Lernziele

- Sie vertiefen Konzepte aus der Vorlesung wie Zeiger, Referenzen, Arrays, Klassen und Structs.
- Sie lernen den Umgang mit Dateiformaten, im Spezifischen mit Bildformaten.
- Sie lernen Grundkonzepte der Bildverarbeitung kennen (einem Bereich, in dem C++ in der Praxis oft angewandt wird).

Übersicht

Wir beschäftigen uns in dieser Übung mit Verarbeitung von astronomischen Bildern. Das Szenario für die folgenden Aufgaben ist folgendes: Sie entwickeln Teile einer Steuerungssoftware für ein Teleskop, das aus den Aufnahmen Informationen ausliest, wie z.B. die Einstellung der Helligkeit. Die Software soll ausserdem einen einfachen Algorithmus enthalten, der das hellste Objekt auf dem Bild erkennt.

Erste Schritte mit dem Codeskelett

Das Projekt ist mit dem CMake-System angelegt. Sie können eine Entwicklungsumgebung nach Wahl verwenden, z.B. Visual Studio, CLion oder VS Code. Um die Korrektheit ihrer Lösung zu testen, sind bereits passende Tests vorgegeben.

Umgebung aufsetzen

Im Folgenden sind empfohlene Setups der Entwicklungsumgebung beschrieben, die für das jeweilige Betriebssystem am einfachsten zu handhaben sind.

Windows Visual Studio ist für die Entwicklung auf Windows sehr empfehlenswert. Öffnen Sie den Ordner mit der Codevorlage. Visual Studio sollte jetzt automatisch die CMake-Konfiguration verarbeiten, und anschliessend können Sie in der Menüleiste als Debug-Ziel `astro_test.exe` auswählen, um die Tests zu starten.

macOS / Linux Sie können entweder mit einem leichteren Texteditor wie VS Code und der Konsole arbeiten oder mit einer integrierten Entwicklungsumgebung wie CLion. Die Shell-Skripte `build.sh` und `test.sh` enthalten alle nötigen Schritte.

Um ihre Umgebung zu prüfen, führen Sie einmal das Skript `./build.sh` aus. Sehen Sie in der Ausgabe die Nachricht “Built target TestAstro”, ist alles korrekt aufgesetzt. Sie können jetzt auch das kompilierte Programm mit `./bin/astro` starten, das bereits ohne Ihre Lösungen den Grundablauf durchführt.

Übersicht

- **CMakeLists.txt**: Die Konfiguration des C++-Projekts, muss nicht bearbeitet werden.
- **include**: Hier befinden sich die Headers (`.hpp`) für unser Projekt.
- **src**: Hier befinden sich die Implementierungen (`.cpp`), an denen Sie arbeiten werden.
- **test**: Um Ihre Lösungen direkt prüfen zu können, stellen wir hier Tests zur Verfügung auf Basis von GoogleTest [2] zur Verfügung. Die Dateien sollten nicht bearbeitet werden.
- **Picsi.jar**: Ein Java-Programm, das die mitgelieferten Bilder anzeigen kann.
- **build.sh**: Ein Skript, das die nötigen Schritte für eine Kompilierung mit CMake ausführt.
- **test.sh**: Dieses Skript erlaubt es Ihnen, direkt alle Aufgaben zu testen. Führen Sie es aus (optional mit Aufgabennummer), um Ihre Lösung kompilieren und testen zu lassen.

Die Stellen im Quellcode, die im Rahmen der Aufgaben ergänzt werden sollen, sind jeweils mit einem Kommentar der Form `// TODO [Aufgabe] 1.a) ...` gekennzeichnet. Alle anderen Stellen im Code sollten Sie nicht verändern (ausser um evtl. zusätzliche Importe einzubinden).

Zusätzlich zum manuellen Testen mit dem Programm können Sie Ihre Lösung für jede einzelne Aufgabe anhand der mitgelieferten Tests verifizieren. Diese rufen die von Ihnen implementierten Funktionen auf und prüfen die Ausgaben auf ihre Korrektheit. Sie können diese Tests so oft wie gewünscht ausführen, auch während Ihr Code noch unvollständig ist.

Abgabe

Senden Sie Ihren Quellcode (die Dateien in `src`) als Mail-Anhänge an `christoph.stamm@fhnw.ch`.
Deadline: 29.10.2023

Aufgabe 1: Einlesen von Bildern

Bildformate

Es gibt viele Möglichkeiten, ein Bild in den binären Computerspeicher zu übersetzen. Zu diesem Zweck wurden Formate wie JPEG, PNG, GIF und viele weitere entworfen. Solche Formate sind nicht ganz einfach einzulesen, da sie zum Teil mit komplexen Kompressionsverfahren den Speicherplatzbedarf reduzieren oder möglichst flexibel gestaltet sind und deshalb viele verschiedene Formate beinhalten können. Beim Umgang mit solchen Bildern verlässt man sich deshalb typischerweise auf externe Bibliotheken (engl. *libraries*), um das Rad nicht neu erfinden zu müssen.

Für die vorliegende Übung möchten wir aber den Umgang gesamthaft nachvollziehen und selbst implementieren können. Wir arbeiten mit dem Format namens Portable Anymap Format (PNM) [3]. Dieses wurde mit dem Ziel entwickelt, möglichst simpel zu sein und Übertragung auch via Text statt nur Binärcode zu erlauben.

Es gibt drei Typen des PNM-Formats, die sich im Farbraum unterscheiden: schwarz-weiss (PBM), Graustufen (PGM) und Farben (PPM). Wir werden mit dem Farbformat PPM (Portable Pixel Map) arbeiten. Für die folgenden Aufgaben ist die Formatspezifikation [4] hilfreich, um das Format zu verstehen.

Um die Ein- und Ausgaben visuell prüfen zu können, ist es empfehlenswert, dass eine Applikation auf Ihrem System das PNM-Format lesen kann (bzw. spezifisch das PPM-Format, das wir in unseren Aufgaben verwenden). Auf macOS ist das mit der vorinstallierten *Vorschau* mölich, auf Windows / Linux können Sie das Tool *Picsi* (in der Vorlage als `Picsi.jar` mitgeliefert) verwenden. Öffnen Sie eines der `.ppm`-Bilder im Ordner `img` der Projektvorlage, um zu prüfen, ob Sie das Bildformat lesen können.

Textformat einlesen

Die erste Aufgabe befasst sich mit der Variante *Plain PPM*, welche besonders gut verständlich ist, da sämtliche Daten auch als Text gelesen werden können.[1] Das Bild, das wir in dieser Aufgabe einlesen werden, ist samt Quelltext in Abb. 1 sichtbar.

```
1 P3
2 3 2
3 255
4 255   0 0   0 255   0 0 0 255
5 255 255 0 255 255 255 0 0   0
```

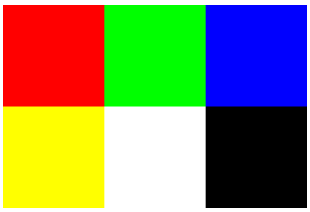


Abbildung 1: Ein winziges Bild aus 6 Pixeln im Plain-PPM-Format, das wir in der ersten Aufgabe einlesen (`img/01_ex_t3.ppm` bzw. `img/01_ex_t6.ppm`).

Orientieren Sie sich an der Spezifikation [4], um das in 1 gezeigte Bild zu verstehen. Im vorgegebenen Projekt ist bereits eine Klasse `RGBImage` definiert, welche die Daten speichern soll (siehe `image.hpp` im `include`-Verzeichnis).

[1] Dies ist üblicherweise nicht der Fall bei Bildformaten, was einfach überprüft werden kann, wenn man ein beliebiges Foto in einem Texteditor versucht zu öffnen.

Aufgabe 1.a) Plain PPM einlesen

Implementieren Sie die Funktion `RGBImage::load` in `image.cpp`, sodass sie Bilder im Format Plain PPM einlesen kann.
Sie dürfen folgende vereinfachende Annahmen treffen:

- Es sind keine Kommentare in der Datei enthalten.
- *Maxval* ist bei allen Bildern 255, d.h., Pixel sind mit 8-Bit-Zahlen ausgedrückt.

Das Programm muss auch erkennen, wenn das gegebene Bild nicht dem Format entspricht oder syntaktische Fehler beinhaltet. In diesem Fall soll es eine Exception ausgeben (`throw runtime_error(…)`). Die mitgelieferten Tests prüfen solche Fälle.

Überprüfen: `./test.sh 1a` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task1a` starten.

Binärformat einlesen

Die Darstellung im Textformat ist besonders bei grösseren Bildern sehr ineffizient, da sie viel Speicherplatz benötigt.

> **Beispiel:** Die Zeichenkette "`255 0 0`" sieht in der binären Darstellung im Speicher so aus (nach Bytes gruppiert):
>
> ```
> 00100010 00110010 00110101 00110101 00100000 00110000 00100000 00110000 00100010
> ```
>
> Jedes Zeichen (Buchstaben, Ziffern, Leerzeichen) entsprechen einem Byte. Speichern wir aber die Zahlen binär ab, sieht es wie folgt aus:
>
> ```
> 11111111 00000000 00000000
> ```
>
> In diesem Beispiel benötigt die binäre Darstellung also nur einen Drittel des Speicherbedarfs (3 Bytes statt 9 Bytes) der Textdarstellung.

Abbildung 2: Ein Beispielbild, das im binären PPM-Format eingelesen wird (`img/03_stars.ppm`).

Wir werden in dieser Aufgabe auch ein grösseres Bild einlesen, das eine Aufnahme des Himmels zeigt (siehe Abb. 2).

Aufgabe 1.b) Binäres PPM einlesen

Erweitern Sie die Funktion `RGBImage::load image.cpp`, um sowohl das Format P3 (Text) als auch P6 (binär) zu unterstützen. Es gelten dieselben Annahmen über die Eingabe wie bisher.

Tipp 1: *Der Operator `<<` ist für Einlesen von binären Daten nicht geeignet. Benutzen Sie stattdessen die Funktion `in.read(…)`, um den ganzen Datenblock in einem einzigen Leseschritt einzulesen.*

Tipp 2: *Nach dem Einlesen des Headers muss zuerst ein Zeilenumbruch übersprungen werden,*

bevor die Pixeldaten eingelesen werden. Dies ist in der Voralge bereits durch `in >> std::ws` *implementiert.*

Überprüfen: `./test.sh 1b` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task1b` starten.

Aufgabe 2: Histogramme

Um ein Bild zu analysieren sind Histogramme ein nützliches Werkzeug. Sie ermöglichen es schnell zu prüfen, ob die Belichtung gut eingestellt ist. Anhand dieser Information können die Einstellungen der Aufnahme angepasst werden.

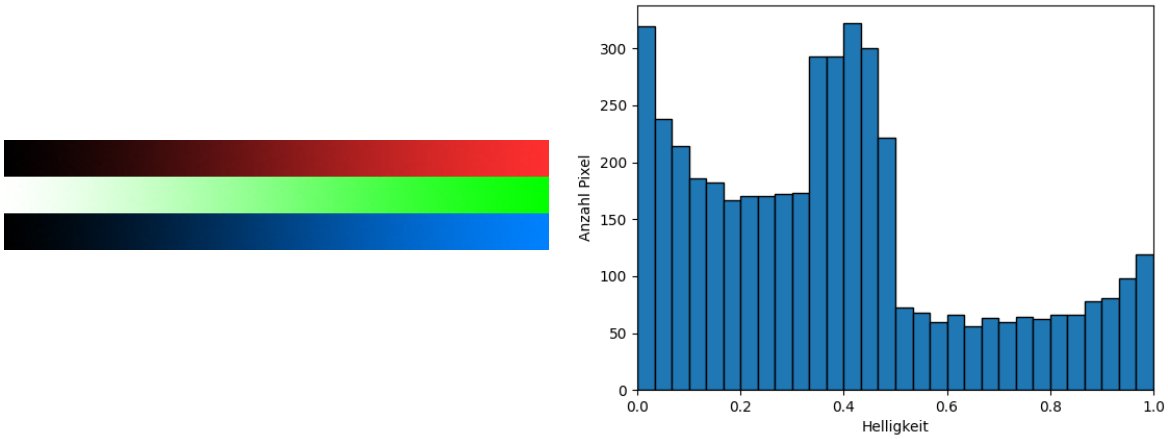


Abbildung 3: Histogramm eines Beispielbildes (`img/02_gradients.ppm`). Die meisten Pixel befinden sich im tieferen Helligkeitsbereich, da zwei der Balken bis ganz in schwarze Farbstufen verlaufen.

Helligkeit eines Pixels

Ein Histogramm wird aus den Helligkeitswerten der einzelnen Pixel berechnet. Üblicherweise wird die Helligkeit als ein gewichteter Mittelwert der drei Farbkanäle definiert, wobei sich die Gewichtung an der menschlichen Wahrnehmung orientierung soll.

Wir verwenden in unserem Fall eine vereinfachte Variante, bei der die drei Farbkanäle gleich gewichtet sind. Die Helligkeit eines Pixels h_{xy} an der Stelle (x, y) ist also wie folgt definiert:

$$h_{xy} = \frac{r_{xy} + g_{xy} + b_{xy}}{3}, \tag{1}$$

wobei r_{xy}, g_{xy} und b_{xy} die Werte der Komponenten Rot, Grün respektive Blau im Bereich $[0, 1]$ bezeichnen.

Aufgabe 2.a) Helligkeit berechnen

Implementieren Sie die folgende Funktion in `image.cpp` gemäss der Definition in Gleichung (1):

```
double RGBPixel::getBrightness() const;
```

Die Ausgabe ist eine Fliesskommazahl im Bereich $[0, 1]$.

Überprüfen: `./test.sh 2a` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task2a` starten.

Berechnung des Histogramms

Ein Histogramm besteht aus n Klassen (engl. *Bins*). Für jede Klasse i ist der Wert des Histogramms $H(i)$ die Anzahl Pixel, die in diese Klasse fallen. In unserem Fall bewegen sich die Helligkeitswerte h_{xy}

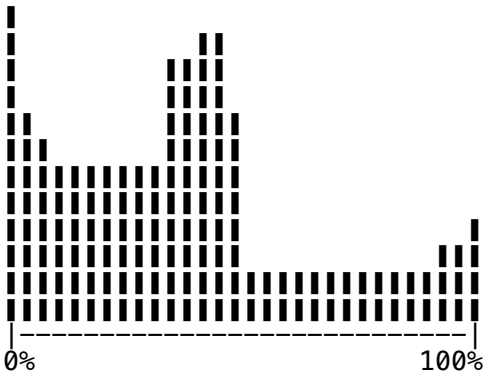


Abbildung 4: Eine Text-Darstellung in der Konsole des Histogramms in Abb. 3.

zwischen 0 und 1, weshalb die Klassen jeweils Helligkeitswerte in einem $\frac{1}{n}$ weiten Bereich abdecken.

Oder beispielhaft ausgedrückt: Wählen wir $n = 10$, müssen wir die Helligkeitswerte des ganzen Bildes in zehn Klassen aufteilen. Jeder Pixel wird dann einem der Bereiche $[0, 0.1)$, $[0.1, 0.2)$ usw. zugeordnet. Fällt ein Wert genau auf die Grenze zwischen zwei Bereichen, d.h., ein Pixel hat Helligkeit $h = 0.1$, soll er der höheren Klasse zugeordnet werden.

Wir können das Histogramm also wie folgt berechnen:

$$H(i) \mathrel{\widehat{=}} \text{Anzahl Pixel, für die gilt:} \quad \begin{cases} \frac{i}{n} \leq h_{xy} < \frac{i+1}{n} & \text{falls } 0 \leq i \leq n-2 \\ \frac{i}{n} \leq h_{xy} \leq \frac{i+1}{n} & \text{falls } i = n-1 \end{cases} \tag{2}$$

Aufgabe 2.b) Histogramm-Berechnung

Implementieren Sie die Funktion `process`, die als Eingabe ein Bild und eine Anzahl Klassen (Bins) akzeptiert:

```
Histogram::process(const RGBImage& img, int numBins);
```

Überprüfen: `./test.sh 2b` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task2b` starten.

Die berechneten Histogramme werden zwar von den Tests überprüft, sind aber ohne visuelle Darstellung nicht einfach zu interpretieren. Dazu implementieren Sie als nächstes eine Ausgabe zur Konsole.

Aufgabe 2.c) Histogramm-Ausgabe

Implementieren Sie die Funktion `Histogram::print(int height)`, sodass die Ausgabe wie in Abb. 4 aussieht (Beispiel für `height` = 12). Der Code für das Generieren der x-Achse ist vorgegeben und schreibt für jede Klasse eine Spalte.
Ergänzen Sie die Funktion so, dass in jeder Spalte ein Balken aus Blockzeichen ausgegeben wird, der dem Wert für die jeweilige Klasse entspricht. Die Skalierung der y-Koordinate soll linear sein und so erfolgen, dass die höchste Spalte genau `height` Blockzeichen hoch ist.
Die Balken sollen auf ganze Zeichen abgerundet werden. Das heisst, falls der eigentliche Wert in der Mitte eines Zeichens liegt, soll kein Block gezeichnet werden. *Beispiel:* in Abb. 4 ist der Wert der rechtesten 4.75 Balken hoch. Trotzdem werden nur 4 Balken gezeichnet.

Tipp: *Gehen Sie zeilenweise vor und prüfen Sie für jede Koordinate, ob ein Blockzeichen oder ein Leerzeichen ausgegeben werden soll.*

Überprüfen: `./test.sh 2c` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task2c` starten.

Die resultierenden Histogramme aus dieser Aufgabe könnten in der Realität verwendet werden, um die Belichtungseinstellung des Teleskops zu justieren (manuell oder automatisch), damit optimales Bildmaterial entsteht.

Aufgabe 3: Objekterkennung

In den letzten beiden Aufgaben entwickeln wir einen Algorithmus, der aus dem Bild die Position des hellsten Himmelskörpers ermittelt.

Aufgabe 3.a) Bestimmung eines hellsten Pixels

Implementieren Sie die folgende Funktion in `image.cpp`:

```
Coordinate RGBImage::findBrightestPixel() const;
```

Sie identifiziert einen hellsten Pixel im Bild gemäss der Helligkeits-Metrik, die Sie bereits in `RGBImage::getBrightness` implementiert haben. Falls es mehrere hellste Pixel im Bild gibt (z.B. einige komplett weisse Pixel), darf Ihre Lösung einen beliebigen Pixel zurückgeben.

Überprüfen: `./test.sh 3a` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task3a` starten.

Leider ist diese Funktion alleine noch nicht besonders nützlich. Vergleichen Sie den Ausgabewert Ihrer Lösung mit dem intuitiven Verständnis eines hellsten Objekts am Himmel: Stimmen die Positionen überein?

Das Problem ist, dass es im Bild einige komplett weisse Pixel, d.h. mit Wert $(255, 255, 255)$ gibt. Unter diesen möchten wir solche priorisieren, die auch von hellen Pixeln umgeben sind und so ein grösseres Objekt bzw. helleres Objekt bilden. Dies kann aber nicht erkannt werden, wenn nur einzelne Pixel analysiert werden. Im letzten (optionalen) Teil wird dieses Problem mithilfe eines Tricks gelöst, damit die eben implementierte Funktion nützlicher wird.

Zuerst brauchen wir eine Möglichkeit, verarbeitete Bilder von unserem Programm wieder zu speichern. So können wir die Resultate zusätzlich von Auge verifizieren und mögliche Fehlerursachen erkennen.

Aufgabe 3.b) Bildspeicherung

Implementieren Sie die folgende Funktion:

```
void RGBImage::write(ostream& out) const;
```

Benutzen Sie das binäre P6-Format von PPM [4], das unser Programm bereits einlesen kann.

Überprüfen: `./test.sh 3b` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task3b` starten.

Aufgabe 4: Bildverarbeitung durch Filter (optional)

Diese Aufgabe ist das letzte Stück zur Lösung und zeigt Ihnen ein weiteres Konzept aus dem Bereich der Bildverarbeitung. Das Lösen der Aufgabe ist optional.

Wenn wir zuerst einen Unschärfe-Filter über das Bild legen, bevor der hellste Pixel bestimmt wird, werden kleine Himmelskörper fast gänzlich verschwinden und nur grössere weiterhin hell bleiben. Auch Effekte wie Bildrauschen werden dadurch abgeschwächt.

Grafische Filter

Ein Filter kann auf ein Bild angewendet werden, um es zu verändern. Oft werden Unschärfefilter, auch Weichzeichnungsfilter, verwendet. Diese sind sowohl in der Gestaltung als auch in der Bildverarbeitung (z.B. zur Erkennung von Objekten) nützlich.

Wir werden uns zur Vereinfachung nur mit Filtern beschäftigen, die das Bild in Graustufen verarbeiten. Im Grundsatz funktioniert ein Unschärfefilter so, dass jeder Pixel als gewichtetes Mittel der umliegenden Pixel berechnet wird.

Wir bezeichnen die Grösse des Filters als r (Radius). So können wir die Helligkeit von Pixel $p'(x, y)$ im

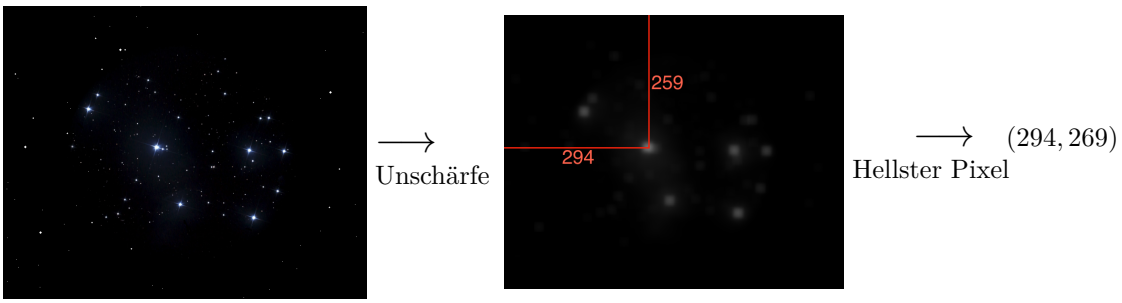


Abbildung 5: Der Algorithmus, der das hellste Himmelsobjekt mithilfe eines Zwischenschritts findet (am Beispiel von `img/03_stars.ppm`).

resultieren Bild wie folgt definieren:

$$p'(x,y) = \frac{1}{W} \sum_{\Delta x=-r}^{r} \sum_{\Delta y=-r}^{r} p(x+\Delta x, y+\Delta y) \cdot w(\Delta x, \Delta y) \tag{3}$$

wobei $p(x,y)$ für die Helligkeit des Pixels im ursprünglichen Bild steht und $w(\Delta x, \Delta y)$ für das Gewicht. Damit die totale Helligkeit nicht verändert wird, muss ganz am Ende durch W, die Summe aller Gewichte geteilt werden.

Ein Beispiel für $r = 1$ und $w(\Delta x, \Delta y) = 1$ ist in Abb. 6 dargestellt und durch die folgende Gleichung gegeben:

$$p'(x,y) = \frac{1}{9} \sum_{\Delta x=-1}^{1} \sum_{\Delta y=-1}^{1} p(x+\Delta x, y+\Delta y) \tag{4}$$

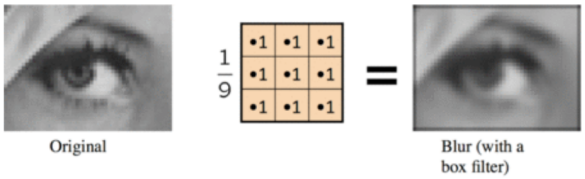


Abbildung 6: Ein einfacher Unschärfefilter (sog. Box-Filter). Jeder Pixel in der Ausgabe ist das Resultat des Mittelwerts des Pixels in der Eingabe und der 8 umliegenden Pixeln. Am Rand sind dunklere Pixel bemerkbar.
Quelle: Serena Yeung, `https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html`

Es ist möglich, dass in der Eingabe Pixel erscheinen, die ausserhalb des Bildes liegen und nicht exisieren, z.B. $(-1, 0)$. In diesem Fall sollen die Pixel einfach übersprungen werden für die Berechnung.

Aufgabe 4.a) Box-Blur

Implementieren Sie einen 3×3-Box-Filter wie in Gleichung (4). Die folgende Funktion in `filter.cpp` enthält eine Vorlage:

```
void applyBoxBlur(const RGBImage& src, RGBImage& dest);
```

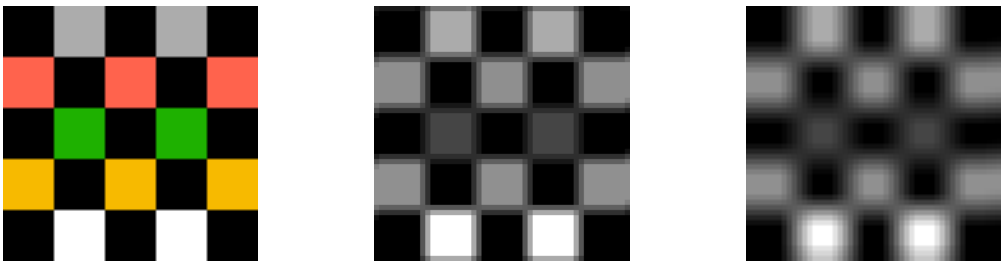
Nach der Ausführung des Tests finden Sie ihr Resultat unter `img/02_checkerboard_boxblur.ppm`.

Überprüfen: `./test.sh 4a` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task4a` starten.

Gauss'sche Unschärfe

Die Gauss'sche Unschärfe [1] ist eine von vielen Möglichkeiten, das gewichtete Mittel zu definieren. Sie liefert einen realistischeren und weicheren Effekt als die Box-Methode aus der letzten Aufgabe und ist ideal, um unser Ziel zu erreichen. Die folgende Gleichung definiert die Gauss-Funktion:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5}$$



(a) Original (b) 3×3-Box-Filter (c) Gauss-Filter mit $\sigma = 3$

Abbildung 7: Resultate verschiedener Unschärfe-Filter (in Graustufen) auf ein Beispielbild (`img/02_checkerboard.ppm`).

wobei x und y für den Abstand vom ursprünglichen Pixel stehen und σ ein Parameter ist, um die Stärke der Unschärfe einzustellen.

Das folgende Beispiel zeigt die Werte einer Gauss-Funktion mit $\sigma = 2$ in einer 5×5-Matrix, gerundet auf 3 Nachkommastellen:

$$G = \begin{bmatrix} 0.073 & 0.107 & 0.121 & 0.107 & 0.073 \\ 0.107 & 0.155 & 0.176 & 0.155 & 0.107 \\ 0.121 & 0.176 & 0.199 & 0.176 & 0.121 \\ 0.107 & 0.155 & 0.176 & 0.155 & 0.107 \\ 0.073 & 0.107 & 0.121 & 0.107 & 0.073 \end{bmatrix}$$

Die Berechnung des Gauss-Filters ist bereits in der folgenden Funktion implementiert:

```
double computeGaussian(int dx, int dy, double sigma);
```

Diese können Sie nutzen, um im nächsten Schritt den Gauss-Filter auf das Bild anzuwenden.

Aufgabe 4.b) Gauss'sche Unschärfe

Implementieren Sie die folgende Funktion in `filter.cpp`:

```
void applyGaussianBlur(const RGBImage& src, RGBImage& dest,
                       double sigma);
```

Das Resultat soll wie in Abb. 7 aussehen. Nach der Ausführung des Tests finden Sie ihr Resultat unter `img/02_checkerboard_gaussian.ppm`. Sie können es öffnen, um das Resultat zu inspizieren und mögliche Fehler zu finden.

Tipp 1: *Die Variablen sind im Codeskelett bereits vorgegeben.*

Tipp 2: *Für die Berechnung ist es zu empfehlen, die Pixeldaten in Fliesskommazahlen statt in ganzen Zahlen zu speichern. Vergessen Sie am Ende nicht, die Ausgabewerte wieder von* 0 *bis* 255 *zu skalieren.*

Überprüfen: `./test.sh 4b` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task4b` starten.

Hellstes Himmelsobjekt finden

Jetzt sind wir bereit, den verbesserten Algorithmus zusammenzusetzen. Es sind keine weitere Schritte mehr nötig zu implementieren.

Aufgabe 4.c) Hellster Himmelskörper, zweiter Versuch

Testen Sie Ihre Lösung. Sie können auch weitere Bilder als die mitgelieferten verarbeiten, sofern Sie sie zuerst in das PPM-Format konvertieren (mit Picsi).

Überprüfen: `./test.sh 4c` in Konsole oder `astro_test.exe` mit Parameter `--gtest_filter=Task4c` starten.

Literatur

[1] Gaussian blur - Wikipedia. https://en.wikipedia.org/wiki/Gaussian_blur.

[2] GoogleTest User's Guide. https://google.github.io/googletest/.

[3] The PNM format. https://netpbm.sourceforge.net/doc/pnm.html.

[4] PPM format specification. https://netpbm.sourceforge.net/doc/ppm.html.