wollen wir die Umrechnung zwischen den beiden Systemen üben.

(a) Wir bezeichen die reellwertigen Koordinaten mit (x, y), die Pixelkoordinaten mit `(ix, iy)`. Außerdem bezeichnen wir durch `(cx, cy)` die Koordinaten des Pixels in der Mitte des Bildes (bei Bildern mit geradzahliger Breite oder Höhe ist dies das Pixel rechts bzw. unterhalb der exakten Bildmitte) und die Bildgröße mit $W \times H$.

Die Umrechnung der Koordinatensysteme ist eindeutig definiert, indem wir die Seitenlänge d jedes (quadratischen) Pixels angeben, sowie für das mittlere Pixel die korrespondierenden reellwertigen Koordinaten $(x_{\text{center}}, y_{\text{center}})$. Geben Sie eine Rechenvorschrift an, um für jedes `(ix, iy)` das korrespondierende (x, y) zu berechnen.

(b) Diese Rechenvorschrift lässt sich nutzen, um eine zweidimensionale Funktion, die also von x und y abhängt, zu plotten. Schreiben Sie ein Programm, das zwei Bilder mit 640x480 Pixeln mithilfe der `Image`-Klasse (siehe `image.hpp` auf Moodle) erstellt. Initialisieren Sie die Pixel des einen Bildes mit der Funktion

$$f(x, y) = \left\lfloor 255 \cdot \left(\sin(x^2 + y^2)\right)^2 \right\rfloor$$

($\lfloor ... \rfloor$ bedeutet Abrunden) und der Pixelgröße $d = 1/64$. Das andere Bild initialisieren Sie mit der Funktion

$$f(x, y) = \left\lfloor 255 \cdot \left(\sin\left(\frac{x^2 + y^2}{4096}\right)\right)^2 \right\rfloor$$

und $d = 1$. In beiden Fällen soll $x_{\text{center}} = 0$ und $y_{\text{center}} = 0$ gelten. Wenn Ihre Koordinatenumrechnung korrekt ist, müssen beide Bilder gleich sein – testen Sie dies mittels `assert()`. Die Bilder sollten Ringe um die Bildmitte darstellen, die nach außen immer schmaler werden.

Geben Sie Ihre Lösung im File `koordinaten.cpp` ab (die Antwort zu Teilaufgabe (a) kann in einem Kommentar gegeben werden), sowie das Bild als `koordinaten.pgm`.

Aufgabe 12.2 Visualisierung des Mandelbrotfraktals [24 Punkte]

In dieser Übungsaufgabe geht es darum, die `Image`-Klasse aus `image.hpp` zu verwenden, um ein Programm zu schreiben, das sogenannte Mandelbrotfraktale berechnet und als Bild

visualisiert. Diese Fraktale sind ein gutes Beispiel dafür, dass schon sehr einfache mathematische Regeln zu einem Verhalten führen können, das sich nicht mehr analytisch beschreiben lässt. Daher spielen Mandelbrotfraktale eine bedeutende Rolle in der Chaosforschung. Aber auch praktische Anwendungen existieren, beispielsweise im Zusammenhang mit Bildkompression. Noch berühmter sind sie jedoch für die Schönheit ihrer Visualisierungen, zusammen mit der Eigenschaft, dass man (theoretisch) beliebig weit hineinzoomen kann und ständig neue fantastische Details hervortreten. Weitere Informationen finden Sie auf Wikipedia unter `https://de.wikipedia.org/wiki/Mandelbrot-Menge`. Einen extremen Zoom können Sie sich unter `https://youtu.be/zXTpASSd9xE` ansehen.

Die Grundlage für Mandelbrotfraktale bilden die komplexen Zahlen, wobei es sich um eine Erweiterung der reellen Zahlen handelt. Jede komplexe Zahl z besteht aus einem Realteil a und einem Imaginärteil ib, so dass $z = a + \mathrm{i}b \in \mathbb{C}$ (mit $a, b \in \mathbb{R}$). i ist die *imaginäre Einheit*, die so definiert ist, dass $\mathrm{i}^2 = -1$ gilt (mehr Informationen finden Sie auf Wikipedia unter `https://de.wikipedia.org/wiki/Komplexe_Zahl`).

Im Zentrum der Untersuchung von Mandelbrotfraktalen steht die komplexwertige Folge

$$z_{n+1} = z_n^2 + c \tag{1}$$

für $z_n, z_{n+1}, c \in \mathbb{C}$. Als Startwert der Folge verwendet man $z_0 = 0 + 0\mathrm{i}$. Das besondere an dieser Folge ist, dass ihr Divergenzverhalten in Abhängigkeit von der komplexen Zahl c höchst kompliziert ist. Die typischen Mandelbrotbilder erhält man, indem man in jedem Pixel die *Divergenzgeschwindigkeit* der Folge visualisiert. Die Divergenzgeschwindigeit gibt an, wie schnell der Betrag $|z_n|$ groß wird, und dies variiert in Abhängigkeit von c sehr stark. Am einfachsten lässt sich die Geschwindigkeit messen, indem man zählt, wieviele Iterationen (also welches n) man braucht, bis $|z_n|^2$ größer als eine feste Schranke wird, z.B. 1000.

Die Kunst besteht darin, $c = x + \mathrm{i}y$ so zu wählen, dass interessante Muster entstehen. Man definiert dazu wie in Aufgabe 12.1 eine Umrechnung von den Pixelkoordinaten `ix` und `iy` in die reellwertigen Koordinaten x und y, indem man x_{center}, y_{center} und d geschickt festlegt.

Hinweise: Diese Übung ist rechenintensiv. Aktivieren Sie mit dem Kommandozeilenargument `-O2` (gcc/MinGW) bzw. `/O2` (Visual Studio) die Compileroptimierungen, um die Rechenzeit Ihres Programms zu verkürzen (vergleiche Übung 7.2). Benutzen Sie den Typ `std::complex<double>` aus dem Header `<complex>`. Dieser Datentyp stellt die üblichen arithmetischen Operatoren `+`, `-`, `*`, `/` bereit, sodass Sie komplexwertige Rechnungen mit derselben Syntax wie bei normalen `double`-Werten durchführen können.

Geben Sie Ihre Lösung in der Datei `mandelbrot.cpp` ab.

(a) Definieren Sie die Klasse `FractalView` mit den (privaten) Membervariablen vom Typ `double` `x_center_`, `y_center_` und `pixel_side_`, um die Bildposition und Pixelgröße zu speichern. Diese Attribute bestimmen (wie in Aufgabe 12.1) die Umrechnung zwischen Pixel- und reellwertigen Koordinaten und damit den Ausschnitt des Fraktals, den wir berechnen. Definieren Sie außerdem einen Konstruktor, über den diese drei Attribute initialisiert werden können.

(b) Implementieren Sie eine private Memberfunktion

```
int iterations_until_limit(std::complex<double> const& c) const
```

die zählt, wie viele Iterationen nötig sind, bis das Betragsquadrat der Folgenglieder einen bestimmten Wert (der in der Funktion fest verdrahtet sein kann, z.B. 1000)

überschreitet, wenn man das gegebene `c` in Formel (1) einsetzt. Das Betragsquadrat einer komplexen Zahl z können Sie mit `std::norm(z)` bestimmen. Rechnen Sie *nicht* den Betrag selbst aus (das macht `std::abs(z)`), da dafür zusätzlich die Berechnung einer Wurzel erforderlich ist, was das Programm langsamer macht.

Sie müssen die Schleife außerdem nach einer maximalen Anzahl von Iterationen abbrechen, da es auch Werte für c gibt, bei denen die Folge konvergiert und somit niemals größer als die Schranke wird. Ohne zusätzliche Abbruchbedingung bekommen Sie dann eine Endlosschleife. Ein guter Wert für die maximale Iterationszahl ist immer ein Kompromiss: Ist er zu niedrig, fehlen vor allem in hohen Zoomstufen Details; bei zu großen Werten wird die Berechnung langsam. Wir empfehlen den Wert 512.

(c) Implementieren Sie eine private Memberfunktion

```
std::complex<double>
pixel_to_complex(Image const & img, int ix, int iy) const
```

die mit Hilfe der Membervariablen `x_center_`, `y_center_` und `pixel_side_` die Pixelkoordinaten `ix`, `iy` zuerst in reelle Zahlen x, y und dann in die komplexe Zahl $c = x + \mathrm{i}y$ umrechnet. (Man stellt sich komplexe Zahlen also als zweidimensionale Punkte vor, deren x und y-Koordinaten dem Real- und Imaginärteil entsprechen – die sogenannte *komplexe Zahlenebene*.)

(d) Implementieren Sie eine private Memberfunktion

```
uint16_t color_scheme(int num_iterations) const
```

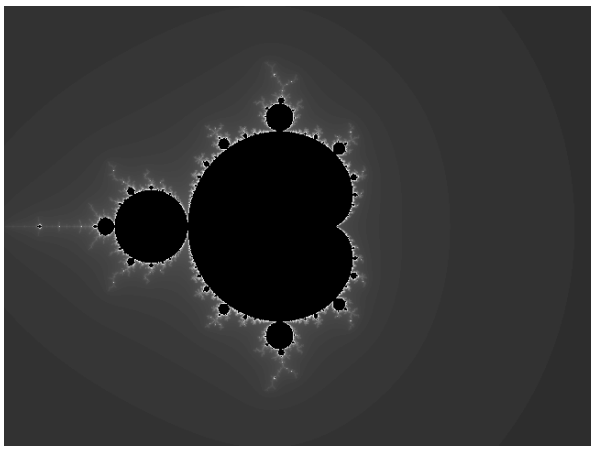
die für jede Anzahl von Iterationen einen Grauwert im Bereich 0 bis 255 zurückgibt. Berechnen Sie dazu die Kubikwurzel der Anzahl mit passender Skalierung. Um das klassische Mandelbrot-Bild zu erhalten, müssen die konvergenten Bereiche des Bildes (die Pixel, wo `num_iterations` den Maximalwert erreicht hat) explizit schwarz, also mit dem Grauwert 0, eingefärbt werden.

(e) Implementieren Sie eine öffentliche Memberfunktion

```
void render_mandelbrot(Image & img) const
```

die für jedes Pixel des übergebenen Bildes die Anzahl der Iterationen bestimmt und dem Pixel darauf basierend einen Grauwert zuweist.

(f) Initialisieren Sie in `main()` ein `FractalView`-Objekt mit `x_center_ = 0`, `y_center_ = 0`. Wählen Sie `pixel_side_` so, dass die Bildbreite im reellwertigen System gleich 4.0 ist. Rendern Sie das Fraktal in ein `Image` der Größe 640x480 und geben Sie dieses Bild als `mandelbrot_overview.pgm` ab. Es sollte dem folgenden Bild ähneln (das Fraktal wird deshalb auch als “Apfelmännchen” bezeichnet):



(g) Wir wollen dem Nutzer nun die Möglichkeit geben, durch das Bild zu navigieren, um besonders interessante Bereiche des Fraktals zu finden und darzustellen. Hierzu benötigen wir ein Benutzerinterface, das in einer Endlosschleife eine Eingabe vom Benutzer einliest, die den Bildausschnitt verschiebt oder zoomt. Danach soll das Fraktal neu berechnet und als Bild unter dem Namen `mandelbrot.pgm` gespeichert werden. Falls die Bilddatei bereits existiert, soll sie überschrieben werden. Einige Bildbetrachter-Programme merken, wenn die gerade betrachtete Datei geändert wurde und laden das Bild automatisch neu. Dann werden die Benutzereingaben bei gleichzeitig geöffnetem Bildbetrachter wirklich interaktiv. Ist Ihr Bildbetrachter nicht so schlau, müssen Sie das Bild jedesmal händisch aktualisieren.

Schreiben Sie ein Programm, das folgende Benutzereingaben unterstützt:

- Bei der Eingabe von 'a' soll sich der Bildausschnitt um 1/4 der Bildbreite nach links bewegen,
- bei der Eingabe von 'd' um 1/4 der Bildbreite nach rechts,
- bei der Eingabe von 'w' um 1/4 der Bildhöhe nach oben,
- bei der Eingabe von 's' um 1/4 der Bildhöhe nach unten.
- Bei der Eingabe von '+' soll um den Faktor 2 hineingezoomt werden.
- Bei der Eingabe von '-' soll um den Faktor 2 herausgezoomt werden.
- Bei der Eingabe von 'b' soll die Eingabeschleife beendet werden.

Implementieren Sie dafür geeignete Setter-Funktionen in der Klasse `FractalView`. Geben Sie nach jeder Änderung die aktuellen Werte von `x_center_ = 0`, `y_center_ = 0` und `pixel_side_` aus, so dass man später ein `FractalView`-Objekt direkt für diesen Ausschnitt initialisieren kann. Verwenden Sie ein Bild der Größe 640x480 Pixel und beginnen Sie mit den gleichen Einstellungen wie in (f).

(h) In manchen Regionen des Fraktals ändert sich die Anzahl der Iterationen drastisch von Pixel zu Pixel, und das Bild wirkt dadurch verrauscht. Man kann dies durch sogenanntes *Anti-Aliasing* beheben: Wird die Anzahl der Iterationen im Pixel `(ix, iy)` abgefragt, verwendet man stattdessen die vier *Subpixel-Koordinaten* $\left(\texttt{ix} \pm \frac{1}{4}, \texttt{iy} \pm \frac{1}{4}\right)$ and gibt den Mittelwert dieser Ergebnisse zurück. Implementieren Sie dieses Verhalten (Tipp: dazu müssen Sie an einigen Stellen `int` durch `double` ersetzen).

(i) Man kann die Bilder noch verschönern, indem man die Anzahl der Iterationen geschickter in Grauwerte umwandelt, z.B. mit Hilfe einer periodischen Funktion. Experimentieren Sie mit verschiedenen Transformationen, bis Sie mit den Ergebnissen zufrieden sind, und modifizieren Sie die Funktion `color_scheme()` entsprechend.

(j) Initialisieren Sie am Ende von `main()` ein `FractalView`-Objekt für einen interessanten Ausschnitt des Fraktals, rendern und speichern Sie das Bild und geben Sie es als `mandelbrot_zoom.pgm` ab.

(k) Wie weit können Sie in das Bild (sinnvoll) hineinzoomen? Es genügt hier eine grobe Angabe der Größenordnung des Pixeldurchmessers. Welche Probleme treten bei zu starkem Zoom auf, und was ist der Grund für diese Probleme?

Bonusaufgabe: Ein anderes schönes Fraktal, eine Variante der sogenannten *Julia-Mengen*, erhalten Sie, wenn Sie Formel (1) mit $z_0 = x + \mathrm{i}y$ initialisieren und c als ortsunabhängige Konstante definieren, siehe `https://de.wikipedia.org/wiki/Julia-Menge`. Implementieren Sie diese Variante und fügen Sie geeignete Benutzereingaben hinzu, um günstige Werte für c zu finden.

Aufgabe 12.3 Farbdarstellung des Mandelbrotfraktals [9 Punkte]

Mandelbrotfraktale sehen natürlich in Farbe wesentlich besser aus. Sie sollen hier eine Farbversion von Aufgabe 12.2 implementieren und in der Datei `mandelbrot_color.cpp` abgeben.

Für Farbbilder wird üblicherweise die *RGB*-Farbdarstellung verwendet, bei der jedes Pixel seinen Rot-, Grün- und Blauanteil speichert. Auf Moodle finden Sie die Datei `image_rgb.hpp`, die dafür eine Klasse `RGBColor`, das Template `Image<RGBColor>` sowie die Import/Export-Funktionen `readPPM()` und `writePPM()` enthalten (das PPM-Format ist die Farbversion des PGM-Formats).

(a) Kopieren Sie den Inhalt von `mandelbrot.cpp` aus Aufgabe 12.2 in die neue Datei `mandelbrot_color.cpp`. Ersetzen Sie dann `#include "image.hpp"` durch `#include "image_rgb.hpp"` und führen Sie die notwendigen Folgeänderungen durch. Unter anderem muss die Memberfunktion `color_scheme` jetzt ein Ergebnis vom Typ `RGBColor` zurückgeben. Setzen Sie dafür die drei Farbkanäle auf den Grauwert, der in der Originalversion der Funktion zurückgegeben wurde (bei gleichem Rot-, Grün- und Blauanteil bekommt man wieder Grau).

(b) Initialisieren Sie in `main()` ein `FractalView`-Object mit den Grundeinstellungen für den Apfelmännchen-Ausschnitt, rendern Sie ein Farbbild der Größe 640x480 und speichern Sie es ab. Dieser Schritt dient der Kontrolle – das Ergebnis muss genauso aussehen wie das entsprechende Graubild aus Aufgabe 12.2.

(c) Um das Bild farbig zu machen, müssen wir die Anzahl der Iterationen jetzt in Farben umrechnen. Dafür eignet sich am besten die *HSV*-Farbdarstellung. Hierbei wird die Farbe nicht durch Rot-, Grün-, und Blauanteil dargestellt, sondern durch *Hue* (Farbton), *Saturation* (die Kräftigkeit der Farbe) und *Value* (die Helligkeit). Dabei liegt Hue zwischen 0 und 360 Grad (entspricht den Regenbogenfarben von Rot über Gelb, Grün und Blau bis Purpur), Saturation zwischen 0 und 1 (von farblos zur reinen Farbe) und Value ebenfalls zwischen 0 und 1 (von dunkel nach hell). Nähere Informationen finden Sie in der Wikipedia unter `https://de.wikipedia.org/wiki/HSV-Farbraum`. Die Datei `image_rgb.hpp` enthält eine Funktion `RGBColor hsv_to_rgb(double H, double S, double V)`, die aus einem gegebenen HSV-Wert den zugehörigen RGB-Wert berechnet.

Experimentieren Sie mit verschiedenen Rechenvorschriften für die Umwandlung, bis Sie mit dem Ergebnis zufrieden sind, und ändern Sie die Funktion `color_scheme` entsprechend. Um ein buntes Bild zu erhalten, müssen Sie vor allem H variieren und dürfen S nicht zu klein wählen.

(d) Generieren Sie jetzt eine Farbversion des Apfelmännchen-Bildes (mit den Grundeinstellungen von `FractalView`, Größe 640x480) und geben Sie es als `mandelbrot_color_overview.ppm` ab. Suchen Sie dann mit den Tastenkombinationen einen schönen Ausschnitt des Fraktals und geben Sie das resultierende Bild als `mandelbrot_color_zoom.ppm` ab.

Bitte laden Sie Ihre Lösung spätestens bis 1. Februar 2017, 9:00 Uhr in Moodle hoch.