

Seam Carving in C (15 Punkte)



In diesem Projekt werden Sie ein Programm zur *intelligenten Bildskalierung* (engl. „Seam Carving“) implementieren. Intelligente Bildskalierung ist ein Verfahren zur Skalierung des Seitenverhältnisses von Bildern bei dem Proportionen und Form relevanter Bildinhalte erhalten bleiben oder möglichst wenig verzerrt werden. Es wurde ursprünglich von Shai Avidan vom Mitsubishi Electric Research Labs (MERL) und Ariel Shamir (Interdisciplinary Center und MERL) auf der Computergrafik-Konferenz SIGGRAPH 2007 vorgestellt. Eine detaillierte Beschreibung des Algorithmus, Beispiele in Bildern und Videos sowie weitere Verwendungsmöglichkeiten und Erweiterungen finden Sie auf der Website der Erfinder:

<http://www.faculty.idc.ac.il/arik/SCWeb/imret/>.

Im Gegensatz zum vorherigen Projekt, erhalten Sie nur ein `Makefile`. Der weitere Entwurf des Projektes ist Ihnen freigestellt. Das Projekt kann mit Hilfe des Befehls `make` einfach auf der Befehlszeile kompiliert und gebunden werden. Sie können weitere `*.c` und `*.h` Dateien nach Belieben im `src` Verzeichnis anlegen. Die Ausgabedateien des Übersetzers – inklusive der erzeugten Anwendung `carve` – finden Sie im `build` Verzeichnis. Sie erhalten das Programmierprojekt mit

```
git clone https://prog2scm.cdl.uni-saarland.de/git/project3/<NAME>
```

Ersetzen Sie wie gewohnt `<NAME>` durch Ihren Benutzernamen.

1 Intelligente Bildskalierung

Der Algorithmus erstellt eine Entbehrlichkeitswertung für alle Bereiche/Pixel des Bildes und sucht anhand dieser dann Pfade möglichst unwichtiger Pixel, die quer zur Stauchungs- oder Streckungsrichtung verlaufen und fügt an diesen Fugen (engl. *seams*) Pixel ein oder entfernt sie. Unterschiedliche Varianten des Grundalgorithmus unterscheiden sich in der Regel in den Strategien zur Erstellung der Entbehrlichkeitswertung. Im von uns betrachteten, ursprünglichen Fall geschieht dies nur anhand des Unterschieds der Farbwerte zwischen benachbarten Pixeln. Außerdem beschränken wir uns auf den Fall der Bildverkleinerung.

Im Wesentlichen arbeitet der Algorithmus wie folgt:

- Berechnung der lokalen Energie jedes Pixels.
- Berechnung der Energie aller zusammenhängender vertikaler Pixel-Pfade.
- Berechnung des Pfades mit minimaler Energie.
- Entfernen der Pixel dieses Pfades aus dem Bild.

1.1 Algorithmus

Die Energiewerte jedes einzelnen Pixels müssen in Beziehung gesetzt werden um die Kosten aller möglichen Fugen des Bildes zu berechnen und die Fuge mit den minimalen Kosten auszuwählen. Die Kosten aller Fugen werden berechnet, indem von oben nach unten die minimale Energie seiner Vorgänger entgegen der Fugenrichtung (oben mitte, oben links, oben rechts) zur lokalen Energie jedes Pixels addiert wird.

Die minimalen Kosten eines Pixels setzen sich aus seiner lokalen Energie und den (bereits berechneten) minimalen Kosten seiner Nachbarn entgegen der Fugenrichtung zusammen. Die lokale Energie eines Pixels berechnet sich aus der Summe der berechneten Farbunterschiede (siehe Abschnitt 2.3) zwischen dem aktuellen und dem linken und dem oberen Pixel zusammen (falls vorhanden).

Die zu entfernende, optimale Fuge, entspricht dem Pfad zusammenhängender Pixel, dessen Kosten minimal sind. Weil die berechneten Kosten jedes Pixels bereits eine Akkumulation der Kosten aller darüber liegender Pixel sind, muss dafür nur die unterste Zeile betrachtet werden. Das Pixel mit den minimalen Kosten in dieser Zeile ist das Pixel, an dem die optimale Fuge endet.

Um die Entfernung von Fugen eindeutig zu machen, verwenden wir folgende Regeln: Falls es mehrere optimale Fugen mit unterschiedlicher Endspalte gibt, so ist die Fuge mit der kleinsten x-Koordinate zu entfernen. Hat ein Pixel mehrere optimale Nachbarn, so ist der Nachbar senkrecht nach oben, gefolgt vom linken oberen Nachbarn zu bevorzugen.

1.2 Dynamische Programmierung

Die Berechnung von Energie und optimaler Fuge ist prinzipiell auch mit einer “brute-force”-Methode möglich: Die Berechnung der akkumulierten Kosten eines Pixels benötigt die Kosten der Nachbarpixel entgegen der Fugenrichtung. Ein naiver Ansatz würde für ein beliebiges Pixel rekursiv dieselbe Methode aufrufen bis es in der ersten Zeile ankommt. Auf diese Weise würden jedoch die Kosten vieler benötigter Pixel mehrfach berechnet werden, da z.B. zwei benachbarte Pixel sehr viele gemeinsame Pfade haben, auf denen sie zu erreichen sind. Dies kostet bei größeren Bildern bedeutend mehr Rechenzeit, wodurch die Technik für einen Benutzer nicht mehr bequem anwendbar ist.

Das im vorherigen Abschnitt gezeigte Verfahren verwendet *dynamische Programmierung* um dieses Effizienzproblem zu lösen: Die Berechnung beginnt ganz oben und speichert die akkumulierten Kosten jedes Pixels in einer Reihung. In der nächsten Zeile werden die Ergebnisse der im letzten Schritt berechneten Pixel aus dieser Reihung geladen, anstatt sie rekursiv neu zu berechnen.

2 Allgemeine Hinweise zur Bearbeitung des Projekts

2.1 Befehlszeilenargumente

Ihre Anwendung soll folgende Befehlszeilensyntax implementieren:

```
carve [-h] [-s] [-p] [-n <count>] <image file>
```

Es folgt eine Beschreibung der Befehlszeilenargumente im Einzelnen:

- `-h` zeigt die Hilfe an und beendet die Anwendung.
- `-s` gibt Statistiken über das Eingabebild aus und beendet die Anwendung (siehe Aufgabe 1).
- `-p` gibt die Spaltenindizes des minimalen Pfades aus und beendet die Anwendung (siehe Aufgabe 2).
- `-n <count>` führt `<count>` Schritte des Algorithmus aus; falls nicht angegeben läuft die Anwendung bis zum Ende (siehe Aufgabe 3).
- `<image file>` ist das Eingabebild; beachten Sie das dieses Argument an jeder Position der Befehlszeile auftauchen kann und *nicht* notwendigerweise das letzte Argument ist.

Sie dürfen zum Implementieren des Befehlszeilenargumentenparsers die POSIX-Funktion `getopt` benutzen. Lesen Sie dazu das Handbuch, das Sie mit `man 3 getopt` aufrufen können.

2.2 Bildformat

Als Ein- und Ausgabeformat verwenden wir wieder das *portable pixmap format*, das Sie bereits aus Projekt 2 kennen. Beachten Sie, dass, anders als in Projekt 2 behauptet, die korrekte Dateiendung für *RGB*-Bilder im *portable pixmap format ppm* und nicht *pbm* ist. In diesem Projekt verwenden wir nun die korrekte Endung *ppm*.

Für dieses Projekt brauchen Sie auch eine Routine zum Einlesen von *ppm*-Dateien. Beachten Sie, dass Daten in der Eingabedatei von beliebigen *whitespaces* getrennt sein können (siehe `man 3 isspace`). Sie können davon ausgehen, dass die Eingabedateien *keine* Kommentare enthalten. Schreiben Sie allerdings Ihre Einleseroutine robust, die Fehler in der Eingabedatei abfängt und die Anwendung ggf. mit `EXIT_FAILURE` beendet (siehe `man 3 exit`). Zu diesen Fehlern gehören zu wenig/zu viel Bildpunkte, Höhen-/Breitenangaben kleiner gleich null und ähnliches. Sie dürfen in diesem Fall *keine* Ausgaben auf `stderr` erzeugen, um alle Tests zu bestehen.

2.3 Farbunterschied zwischen Pixeln

Als Farbunterschied definieren wir die Summe der Quadrate der Differenzen der einzelnen Farbkomponenten:

$$(red(x) - red(y))^2 + (green(x) - green(y))^2 + (blue(x) - blue(y))^2$$

2.4 Weitere Hinweise

- Wir empfehlen dringend das mitgelieferte `Makefile` nicht zu verändern, da wir auf unseren Testservern diese Datei durch unsere eigene ersetzen.
- Buchen Sie bitte keine weiteren Bilddateien oder andere Daten, die keine Quelldateien sind, in Ihr git-Depot ein.
- Sie können nach Belieben Quelldateien im `src` Verzeichnis anlegen. Legen Sie dort jedoch keine weiteren Unterverzeichnisse an.
- Wenn Sie neue Quelldateien anlegen, müssen Sie das Projekt einmalig mit `make clean; make` neu bauen, damit `make` die neuen Dateien in Betracht zieht.
- Wenn Sie gerade erst begonnen haben zu implementieren, werden die Test-Bilder `small1` und `small2` hilfreich sein (siehe auch Abschnitt 4).

3 Aufgaben

Aufgabe 1: Statistiken ausgeben, 6 Punkte

Implementieren Sie einen Kommandozeilenschalter `-s`, der Statistiken zum eingelesenen Bild ausgibt und anschließend das Programm beendet. Geben Sie Bildbreite und -höhe sowie dessen Helligkeit aus.

Gehen Sie wie folgt vor, um die Helligkeit zu berechnen:

- Die Helligkeit eines Bildpunktes erhalten Sie, indem Sie alle drei Farbkanäle addieren und durch drei teilen:
$$(red(x) + green(x) + blue(x)) / 3$$
- Die Helligkeit des gesamten Bildes erhalten Sie, indem Sie die Helligkeit aller Bildpunkte aufaddieren und durch die Anzahl der Pixel teilen.
- Benutzen Sie Ganzzahlarithmetik und *kein* kaufmännisches Runden.

Benutzen Sie für die Ausgabe folgendes Format und geben Sie keine weiteren Ausgaben auf `stdout` aus:

```
1 printf("width: %u\n", <width>);
2 printf("height: %u\n", <height>);
3 printf("brightness: %u\n", <brightness>);
```

Aufgabe 2: Minimalen Pfad ausgeben, 6 Punkte

Implementieren Sie einen Kommandozeilenschalter `-p`, der gemäß des Algorithmus den Pfad mit *minimaler* Energie durch das eingelesene Bild ausgibt und anschließend das Programm beendet.

- Geben Sie für jede Bildzeile einen `x`-Index aus, der auf dem gefundenen Pfad liegt.
- Geben Sie einen Index pro Zeile aus.
- Geben Sie den Pfad von unten nach oben aus.
- Geben Sie keine weiteren Ausgaben auf `stdout` aus.

Aufgabe 3: Intelligentes Skalieren des Bildes, 6 Punkte

Implementieren Sie einen Kommandozeilenschalter `-n <count>`, der `<count>` Schritte des Algorithmus durchführt und das Ergebnis in einer Ausgabedatei `out.ppm` abspeichert.

- Jeder Schritt verringert die Breite des Bildes um einen Pixel; insbesondere ergibt `-n 0` eine Kopie des Originalbildes.
- Für jeden Pixel Breite, den Sie das Bild verkleinern, fügen Sie einen schwarzen Pixel an den rechten Rand des Bildes ein. Dies führt dazu, dass das Ausgabebild die gleiche Breite hat, wie das Eingabebild. Allerdings hat das Ausgabebild einen schwarzen rechten Rand der Breite `<count>`.
- Ist `-n -1` oder gar kein Schalter angegeben, läuft der Algorithmus bis zum Ende. Das heißt, es werden gemäß der Bildbreite viele Schritte durchgeführt und das Ergebnis ist ein schwarzes Bild.

4 Tests

Sie können im Basisverzeichnis die *public* Tests mit dem Kommando `./run-tests.py` selbst durchführen. Die Beispiel-Eingabebilder finden Sie im `data` Verzeichnis. Die Referenzausgaben für die Tests finden Sie im `test_data` Verzeichnis. Sie müssen alle public Tests eines Aufgabenteils bestehen, um in diesem Aufgabenteil überhaupt Punkte zu erlangen. Die public Tests werden außerdem in regelmäßigen Abständen zusammen mit geheimen *daily* Tests auf unserem Testserver ausgeführt. Sie erhalten über den Ausgang der Tests eine Benachrichtigung per Email. Nach Ende des Projekts wird Ihre Abgabe mithilfe weiterer *secret* Tests ausgewertet. Wir ziehen dazu den Zustand Ihres git-Depots im `master` Zweig *vom 7.6.2016 um 23:59* heran.

4.1 Einzelne Tests

Sie können auch einzelne Tests ausführen, was insbesondere dann nützlich ist, wenn größere Teile Ihrer Implementierung noch fehlen.

- Mit `./run-tests.py -l` können Sie sich die Namen aller Tests anzeigen lassen.
- Mit `./run-tests.py -f <name>` lassen Sie nur den Test `<name>` laufen.