Einführung in die VIGRA*PLT* Bibliothek

Version 0.5.0

Benjamin Seppke

AB KOGS
Dept. Informatik
Universität Hamburg

Inhalt

- Konzepte
- Grundlegende Funktionen
- Beispiele
- Basis für eigene Algorithmen

Inhalt

- Konzepte
  - Basis
  - Repräsentation von Bildern
- Grundlegende Funktionen
- Beispiele
- Basis für eigene Algorithmen

Basis I

- PLT Scheme, eine Scheme Implementation mit vielen Vorteilen:
  - Gute Verfügbarkeit für viele Betriebssysteme
  - Schöner interaktiver Editor: DrScheme
  - Erfahrungen aus SE3
  - Viele vorhandene Teachpacks
  - Seit kurzen: Verfügbarkeit eines OO-Systems (CLOS)

Basis II

- Library stellt Anbindung an die VIGRA-Bibliothek her
  (über C++ → C-Wrapper)
- Relativ leichte Installation
- Sehr einfache Einbindung über:

```
(require vigraplt/vigraplt)
```

Repräsentation von Bildern I

- Alle Bilder sind Exemplare der Struktur `image`
- Images sind List von Bildkanälen
- Interne Repräsentation von Bildkanälen als Scheme `cvector` vom Typ `float`
  - Für Interaktion über das Foreign Function Interface (FFI) und entspricht einem C-Array
  - Fest im PLT Scheme FFI enthalten
  - Nur eindimensional definiert, aber multidimensional erweitert!

Repräsentation von Bildern II

- Wertebereiche von Kanälen nach Import
  - `[min...max]`
  - Meistens `[0.0 … 255.0]`, allerdings nicht immer (Gegenbeispiel: Tiff)
- Beispiel: Grauwert-Bild nach dem Import:
- '([[0.0 … 0.0]
  …
  [0.0 … 0.0]])

Inhalt

- Konzepte
- Grundlegende Funktionen
  - Funktionen von Images
  - Bild I/O und Konvertierung
  - Funktionen höherer Ordnung
- Beispiele
- Basis für eigene Algorithmen

Funktionen von Images I

- Erzeugen eines leeren Bildes mit Breite *w* und Höhe *h* und *numbands* Kanälen:
  `(make-image w h numbands)`
- Oder mit einem initialen Wert i:
  `(make-image w h numbands . i)`
  - Beispiel für Grauwert-Bild:
    ```
    (define gimg
     (make-image 10 10 1 0.0)
    ```
  - Beispiel für Farb(RGB)-Bild:
    ```
    (define fimg
     (make-image 10 10 3 0.0 255.0 0.0)
    ```

Funktionen von Images II

- Bandinformationen (Anzahl Kanäle)

```
(image-numbands image)
```

- Größeninformationen
  - Die Höhe eines Bildes
    `(image-height image)`
  - Die Breite des Bildes
    `(image-width image)`

Funktionen von Images III

- Zugriffsfunktionen für Bilder
  - Eine Kopie des Bildes
    `(copy-image image)`
  - Den Farbwert im Kanal band_id an der Stelle x,y auslesen
    `(image-ref image x y band_id)`
  - Den Farbwert im Kanal band_id an der Stelle x,y auf i setzen
    `(image-set! image x y band_id i)`

Bilder Input/Output

- Bilder laden, speichern und anzeigen
  - Ein Bild einlesen
    `(loadimage filename)`
  - Ein Bild abspeichern
    `(saveimage image filename)`
  - Ein Bild im Viewer anzeigen (optional Fenstertitel)
    `(show-image image . windowtitle)`

Konvertierung von Bildern

- Umwandlung in eine Liste (pro Kanal: Liste von Zeilen-Listen)
  `(image->list image)`
- Umwandlung einer Liste (pro Kanal: List von Zeilen-Listen) in ein Bild
  `(list->image lst)`

Funktionen höherer Ordnung

- Abbilden (wie mit dem bekannten “`map`“)
  `(image-map func image)`
- „Falten“ eines Bildes (wie mit dem bekannten „`reduce`“)
  `(image-reduce func image seed)`
- Eine Funktion auf jede Pixel-Koordinate anwenden
  `(image-for-each func image)`

Inhalt

- Konzepte
- Grundlegende Funktionen
- Beispiele
  - Konvertierung von Bildern und Bildformaten
  - Invertieren von Bildern
- Basis für eigene Algorithmen

Konvertierung von Bildformaten

1. Schritt: Einlesen eines Bildes (PNG)

```
(define myImage
   (loadimage „test-image.png“))
```

2. Schritt: Betrachten des Bildes

```
(show-image myImage
                „Image-Viewer: Testbild“)
```

3. Schritt: Abspeichern im TIFF-Format

```
(saveimage myGrayImage „test-image.tif“)
```

Invertierung von Bildern

- Angenommen `myImage` sei ein Grauwertbild.
- Aufgabe: Invertieren Sie das Bild!
- Beispiel:

- Formal:

$$\forall \vec{p} \in \Omega: f'(\vec{p}) = maxvalue - f(\vec{p})$$

wobei $maxvalue$ der maximal mögliche Farbwert ist (meist 255 bei importierten Grauwertbildern.

- Idee: Eine Funktion, die einen Grauwert invertiert und diese Funktion auf das Bild „mappen“.

Invertierung von Bildern

- Die Funktion für Grauwertbilder:

```
(define (invert intensity) (- 255.0 intensity))
```

- Anwendung der Funktionen auf das Bild:

```
(define myInvertedImage
   (image-map invert myImage)
```

- Falls das Bild nicht im Wertebereich 0..255 liegt, müssen wir es leicht abändern:

```
(define (invert_max maxvalue intensity)
   (- maxvalue intensity))
```

- Anwendung dieser Funktion:

```
(define myInvertedImage2
   (image-map (curry invert_max
                     (image-reduce max myImage 0))
              myImage)
```

Inhalt

- Konzepte
- Einlesen und Abspeichern von Bildern
- Beispiele
- Basis für eigene Algorithmen
  - Iterationen über Pixel
  - Referenz- oder Wertsemantik?

Iterationen über Pixel I

- Die meisten Algorithmen müssen Bilder „ablaufen“, wie zum Beispiel bei der Konturextraktion
- Dazu unterschieden wir zwei Fälle
  - Iterieren, falls es in einer geordneten Abfolge passiert
    (z.B. erst zeilen- dann spaltenweise) und
  - Traversieren, wenn die Abfolge der Pixel mehr oder weniger ungeordnet ist (z.B. bei der Konturextraktion)

Iterationen über Pixel II

- Falls iteriert werden muss, so kann man zusätzlich noch unterscheiden zwischen:
  - Rein lokalen Verfahren, die nur den momentanen Pixelwert benötigen (z.B. Schwellenwertverfahren)

    → Verwendung von `image-map[!]`
  - Verfahren die Informationen über die aktuelle Position benötigen (z.B. um benachbarte Pixel zu ermitteln)

    → Verwendung von `image-for-each`

Referenz- oder Wertsemantik?

- Ein Bild sagt mehr als 1000 Worte...
- aber es belegt auch mehr als 1000x so viel Speicher!
- Um Speicher zu sparen, kann man auch referenziell auf den Bildern arbeiten:
  - Die Funktionen, die mit einem ! gekennzeichnet sind, manipulieren die eigentlichen Daten!
    - Speichersparnis 👍
    - Bezugstransparenz 👎
- Achtung! Im Zweifelsfall lieber mit Kopien arbeiten und den höheren Speicherbedarf in Kauf nehmen!
- Und: nicht alle Algorithmen lassen sich In-Place programmieren!

Ende der Einführung

Vielen Dank für die Aufmerksamkeit!