1 Patch Basiertes Inpainting

In dieser Übung implementieren wir Patchbasiertes Inpainting nach Criminisi et. al. Hierbei setzen wir in diesem Übungsblatt noch nicht da gesamte Verfahren um, sondern eine vereinfachte Form. In der kommenden Übung befassen wir uns dann mit dem vollständigen Verfahren.

Das Originalpaper finden Sie hier: https://www.irisa.fr/vista/Papers/2004_ip_criminisi.pdf.

In ihrem codeverzeichnis finden Sie die Dateien image.jpg und mask.png.

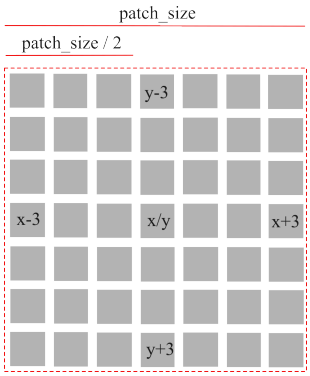
1.1 Daten einlesen

Lesen Sie die Dateien image.jpg (als rgb-Bild) und mask.png (als 8-bit Graustufen Bild) ein.

Spezifikation: Die Maske weist die Werte 0 (Pixel definiert) und 255 (Pixel nicht definiert) auf, andere Werte sind ein Fehler. Die Maske benötigen wir als numpy array mit unsigned int 8. Das Bild benötigen wir als numpy array mit float32 und 3 Farbkanälen.

Löschen Sie die Pixel im Farbbild, die laut Maske nicht definiert sein sollten.

Koordinaten: Patches sind quadratische Bereiche von Pixeln mit einer Kantenlänge von `patch_size`. Die `patch_size` ist immer ungerade, für diese Übung 7. Die Koordinaten eines Patches beziehen sich immer auf den mittleren Pixel des Patches und erstrecken sich von $-$patch_size/2 bis +patch_size/2. D.h. der Patch (10/5) geht beispielsweise von (7/2) bis (13/8).



Wir machen uns das Leben später deutlicher einfacher, wenn wir fordern, dass alle Pixel im Umkreis von *patch_size*/2 um den Rand des Pixels definiert sind. D.h. der Rand des Bildes beinhaltet keine undefinierten Pixel.

a) Überlegen Sie sich, was für die Eingabe getestet werden sollte. Einige Ideen: 1. Bild und Maske sollten die gleiche Auflösung haben. 2. Die Maske sollte als 8-Bit Graustufenbild vorliegen und nur die Werte 0 und 255 beinhalten. Das Bild sollte als rgb Bild vorliegen. Wenn Sie die Einschränkung am Rand des Bilden (s.o.) nutzen möchten, könnte das ebenfalls bereits getestet werdne.

b) Ergänzen Sie Unittests für das Laden von Maske und Bild.

c) Implementieren Sie die Ladefunktion.

d) Implementieren Sie eine Funktion, die Bild und Maske anzeigt. Hierbei sollte die Maske halbtransparent oder als Silhouette über dem Bild angezeigt werden.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.2 Fillfront extrahieren

Als nächstest benötigen wir eine Funktion `extract_fillfront`, die aus dem Bild die Fillfront extrahiert. Die Funktion akzeptiert eine Maske und gibt als Resultat eine Liste zurück. Die Liste enhält die Koor-

dinaten (x, y) aller Pixel, die Teil der Fillfront sind. Ein Pixel ist Teil der Fillfront, wenn er selber nicht definiert ist, aber ein direkt angrenzender Pixel (4-Nachbarschaft) definiert ist.

a) Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie die Funktion `extract_fillfront`.

d) Implementieren Sie Funktionalität, um die Fillfront im Bild zu visualisieren (als Silhouette, farbig markierte Pixel o.ä.). War die Visualisierung in Ihrer Teststrategie enthalten?

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.3 Fillfront Priorisieren

Nun bewerten wir die Einträge in der Fillfront gemäß der Formen von Crimini et. al. Hierbei nutzen wir nur den Konfidenz-Term (der Data-Term kommt im nächsten Übungsblatt). Die entsprechende Funktion `compute_confidence` akzeptiert eine Maske, ein array mit den bisherigen Konfidenzen sowie die Koordinaten des zu berechnenden Pixel. Sie liefert die Konfidenz (float) zurück.

a) Überlegen Sie sich, wie diese Funktion getestet werden kann. Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie eine Funktion, die für jeden Pixel der Fillfront die Konfidenz berechnet und den Pixel mit der höchsten Konfidenz zurückgibt.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.4 Kostenfunktion

Die nächste Komponente ist eine Kostenfunktion: wir berechnen den L_2 Abstand zwischen den Pixeln zweier Patches. Dabei hat ein Patch (der Ziel-Patch) einige undefinierte Pixel. Der Quellpatch hat nur definierte Pixel. Die Kostenfunktion kann als Formel geschrieben werden als:

$$L_2 := \sqrt{\sum_{i \in Patch} \begin{cases} (P^i_{source} - P^i_{target})^2, & \text{wenn } P^i_{target} \text{ definiert} \\ 0, & \text{sonst} \end{cases}}$$

a) Überlegen Sie sich, diese Funktion getestet werden kann. Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie eine Funktion.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.5 Auswahl des besten Patches

Mithilfe der Kostenfunktion können wir nun die in Frage kommenden Quellpatches bewerten. Die Funktion `find_best_patch` akzeptiert als Argumente: das Bild, die Maske, sowie die Koordinaten eines Zielpatches. Sie bewertete alle vollständig definierten Patches in der Quellregion gemäß der Kostenfunktion. Die liefert die Koordinaten des Quellpatches mit der niedrigsten Kostenfunktion zurück.

a) Überlegen Sie sich, dies nun getestet werden kann. Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie eine Funktion `find_best_patch`.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.6 Einen Inpainting Schritt ausführen

Nun können wir einen Inpaintingschritt ausführen. Die Funktion `perform_inpainting_step` akzeptiert das Bild, die Maske, die Konfidenz, die Koordinaten eines Quellpatches und die Koordinaten eines Zielpatches. Sie führt ein Update auf Bild, Maske und Konfidenzarray aus.

a) Überlegen Sie sich, wie dies nun getestet werden kann. Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie eine Funktion `perform_inpainting_step`. Diese besteht aus drei Teilschritten: im Bild werden die Pixelwerte von Quellpatch nach Zielpatch kopiert. In der Maske werden alle Pixel des Zielpatches auf "definiert"gesetzt. Im Konfidenz-Array werden die Konfidenzen für alle Pixel des Zielpatches neu berechnet.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt:

1.7 Iteratives Inpainting

Fast geschafft: nun können wir die Komponenten zusammensetzen. In einer Schleife berechnen wir die Füllfront, wählen den besten Zielpatch aus, wählen dazu passend den besten Quellpatch aus und führen den Inpainting-Schritt aus. Dann iterieren wir, bis die Fillfront leer ist (gleichbedeutend mit: keine Pixel mehr undefiniert).

a) Überlegen Sie sich, wie der Gesamtalgorithmus getestet werden kann. Beschreiben Sie Ihre Teststrategie.

b) Implementieren Sie die Tests.

c) Implementieren Sie eine Funktion `inpaint_image`.

d) Implementieren Sie eine Funktion, die das Zwischenergebnis jedes Inpainting Schrittes als Bild abspeichert. Erzeugen Sie für die Beispiele im Verzeichnis Animationen, die Ihr Inpainting visualisieren. **Bitte checken Sie die Animationen nicht ins Github Repository ein.** Sie kann in der Übung gezeigt werden.

Für die Bearbeitung der Aufgabe habe ich folgende Zeit benötigt: