Programmierkurs
Dr. Steffen Müthing
IWR, Universität Heidelberg

Aufgabenblatt 7
Abgabe 10. Dezember 2018
ipk-exercises:2018-ws-34-gafbb096

Aufgabenblatt 7

Allgemeine Hinweise:

- Für die Aufgaben auf diesem Übungsblatt müssen Sie am 10.12. votieren.
- Lesen Sie sich die Aufgaben jeweils vollständig durch, bevor Sie anfangen, sie zu bearbeiten!
- **Wichtig:** Stellen Sie sicher, dass die Signatur Ihrer Funktion genau damit übereinstimmt, was auf dem Blatt angegeben ist (Typen von Paramtern und Rückgabewerten)!

Aufgabe 1

Flächenberechnung für Polygone

(a): 1 Punkt, (b): 1 Punkt, (c): 1 Punkt

Ein Polygon ist eine geometrische Figur, die aus einem geschlossenen stückweise linearen Streckenzug besteht. Dazu werden n Punkte $p_i = (x_i, y_i), i \in \{0, \dots, n-1\}$ in der Ebene $\mathbb{R}^2$ definiert und jeweils eine gerade Linie von p_i zu p_{i+1} gezogen. Geschlossen wird die Kurve, indem man abschließend eine Linie von p_{n-1} zu $p_n := p_0$ zieht. Das zweidimensionale Volumen (d.h. die Fläche) eines solchen Polygons ist gegeben durch

$$A = \frac{1}{2} \cdot \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

mit dem letzen Punkt p_n wie oben definiert. Wenn die Punkte im mathematisch positiven Sinne (gegen den Uhrzeigersinn sortiert) angegeben sind, entspricht das der üblichen Vorstellung eines Flächenmaßes, bei mathematisch negativer Angabe (im Uhrzeigersinn) ist A negativ.

Schreiben Sie ein Programm, das das Volumen eines gegebenen Polygons berechnet. Gehen Sie dabei wie folgt vor:

(a) Schreiben Sie einen Header `point.hh` und eine Implementierung `point.cc` mit einer Klasse namens `Point`, die die Koordinaten eines Punktes in der Ebene speichern kann (zwei `double`-Werte). Hierbei ist folgendes zu beachten:

- Die Member-Variablen sollen `private` sein.
- Es soll einen Default-Konstruktor geben, der den Punkt $(0, 0)$ erzeugt.
- Es soll einen Konstruktor geben, der die x- und y-Koordinaten als zwei Parameter übergeben bekommt.
- Die Klasse soll Accessor-Methoden

```
1 double x() const;
2 double y() const;
```

bereitstellen.

(b) Schreiben Sie einen Header `polygon.hh` und eine Implementierung `polygon.cc` mit einer Klasse `Polygon`, die eine Liste (also einen `std::vector`) von `Point`s enthält.

- Sämtliche Member-Variablen der Klasse sollen `private` sein.
- Schreiben Sie einen Konstruktor mit folgender Signatur:

```
Polygon(const std::vector<Point>& corners)
```

Dieser Konstruktor soll die interne Liste von Ecken aus der gegebenen Liste von Punkten initialisieren. Sie können hierbei davon ausgehen, dass die Punkte korrekt gegen den Uhrzeigersinn sortiert sind. Der Konstruktor soll alle Arbeit in der constructor initializer list machen.

- Schreiben Sie einen Konstruktor mit folgender Signatur:

```
Polygon(const std::vector<double>& x,
        const std::vector<double>& y
       )
```

Dieser Konstruktor soll die Ecken aus den beiden gegebenen Listen von Koordinaten für x und y initialisieren. Dies muss im Body des Konstruktors passieren. Auch hier müssen Sie die Ecken nicht auf korrekte Sortierung prüfen.

- Schreiben Sie eine Methode `std::size_t corners() const`, die die Anzahl an Ecken zurückgibt.
- Schreiben Sie eine Methode `const Point& corner(std::size_t i) const`, die die i-te Ecke des Polygons zurückgibt.

(c) Schreiben Sie eine Methode `double volume() const`, die das Volumen des Polygons mit der obigen Formel bestimmt. Stellen Sie sicher, dass diese Funktion auch für "Polygone" mit 0, 1 oder 2 Ecken funktioniert, wobei wir in diesen Fällen $A := 0$ setzen.

Testen Sie Ihre Implementierung, indem Sie die Volumina der ersten 10 regelmäßigen n-Ecke berechnen. Die i-te Koordinate eines regelmäßigen n-Ecks ist durch die folgende Formel gegeben:

$$p_i = \left(\cos\left(\frac{i}{n}\cdot 2\pi\right), \sin\left(\frac{i}{n}\cdot 2\pi\right)\right).$$

Die trigonometrischen Funktionen (`std::sin(double)` etc.) sind in der Standardbibliothek im Header `cmath` definiert, π müssen Sie als Konstante selbst definieren:

```
const double pi = M_PI;
```

Für solche regelmäßigen n-Ecke gilt

$$A = \frac{n}{2}\cdot \sin\left(\frac{2\pi}{n}\right),$$

und damit für eine Auswahl der ersten n-Ecke:

Ecken	1	2	3	4	6	8
Fläche	0	0	$\frac{3}{4}\cdot 3^{1/2}$	2	$\frac{3}{2}\cdot 3^{1/2}$	$2\cdot 2^{1/2}$

Aufgabe 2

Pixelgraphiken

(b-f): 1 Punkt, (g): 1 Punkt, (h): 1 Punkt

Graphiken werden in Computern auf zwei grundlegend verschiedene Weisen behandelt: mit Vektorgraphiken, die Bilder abstrakt durch Kurvensegmente und geometrische Figuren darstellen, und mit Pixel- bzw. Rastergraphiken, die Bilder als ein diskretes zweidimensionales Gitter aus kleinen Bildpunkten, sogenannten Pixeln, behandeln. In dieser Aufgabe wollen wir uns mit dem letztgenannten Ansatz näher beschäftigen.

Hinweise:

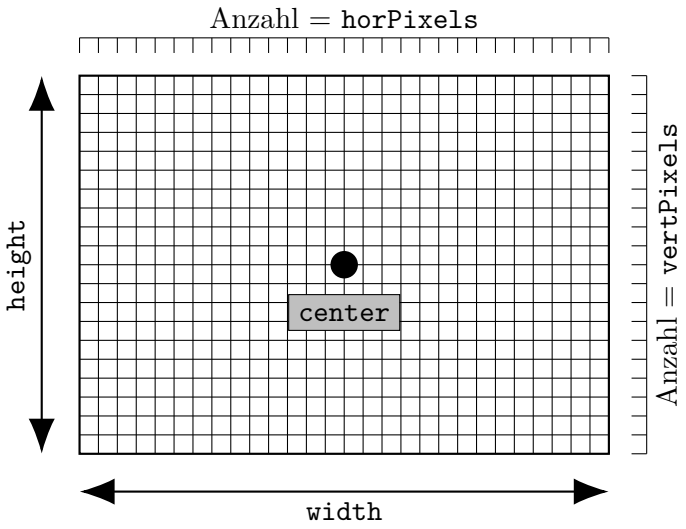
- Erstellen Sie für die Aufgabe ein CMake-Buildsystem, das die einzelnen Programme baut.
- Verwenden Sie in allen Konstruktoren eine constructor initializer list.

(a) Verwenden Sie zum Repräsentieren der Pixel-Koordinaten die Klasse `Point` aus der vorherigen Aufgabe.

(b) Schreiben Sie einen Header `canvas.hh` und eine Implementierung `canvas.cc` mit einer Klasse `Canvas` (Leinwand), die die Grauwerte der Pixel eines Bildes mit zugehöriger Geometrie-Information darstellt. Dies stellen wir durch folgende Informationen (Member-Variablen) dar:

- Eine Anzahl an Pixeln in x- und in y-Richtung.
- Einen `Point`, der die Koordinate der Bildmitte angibt.
- Zwei `double`-Werten für die physikalische Höhe und Breite des Bildes.

Das ist auf folgendem Bild noch einmal illustriert:



Dazu soll die Klasse ein zweidimensionales Array zum Speichern von Grauwerten für jeden Pixel enthalten. Dieses implementieren Sie als `std::vector<std::vector<int> >`. Bis auf das Pixel-Array sollen die Member-Variablen `const` sein, damit sie vor unbeabsichtigten Änderungen geschützt sind.

(c) Einen Konstruktor

```
1 Canvas(const Point& center, double width, double height,
2         int horPixels, int vertPixels);
```

der mit `center` als Zentrum ein `Canvas` mit `horPixels` pro Zeile und `vertPixels` pro Spalte konstruiert.

(d) Eine Methode `int brightness(int i, int j) const`, die die Helligkeit des Pixels mit den Koordinaten (i, j) zurückgibt.

(e) Eine Methode `void setBrightness(int i, int j, int brightness)`, die die Helligkeit des Pixels mit den Koordinaten (i, j) auf `brightness` setzt.

(f) Eine Methode `Point coord(int i, int j) const`, die mithilfe der gespeicherten Geometrie-Informationen die geometrischen Koordinaten des Punktes berechnet, der dem Pixel in der i-ten Spalte und j-ten Zeile (von links unten aus gesehen) entspricht. Dabei dürfen Sie der Einfachheit halber die linke untere Ecke des Pixels als Punkt ausgeben statt der Mitte des Pixels.

(g) Testen Sie die obige Funktionalität ausgiebig, bevor Sie den Rest der Aufgabe bearbeiten. Gerade beim Berechnen der Koordinaten können leicht Fehler unterlaufen, die man ohne Testen nicht unbedingt sofort bemerkt. Prüfen Sie zumindest die korrekte Berechnung der vier Ecken, die bis auf die Breite eines Pixels mit folgendem übereinstimmen sollten:

- Linke untere Ecke: $i = 0$, $j = 0$, $x = x_{\text{center}} - \frac{1}{2}\text{width}$, $y = y_{\text{center}} - \frac{1}{2}\text{height}$
- Rechte untere Ecke: $i = \text{horPixels} - 1$, $j = 0$, $x = x_{\text{center}} + \frac{1}{2}\text{width}$, $y = y_{\text{center}} - \frac{1}{2}\text{height}$
- Linke obere Ecke: $i = 0$, $j = \text{vertPixels} - 1$, $x = x_{\text{center}} - \frac{1}{2}\text{width}$, $y = y_{\text{center}} + \frac{1}{2}\text{height}$

- Rechte obere Ecke: $i = \text{horPixels} - 1$, $j = \text{vertPixels} - 1$, $x = x_{\text{center}} + \frac{1}{2}\text{width}$, $y = y_{\text{center}} + \frac{1}{2}\text{height}$

Schreiben Sie hierfür ein Test-Programm `testcorners`, das sich an die in der Vorlesung vorgestellten Konventionen für Tests hält, und fügen Sie diesen Test zum CMake-Buildsystem hinzu, so dass er durch einen Aufruf von `ctest` ausgeführt wird.

(h) Nachdem Sie sichergestellt haben, dass Ihre Methode die korrekten Koordinaten liefert, schreiben Sie eine Datei "plot.cc", mit einer `main()`-Funktion, die einen um den Ursprung zentrierten `Canvas` der Breite 4 und Höhe 3 erzeugt, der 4000 Pixel in der Horizontalen und 3000 in der Vertikalen verwendet. Tragen Sie dann die Werte der folgenden Funktion als Grauwerte ein:

$$f(x, y) := \max(0, 100 \cdot (\sin(x^{-1}) \cdot \sin(y^{-1}) + 1))$$

Benutzen Sie die bereitgestellte Headerdatei `pgm.hh` und Implementierung `pgm.cc`, um die Pixel in eine Bilddatei zu schreiben, und prüfen Sie, dass das Ergebnis Ihren Erwartungen entspricht.

Fügen Sie hierzu der Klasse `Canvas` eine neue Methode `write(const std::string& filename)` hinzu. Das PGM-Bild benötigt keine Informationen über die tatsächliche Grösse oder den Ursprung des Bildes.

Hinweise:

- Um das Pixel-Array zu initialisieren, können Sie folgenden Ausdruck verwenden (angenommen, die Variable mit dem Array heisst `_pixels`):

```
1 _pixels(horPixels,{{vertPixels}})
```

Dabei konstruiert die Anweisung mit den geschweiften Klammern einen Vektor der Länge `vertPixels`, der dann einmal für jede Spalte im Array (`horPixels` mal) kopiert wird.

- Die max-Funktion in der Formel oben soll sicherstellen, dass die Koordinatenachsen keine Auswertungsprobleme verursachen. Die Addition von 1 und Skalierung dienen dazu, Grauwerte im positiven Bereich mit genügend Abstufungen zu erzeugen. Das könnte man mit etwas zusätzlichem Aufwand auch automatisiert erledigen (siehe Fortgeschrittenen-Aufgabe).
- Das erzeugte Bild können Sie sich im Dateimanager anschauen.