UNIVERSITÄT DUISBURG ESSEN

NES Networked Embedded Systems

Miniprojekt 1: Kontrollstrukturen und Bildverarbeitung

Aufgabe 1: Kontrollstrukturen

In dieser Aufgabe geht es um die Nutzung von Kontrollstrukturen zur Umsetzung einfacher logischer und arithmetischer Berechnungen. Die Klasse mit dem Namen *Aufgabe1* enthält hierfür mehrere statische Methoden, die Sie wie folgt implementieren sollen:

1. **boolean decideWithIfOneTwoOrThree(int number)**
   Diese Methode soll den Wert „wahr“ zurückgeben, wenn die als „number“ übergebene Zahl den Wert 1, 2 oder 3 hat. In allen anderen Fällen soll der Wert „falsch“ zurückgegeben werden. Für die Implementierung sollen Sie ein oder mehrere „if“ Statements nutzen.
2. **public static boolean decideWithCaseOneTwoOrThree(int number)**
   Analog zur Aufgabe 1.1. soll diese Methode ebenfalls den Wert „wahr“ für die Eingaben 1, 2 oder 3 und „falsch“ für alle anderen Eingaben zurückgeben. Für die Implementierung dieser Methode sollen Sie jedoch ein oder mehrere „switch“ Statements nutzen.
3. **public static boolean decideWithoutCaseOrIfOneTwoOrThree(int number)**
   Analog zu den Aufgaben 1.1. und 1.2. soll diese Methode ebenfalls den Wert „wahr“ für die Eingaben 1, 2 oder 3 und „falsch“ für alle anderen Eingaben zurückgeben. Für die Implementierung dieser Methode sollen sie jedoch ausschließlich einen einfachen logischen Ausdruck (also insbesondere kein „if“ und kein „switch“ Statement) verwenden.

Überlegen Sie sich die Antworten auf folgende Fragen (falls Sie sich nicht sicher sind, diskutieren Sie sie mit Ihren Kommilitonen oder besuchen Sie eine Fragestunde):

- Ist es möglich beliebige „if“ Statements in „switch“ Statements umzuwandeln? Falls die richtige Antwort nein lautet, welche Bedingungen müssen für eine Umwandlung erfüllt sein?
- Warum war es in Aufgabe 1.3. möglich den Test auf die Zahlen 1, 2 oder 3 auch ganz ohne die Nutzung eines „if“ oder „switch“ Statements umzusetzen?

4. **public static void fillArrayIndexWithFor(int[] array)**
   Diese Methode soll das als Parameter mit dem Namen „array“ übergebene int-Array durchlaufen und dabei für jede Stelle i den Wert dieser Stelle auf i setzen. Beispiel: Der Inhalt eines beliebigen int-Arrays der Länge 4 soll nach Ausführung der Methode [0, 1, 2, 3] sein. Zur Implementierung dieser Methode sollen Sie ein „for“ Statement verwenden.
5. **public static void fillArrayIndexWithWhile(int[] array)**
   Analog zu Aufgabe 1.4 soll diese Methode die Stellen des Arrays mit dem jeweiligen Index befüllen. Ersetzen Sie hierbei jedoch Ihr „for“ Statement durch eine entsprechende „while“ Schleife.

6. **public static void fillArrayReverseIndexWithFor(int[] array)**
   Wie in Aufgabe 1.4. und 1.5. soll auch diese Methode ein int-Array mit Zahlen befüllen. Die Reihenfolge der Zahlen soll jetzt jedoch umgedreht werden. Beispiel: Der Inhalt eines beliebigen int-Arrays der Länge 5 soll nach Ausführung der Methode [4, 3, 2, 1, 0] sein. Nutzen Sie für die Implementierung dieser Methode eine „for“ Schleife.
   Hinweis: Falls Sie nicht wissen, wo Sie anfangen sollen, überlegen Sie sich zunächst eine allgemeingültige Formel, mit der Sie den Wert an der Stelle i in Abhängigkeit der Länge des Arrays berechnen können. Falls Ihnen auch das schwer fällt, schreiben Sie sich zunächst die gewünschten Ausgaben für Arrays verschiedener Längen auf (also z.B. [3, 2, 1, 0] für Arrays mit 4 Elementen, [2, 1, 0] für Arrays mit 3 Elementen, usw.). Dann überlegen Sie sich, um welchen Betrag sich nebeneinanderliegende Werte unterscheiden. Wenn Sie das gemacht haben, sollte Ihnen klar sein, dass Ihre Schleife ausgehend von einem der Länge entsprechenden Startwert in jedem Schritt einen gewissen Betrag abziehen muss, damit Sie den Wert der nächsten Stelle im Array bekommen.
7. **public static void fillArrayReverseIndexWithWhile(int[] array)**
   Diese Methode soll die selbe Funktion wie Aufgabe 1.6. umsetzen. Zur Implementierung sollen Sie jetzt jedoch Ihre „for“ Schleife durch eine entsprechende „while“ Schleife ersetzen.
8. **public static int computeArraySumWithFor(int[] array)**
   Diese Methode soll die Summe aller im Array enthaltenen Zahlen bilden und diese zurückgeben. Das Array soll dabei nur gelesen und nicht verändert werden. Für die Implementierung sollen Sie eine „for“ Schleife verwenden. Beispiel: Die Rückgabe für das Array [1, 2, 3] soll 6 sein. Die Rückgabe für das Array [3, 17, - 5] soll 15 sein.
   Hinweis: Wenn Sie nicht wissen, wo Sie ansetzen sollen, nehmen Sie sich einen Stift und ein Blatt Papier und Summieren Sie von Hand ein paar Beispielarrays – Schritt für Schritt – und versuchen Sie danach Ihr Vorgehen zu verallgemeinern. Also z.B.: Eingabe: [2, 5, 8] – Schritt 1: Summe: 0, Schritt 2: Summe 0+2=2, Schritt 2: Summe 2+5=7, Schritt 3: Summe 7+8=15, Fertig: Ausgabe 15.
9. **public static int computeArraySumWithWhile(int[] array)**
   Diese Methode soll die selbe Funktion wie Aufgabe 1.8 umsetzen. Zur Implementierung sollen Sie jetzt jedoch (ähnlich wie in Aufgabe 1.7.) Ihre „for“ Schleife durch eine „while“ Schleife ersetzen.

Überlegen Sie sich die Antworten auf folgende Fragen (falls Sie sich nicht sicher sind, diskutieren Sie sie mit Ihren Kommilitonen oder besuchen Sie eine Fragestunde):

- Welche Schritte sind üblicherweise notwendig um eine „for“ Schleife in eine „while“ Schleife umzuwandeln? Ist die Umwandlung immer möglich? Falls die richtige Antwort ja lautet, warum gibt es dann überhaupt „for“ Schleifen?

Um Ihre Lösungen zu testen, verwenden Sie die Ausgabe der main-Methode.

Aufgabe 2: Bildverarbeitung

Mit dieser Aufgabe üben Sie die Nutzung von mehrdimensionalen Arrays und Schleifen an einem praktischen Beispiel – der Bildverarbeitung. Lassen Sie sich dabei bitte nicht von der Anwendung und den (für Sie vielleicht neuen) Begriffen ablenken oder gar entmutigen. Um diese Aufgabe zu lösen brauchen Sie kein fortgeschrittenes Verständnis der Informatik. Sie müssen sich nur den Aufgabentext **genau durchlesen**, ihn verstehen und die geforderte Programmlogik umsetzen. Die Programmlogik besteht aus einfachen Schleifen und Verzweigungen und ist nicht wesentlich komplizierter (aber erheblich cooler) als Aufgabe 1.

Hintergrund: Der Grund für die Wahl der Bildverarbeitung als praktisches Beispiel ist die visuelle Natur des Anwendungsbereichs. Wenn Sie sich nicht sicher sind ob ihr Programm richtig ist, können Sie sich einfach die Ausgabe mit einem Bildbetrachtungsprogramm anschauen.

Einführung

Zu dieser Aufgabe gehören 3 Klassen: *Aufgabe2*, *Colors* und *Images*, sowie ein Beispielbild „notre-dame.png“. Kopieren Sie die Klassen in den src-Ordner Ihres Eclipse Projekts und kopieren Sie das Bild in den darüberliegenden Ordner (also direkt in den Projektordner). Die Klasse *Aufgabe2* enthält eine main-Methode, die das Bild lädt, die von Ihnen zu programmierenden Methoden ausführt und die Ergebnisse als neue Bilder speichert. Die generierten Bilder liegen im selben Ordner wie „notre-dame.png“. Damit Sie sie in Eclipse sehen, müssen sie nach Programmende ggfs. „Refresh“ (Rechtsklick auf Ihr Projekt) auswählen oder Sie öffnen den Ordner in einem Dateiexplorer (empfohlen). Bitte beachten Sie, dass jede Ausführung der main-Methode neue Bilder mit einem Zeitstempel im Dateinamen im Projektordner erzeugt. Um die Übersicht zu behalten, löschen Sie diese regelmäßig, aber passen Sie auf, dass Sie nicht das Quellbild „notre-dame.png“ löschen.

Die Klasse *Images* enthält statische Methoden zum Laden und Speichern von Bildern im PNG-Format. Nach dem Laden wird ein Bild als 2-dimensionales int-Array in Java repräsentiert. Jeder Wert in diesem Array repräsentiert den Farbwert eines bestimmten Pixels des Bilds. Die erste Dimension des Arrays gibt dabei die Position auf der X-Achse an, die von links nach rechts ansteigt. Die zweite Dimension des Arrays gibt die Position auf der Y-Achse an, die von oben nach unten ansteigt. D.h. das Element an der Stelle [0][0] ist der Pixel in der linken oberen Ecke des Bilds und das Element an der Stelle [breite - 1][höhe - 1] ist der Pixel in der rechten unteren Ecke des Bilds.

Die Farbe bzw. der **Farbwert** jedes Pixels besteht aus **drei Farbanteilen**: einem Rotanteil, einem Grünanteil und einem Blauanteil mit denen sich unterschiedliche Farben durch Mischung darstellen lassen. Die Anteile werden dabei jeweils durch eine Zahl zwischen 0 und 255 angegeben. 0 bedeutet dabei „nicht enthalten“ und 255 bedeutet „voll enthalten“. Die Farbe schwarz ergibt sich beispielsweise durch rot=0, grün=0, blau=0. Die Farbe weiß ergibt sich durch rot=255, grün=255, blau=255. Ein kräftiger Rotton ergibt sich z.B. durch rot=255, grün=0, blau=0 und ein Gelbton ergibt sich durch rot=255, grün=255, blau=0. Die drei Farbanteile werden kombiniert in einem int-Wert gespeichert (wie, ist für diese Aufgabe nicht relevant).

Die Klasse *Colors* bietet Ihnen 3 statische Methoden an, mit denen Sie einen Farbanteil aus dem int-Wert eines Pixels extrahieren können:

```
int red = Colors.getR(pixels[x][y]);     // gibt den Rotanteil des Pixels an der Stelle x, y zurück
int green = Colors.getG(pixels[x][y]); // gibt den Grünanteil des Pixels an der Stelle x, y zurück
int blue = Colors.getB(pixels[x][y]);    // gibt den Blauanteil des Pixels an der Stelle x, y zurück
```

Zusätzlich erlaubt Ihnen die Klasse *Colors* die Erstellung einer neuen Farbe auf Basis einer Kombination von Rot-, Grün- und Blauanteilen.

```
pixel[x][y] = Colors.toRGB(red, green, blue);      // setzt den Farbwert des Pixels an der Stelle x, y auf
                                                   // die Mischung von red, green und blue
pixel[0][0] = Colors.toRGB(255, 0, 0);             // macht den Pixel in der linken oberen Ecke rot
```

Hinweise/Tipps:
Für die Bearbeitung der nachfolgenden Aufgaben ist es nicht notwendig, die Klassen *Images* und *Colors* zu verändern.
Da es sich sowohl beim Farbwert, als auch den Farbanteilen um int-Werte handelt, kann man beide beim Programmieren schnell mal verwechseln, vor allem wenn diese Werte dann in Variablen gespeichert sind. Darum machen Sie sich nochmal den Unterschied zwischen beiden Begriffen bewusst und achten bei der Implementierung Ihrer Methoden stets darauf, ob Sie gerade einen Farbwert oder einen Farbanteil verwenden. Gegebenenfalls müssen Sie noch den einen in den anderen Wert mit Hilfe der *Colors* Klasse überführen.

Graustufenkonvertierung

In dieser Teilaufgabe geht es um die Umwandlung eines Farbbilds in Graustufen. Ein Grauwert ist ein Farbwert, bei dem die Rot-, Grün- und Blauanteile gleich sind. D.h.:

```
int grauwert = Colors.toRGB(zahl, zahl, zahl); // wobei zahl >= 0 und zahl <= 255.
```

Implementieren Sie dafür zunächst die Methode **convertColorToGrayscale**, so dass diese den zugehörigen Grauwert für einen beliebigen Farbwert bestimmt. Eine einfache Möglichkeit für die Bestimmung des Grauwerts eines beliebigen Farbwerts ist die Bildung des Durchschnitts der Rot-, Grün- und Blauanteile, d.h.

```
int zahl = (red + green + blue) / 3
```

1. red, green, blue initialisieren
2. durchschnitt berechnen
3. grauwert zurückgeben lassen

Beispiele: Der Grauwert für red=99, green=0, blue=0 soll demnach red=33, green=33, blue=33 sein. Der Grauwert für red=1, green=2, blue=3 soll red=2, green=2, blue=2 sein.

Wenden Sie jetzt innerhalb der Methode **convertPictureToGrayscale** Ihre zuvor erstellte Farbkonvertierungsmethode auf jeden Pixel des Eingabebilds an und geben Sie das Ergebnis als Rückgabewert der Methode aus.

Hinweis: Sie können entweder die Grauwerte direkt dem jeweiligen Pixel zuweisen und dann das „pixels“ Array zurückgeben oder Sie kopieren das gesamte Bild während der Verarbeitung, indem Sie ein zunächst ein neues Array mit den gleichen Dimensionen wie „pixels“ allokieren, die veränderten Werte dort einfügen und dann den Verweis auf die konvertierte Kopie zurückgeben.

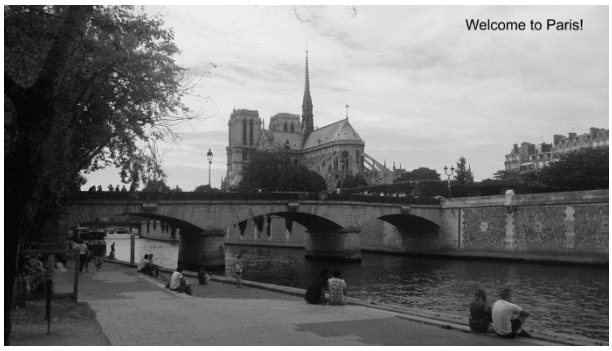


Abbildung 1

forschleife + grauwert

Wenn Sie fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-gray-….png“ an. Wenn alles richtig ist, sollte das Bild so aussehen wie Abbildung 1. Falls alles gekappt hat: herzlichen Glückwunsch zu Ihrem ersten „Filter“ (c.f. Instagram)!

Farbextraktion

In dieser Teilaufgabe geht es um die Extraktion einzelner Farben um einen „künstlerischen“ Effekt zu erzielen. Implementieren Sie hierfür die Methode **andyWarhol,** so dass diese das Bild entlang der X-Achse in 3 gleich große Teile teilt, also:

- x < breite / 3
- breite / 3 <= x < breite *2 / 3
- breite * 2 / 3 <= x

Die Pixel im ersten Bereich sollen so konvertiert werden, dass lediglich der rote Farbanteil erhalten bleibt und der grüne und blaue Anteil auf 0 gesetzt wird. Das erreichen Sie z.B. mit folgendem Aufruf:

```
int newpixel = Colors.toRGB(Colors.getR(oldpixel), 0, 0);
```

Analog dazu soll im 2. Bereich lediglich der grüne Farbanteil und im 3. Bereich lediglich der blaue Farbanteil verbleiben.

Hinweise: Analog zur vorherigen Teilaufgabe können Sie auch hier wieder entweder das Original verändern oder eine Kopie erstellen.

Zur Lösung der Aufgabe können Sie entweder über die 3 Bereiche separat iterieren oder Sie iterieren einmal über das gesamte Bild und entscheiden für jeden einzelnen Pixel in welchem Bereich er liegt. In Abhängigkeit davon können Sie dann die entsprechende Farbe extrahieren und setzen.

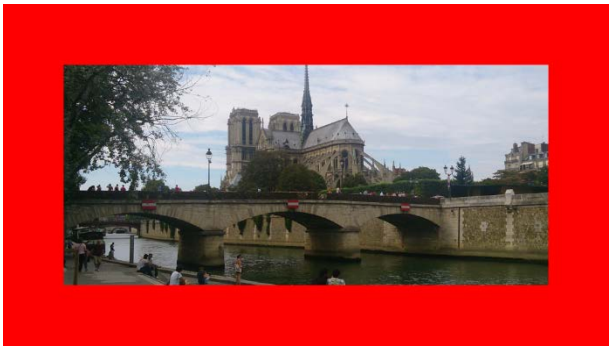
Abbildung 2

Wenn Sie damit fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-warhol-….png an. Wenn alles richtig ist, sollte das Bild so aussehen wie Abbildung 2.

Einfarbiger Rand

In dieser Teilaufgabe geht es um das Erzeugen eines einfarbigen Rands mit konfigurierbarer Farbe und Breite. Implementieren Sie hierfür zunächst die Methode **drawBorderOnPicture**, die den Rand der Breite „borderWidth“ mit der Farbe „borderColor“ einfärbt. Bevor Sie mit der Implementierung beginnen überlegen Sie sich zunächst, wie Sie entscheiden können ob ein Pixel im Randbereich liegt. Für eine „borderWidth“ mit dem Wert 0 gibt es keinen Rand. Für eine „borderWidth“ mit dem Wert 1 soll ein 1-Pixel-breiter Rand das Bild umgeben. Das bedeutet, dass die erste und letzte Zeile (y=0 und y=Höhe-1 sowie die erste und letzte Spalte (x=0 und x=Breite-1) des Bilds den Randbereich bilden. Für „borderWidth“ mit dem Wert 2 sollen zusätzlich die zweite und vorletzte Zeile sowie die zweite und vorletzte Spalte mit der Randfarbe eingefärbt werden, usw.

Hinweise: Auch hier können Sie das Bild wieder entweder direkt manipulieren oder kopieren. Beachten Sie jedoch, dass sie den richtigen Rückgabewert wählen.



Zur Lösung der Aufgabe können Sie entweder über das gesamte Bild iterieren und für jeden Pixel entscheiden, ob dieser im Randbereich liegt oder sie programmieren mehrere Schleifen, mit denen Sie die Pixel im

Randbereich systematisch ändern. Die erste Lösung ist möglicherweise einfacher zu implementieren. Die zweite Lösung ist in vielen Fällen jedoch schneller, da sie die Pixel, die nicht im Randbereich liegen, nicht ändern müssen.

Abbildung 3

Wenn Sie damit fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „note-dame-draw-….png“ an. Wenn alles richtig ist, sollte das Bild so aussehen wie in Abbildung 3.

Ein Nachteil der zuvor implementierten Methode **drawBorderOnPicture** ist der Verlust von Bildinformationen im Randbereich. So verdeckt der rote Rand in Abbildung 3 z.B. den Schriftzug „Welcome to Paris“ im oberen rechten Bildbereich des Originals.

Dieses Problem sollen Sie durch die Implementierung der Methode **extendPictureWithBorder** beheben. Anstatt das bestehende Bild zu „übermalen“, soll zunächst das Zielbild in Abhängigkeit der Bildbreite und -höhe sowie der Rahmengröße allokiert werden. D.h. Sie müssen zunächst berechnen, wie breit und hoch das neue Bild sein soll. Dann müssen Sie ein neues zweidimensionales int-Array erzeugen. Dann müssen Sie das alte Bild an die richtige Stelle des neuen Bilds kopieren und schließlich den Randbereich mit der gegebenen Farbe einfärben.

Hinweise: Zur Lösung dieser Aufgabe ist es erforderlich, dass Sie das Bild kopieren.

Stellen Sie sicher, dass Sie eine Referenz auf die Kopie zurückgeben, da sonst das Original in der Ausgabedatei gespeichert wird und Sie ihre Änderungen nicht sehen können.

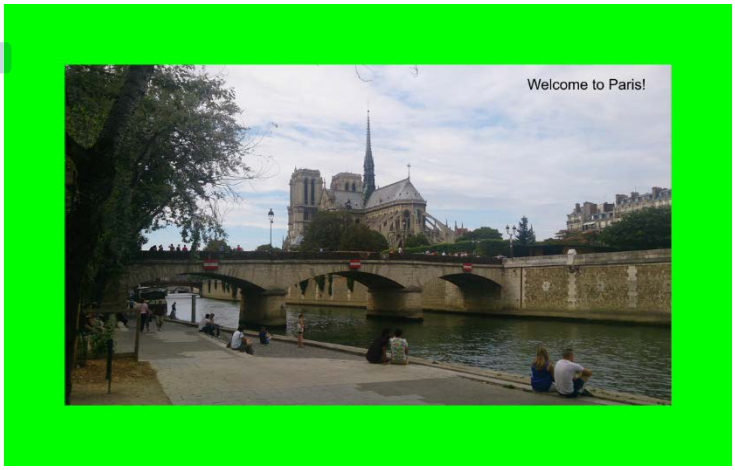


Abbildung 4

Wenn Sie damit fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-extend-….png an. Wenn alles richtig ist, sollte das Bild aussehen wie in Abbildung 4.

---

90 Grad Drehung (für Fortgeschrittene)

Moderne Fotoapparate (und Mobiltelefone) nutzen in der Regel einen Beschleunigungssensor um die Kameraausrichtung beim Fotografieren zu erfassen. Die Kameraausrichtung wird dann verwendet um das Bild automatisch zu drehen, so dass das Foto korrekt ausgerichtet ist. Das automatische Drehen funktioniert in der Regel gut, solange keine Motive am Boden oder an der Decke „geknipst“ werden. Wenn die Ausrichtung nicht korrekt erfasst werden konnte, kann das Bild nicht korrekt gedreht werden. Um dieses Problem zu lösen, sollen Sie die Methode **rotatePicture** implementieren. Die ein Eingabebild um 90 Grad gegen den Uhrzeigersinn drehen soll.

Hierfür müssen Sie sich zunächst überlegen, wie sich die Höhe und Breite eines um 90 Grad gedrehten Bilds im Vergleich zum Original verhält. Dann müssen Sie ein zweidimensionales int-Array mit der richtigen Größe anlegen. Danach müssen Sie sich überlegen, wie die Pixel des Originals auf das gedrehte Bild abzubilden sind.

Hinweis: Es ist keine Schande, wenn Sie sich die allgemeingültige „Formel“ für die Abbildung nicht direkt vorstellen können. Das ist völlig normal. Wichtig ist, dass Sie jetzt nicht aufgeben, sondern das „zu schwierige“ Problem systematisch mit kleineren Schritten angehen.

Eine typische Möglichkeit zum Lösen von „zu schweren“ Aufgaben ist die Nutzung eines konkreten Beispiels: Nehmen Sie einfach einen Stift und ein Blatt Papier und überlegen Sie sich, wo der Pixel in der linken oberen Ecke landet, wenn Sie das Bild um 90 Grad gegen den Uhrzeigersinn drehen. Dann nehmen Sie den Pixel daneben und wiederholen Sie das Ganze. Sie werden sehen, dass sie recht schnell die „Formel“ für die Abbildung „sehen“ und ableiten können.

Abbildung 5

Beispiel:

pixels[i].length - 1

- Links oben (0, 0) wird zu links unten (0, höhe – 1).
- Rechts neben links oben (1, 0) wird zu eins über links unten (0, höhe – 2).
- Rechts neben rechts neben links oben (2, 0) wird zu eins über eins über links unten (0, höhe – 3).
- Eins unter links oben (0, 1) wird zu eins rechts von rechts unten (1, höhe – 1)
- …

pixels[i].length - 2

Beobachtungen:

- Offensichtlich wirkt sich der x-Wert im Original auf den y-Wert im gedrehten Bild aus. Je größer der x-Wert, desto kleiner der y-Wert.
- Ebenso wirkt sich der y-Wert im Original auf den x-Wert im gedrehten Bild aus. Je größer der y-Wert, desto größer der x-Wert.

Wenn Sie mit Ihrer Implementierung fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-rotate-….png“ an. Wenn alles richtig ist, sollte das Bild aussehen wie in Abbildung 5.

Verkleinern (zum Nachdenken)

Digitale Fotoapparate (und Mobiltelefone) sind heutzutage in der Lage, Bilder mit sehr hohen Auflösungen aufzunehmen. Die Metrik dabei ist in der Regel „Mega-Pixel“ (also eine Million Pixel). Das ist zwar schön, wenn Sie gestochen scharfe Urlaubsbilder archivieren möchten. Aber es kann recht unpraktisch sein, wenn Sie ein Urlaubsbild über ein langsames Telefonnetz an Ihre Freunde per Email schicken möchten, da die Zahl der Pixel einen direkten Einfluss auf die Dateigröße hat.

Um die Dateigröße zu reduzieren, sollen Sie in der Methode **shrinkPicture** das Bild verkleinern indem Sie nur jeden n-ten Pixel in ein neues Bild kopieren. D.h. bei einer Eingabe von n=1 soll das Bild im Original zurückgegeben werden. Bei n=2 soll jede zweite Spalte und jede zweite Zeile übersprungen werden. Bei n=3 soll jede zweite und jede dritte Spalte sowie jede zweite und jede dritte Zeile übersprungen werden, usw.

Implementieren Sie hierfür zunächst die Methode **computeShrinkedSize**, die ausgehend von einer Zeilen- bzw. Spaltenzahl (length) und einem Skalierungswert (n) die Höhe bzw. Breite des verkleinerten Bilds berechnet. Beachten Sie hierbei, dass die Höhe bzw. Breite des Originalbilds nicht notwendigerweise durch den Wert n (ganzzahlig) teilbar ist. Sollte dies der Fall sein, erhöhen Sie den Rückgabewert um 1, damit kein möglicher Pixel verloren geht. In anderen Worten: falls die Division von length durch n einen Rest > 0 ergibt (siehe Modulo Operator), soll length/n+1 und nicht nur length/n zurückgegeben werden.

Als nächstes können Sie mit der Implementierung von **shrinkPicture** beginnen indem Sie sich mit Hilfe Ihrer computeShrinkedSize Methode, die Zielbreite und -höhe des verkleinerten Bilds berechnen und sich ein neues int-Array für die Ausgabe erstellen. Iterieren Sie dann über das neue int-Array und kopieren Sie in jedem Schritt den Pixelwert an der entsprechenden Stelle im Originalbild. Wenn Sie sich die „Formel“ für die Abbildung nicht vorstellen können, machen Sie sich einfach ein „Papierbeispiel“ z.B. mit einem 5x3 Bild und dem Wert n=2. Dann ändern Sie n auf 3 und wiederholen das Ganze bis Sie den Zusammenhang „sehen“ und als Formel angeben können.

Wenn Sie mit Ihrer Implementierung fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-shrink-….png“ an. Das Bild sollte eine verkleinerte Version des Originals sein.

Interpolation (zum Knobeln)

Die oben genannte Methode zur Verkleinerung von Bildern sieht zwar schon ganz gut aus, führt aber zu einem starken Datenverlust, da nur jede n-te Zeile und Spalte berücksichtigt wird. Das ist kein Problem für Landschaftsaufnahmen – wie z.B. Notre Dame.

Wenn Sie sich jedoch den Schriftzug in der oberen rechten Ecke des Bilds genauer anschauen, werden Sie leicht feststellen können, dass die Verkleinerung durch „einfaches weglassen“ von Pixeln das Schriftbild deutlich negativ beeinträchtigt. Dieser Effekt wird bei ansteigenden Werten für n schnell problematisch.

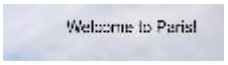


Abbildung 6

Um dieses Problem abzumildern sollten bei der Verkleinerung des Bilds die Farbwerte der „weggelassenen“ Pixel berücksichtigt werden. Eine einfache Möglichkeit hierfür ist das bestimmen des Farbwerts des verbleibenden Pixels auf Basis der durchschnittlichen Rot-, Grün- und Blauanteile aller Pixel im Originalbild, den dieser Pixel repräsentiert. Implementieren Sie hierfür zunächst die Hilfsmethode **computeAverageColor**, die die folgenden 4 Parameter hat:

- pixels: Das zu bearbeitende Bild als 2-dimensionales Array.
- startX, startY: Die Position eines Pixels im Bild, der bei der einfachen Verkleinerung (aus der vorherigen Teilaufgabe) in das neue Bild übertragen wird.
- n: Die Verkleinerungsstufe, mit der Sie die Zahl der Spalten (rechts neben startX) und Zeilen (unterhalb von startY) bestimmen können, die bei der Verkleinerung übersprungen werden (also startX <= x < startX + n, startY <= y < startY + n).

Als Rückgabewert soll die Funktion ausgehend vom Pixel startX, startY über alle weggelassenen Punkte iterieren und dabei die komponentenweisen Durchschnitte der Rot-, Grün- und Blauanteile bilden und diese dann mit Hilfe der Colors.toRGB Methode als int codieren und zurückgeben.

D.h. Sie brauchen zwei verschachtelte Schleifen, die Ihnen ausgehend von einem Startwert für x und y alle Kombinationen von x- und y-Werten bilden (mit Hilfe von n), so dass die Menge der Punkte aus den Punkten bestehen, die bei der Verkleinerung zu einem Pixel zusammengefasst werden sollen. Für jeden dieser Punkte müssen Sie den Rot-, Grün- und Blauanteil extrahieren (Colors.getR, Colors.getG, Colors.getB) und diese dann für alle Punkte aufsummieren. Am Schluss müssen Sie die

Summen jeweils durch die Zahl der aufsummierten Pixel teilen und dann mit Hilfe von Colors.toRGB die resultierende Farbe zurückgeben.

Hinweis: Beachten Sie, dass Sie bei der Implementierung nicht davon ausgehen können, dass die Bildbreite und -höhe durch n teilbar sind. Das bedeutet, dass Sie in Ihrer verschachtelten Schleife auch die Höhe und Breite des Bildes (also die Größen der Dimensionen des „pixel“ arrays) berücksichtigen müssen.

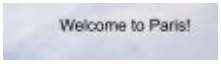


Abbildung 7

Wenn Sie mit der Implementierung fertig sind, müssen Sie noch die Methode **shrinkPictureSmooth** implementieren. Diese besitzt die selbe grundlegende Logik der Methode **shrinkPicture** aus der vorherigen Teilaufgabe (also: kleineres Bild allokieren, für jeden Pixel den „richtigen“ Pixel setzen und die Referenz auf die Kopie zurückgeben). Die einzige Änderung, die Sie sie vornehmen müssen, ist es das Setzen der „richtigen Pixel“ mit einem entsprechenden Aufruf der **computeAverageColor** Methode zu ergänzen**.**

Wenn Sie mit Ihrer Implementierung fertig sind, führen Sie das Programm aus und schauen Sie sich das Bild „notre-dame-smooth-….png“ an. Wenn alles richtig ist, sollte die Schrift in der oberen linken Ecke so aussehen wie in Abbildung 7.