

# **Morphing mit Beier-Neely**

## **Digitale Bildverarbeitung WS2023/24**

Michael Eggers, Johann Rittenschober, Kevin

5. Januar 2024



## **Erklärung**

Hiermit erklären wir, dass die vorliegenden Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet wurden.

Michael Eggers, Johann Rittenschober, Kevin

München, 5. Januar 2024

Matrikelnummer: 00322614

Studiengruppe: Master Informatik VZ/TZ



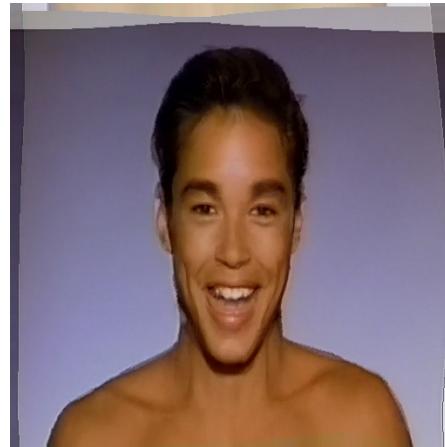
(a) Quellbild



(b) Zielbild



(c) 50% Kreuzblende



(d) 50% Beier-Neely Morph

Abbildung 1.: Gegenüberstellung: einfache Kreuzblende und Beier-Neely Morphing

## Zusammenfassung

Der Übergang von einem Bild in ein Anderes kann durch verschiedene Effekte erreicht werden. Einer der bekanntesten ist die sogenannte Kreuzblende (engl. cross dissolve). Dabei wird jeder Pixel des Quellbildes sukzessive um  $1 - \frac{i}{numIterations}$  abgeschwächt und dafür jeder Pixel des Zielbildes um  $\frac{i}{numIterations}$  multipliziert (verstärkt). Das Resultat aus der Addition dieser beiden Operationen ergibt den Effekt der eben genannten Kreuzblende (1c). Der Übergang ist deutlich wahrnehmbar. Eine Verbesserung erreicht man durch die Verzerrung beider Bilder in die Form des jeweilig anderen (1a nach 1b und umgekehrt). Danach wird wie gehabt die Kreuzblende angewendet. Die Resultierende Animation kann den Eindruck erwecken als verwandle sich das Quell in das Zielbild. Beier und Neely [1] entwickelten dafür einen Feature-basierten Algorithmus um diesen Effekt zu erzielen. Er kam in dem Michael Jackson Musikvideo Black or White zum Einsatz [2].

# **Inhaltsverzeichnis**

<b>1. Festlegung der Features</b>	<b>6</b>
<b>2. Warping der Bilder</b>	<b>7</b>
<b>3. Komposition</b>	<b>9</b>
<b>A. Software</b>	<b>10</b>
A.1. Installation . . . . .	10
<b>Literaturverzeichnis</b>	<b>11</b>

# **Abbildungsverzeichnis**

1.	Gegenüberstellung: einfache Kreuzblende und Beier-Neely Morphing . . . . .	3
1.1.	Setzen eines Linienpaars . . . . .	6
1.2.	Finale Menge an Linienpaaren . . . . .	6
3.1.	Quell- zu Ziel warps . . . . .	9
3.2.	Ziel- zu Quell warps . . . . .	9
3.3.	Finale Komposition . . . . .	9

# 1. Festlegung der Features

Wie eingangs beschrieben handelt es sich um einen Feature basierten Algorithmus. Das heißt, dass die Merkmale eines Objekt im Bild, welches transformiert werden soll, zunächst erfasst werden müssen. Beier und Neely nutzen dazu eine Liste aus gerichteten Linienspaaren: Einer Linie im Quellbild wird genau eine Linie im Zielbild zugeordnet. Dabei werden die Linienspaare so platziert, sodass sie ein Merkmal in den Bildern beschreiben. Zum Beispiel werden die Haaransätze der beiden Personen in Quell- und Zielbild als Merkmale deklariert: Das Linienspaar wird dementsprechend gesetzt, siehe 1.1.

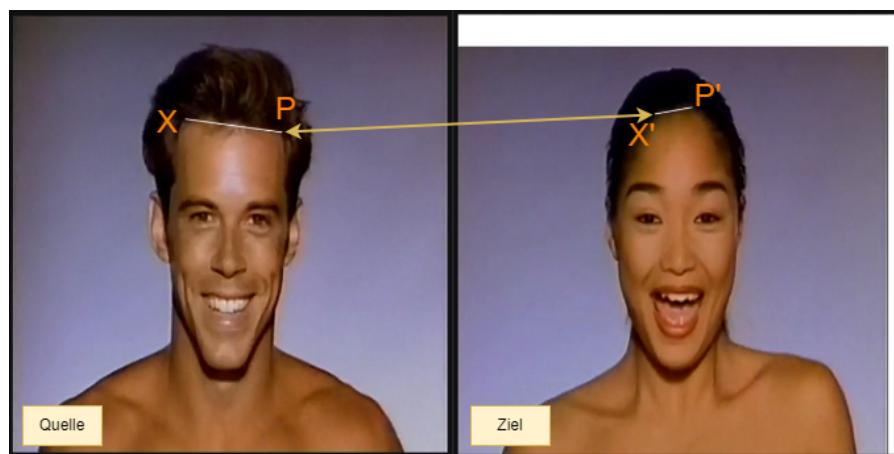


Abbildung 1.1.: Setzen eines Linienspaars

Es ist wichtig zu beachten, dass die Linienspaare gerichtet sind. Die Linien besitzen also sowohl Anfangs- als auch Endpunkt. Abbildung 1.2 zeigt die Linienspaare, welche für das Resultat in 1d verantwortlich waren.

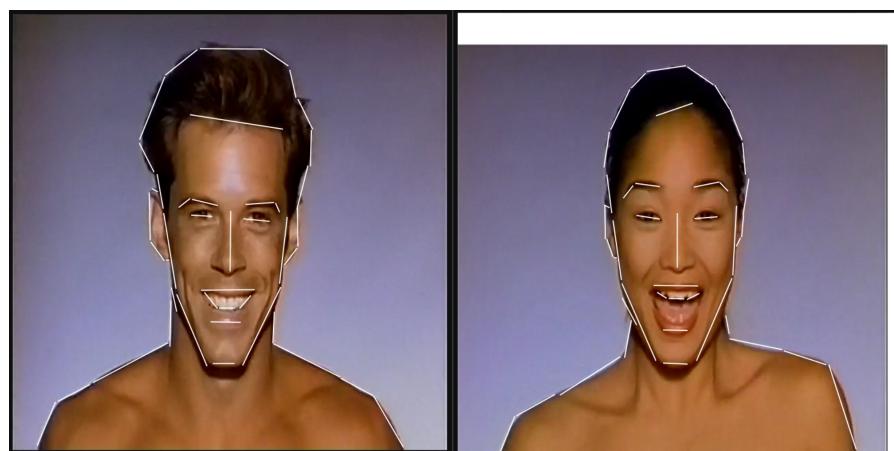


Abbildung 1.2.: Finale Menge an Linienspaaren

## 2. Warping der Bilder

Sobald die Linienpaare gesetzt wurden, kann der eigentliche Algorithmus gestartet werden. Die Pixel des Quellbildes werden in Richtung der Linienpositionen des Zielbildes verschoben. Und die Pixel des Zielbildes wiederum werden in Richtung der Linienpositionen des Quellbildes verschoben. Dies ist ein iterativer Prozess. Je mehr Iterationen gewählt werden, desto fließender ist die Verzerrung. Fuer jeden Pixel im Bild wird jede Linie in Betracht gezogen. Alle Linien haben also einen gewissen Einfluss auf das Warping eines Pixels. Die Laufzeitkomplexitaet des Beier-Neely Algorithmus errechnet sich aus  $\mathcal{O}(numPixels \cdot numLines)$ . Der Einfluss einer Linie auf einen Pixel kann mit einem von Beier-Neely entwickelten Gewicht festgelegt werden [1]:

$$weight = \left( \frac{length^p}{a + dist} \right)^b \quad (2.1)$$

a, b, p sind Konstanten, die einmalig gesetzt werden.

- **length:** Laenge der Linie
- **dist:** senkrechter Abstand des Pixels zur Linie
- **a:** Bestimmt den Einfluss der senkrechten Distanz auf das Gewicht. Wenn a nahe 0 ist, so werden Pixel, die auf der Linie liegen mit einem Gewicht von nahezu  $\infty$  beeinflusst.
- **b:** Kontrolliert den Fall-Off der Linien. Ist b nahe 0, tragen alle Linien, gleich Ihrer senkrechten Distanz zum Pixel, gleichmaessig zur Verzerrung des jenes Pixels bei. Grosses Werte (es wird nicht erwartet, was gross ist), fuehren zu einem hoeheren Gewicht, je naeher der Pixel sich an der jeweiligen Linie befindet.
- **p:** Regelt, ob sich die Linienlaenge auf das Gewicht auswirkt. Bei einem Wert von 0 wird  $length^p$  offensichtlich 0 und das Gewicht ist unabhaengig zur Linienlaenge.

Wie die Parameter festgelegt werden kommt auf die in den Bildern vorhandenen Features an und erfordert in der Regel ein wenig ausprobieren. Beispielsweise haben sich die Werte  $a = 0$ ,  $b = 3.1$  und  $p = 0$  fuer das Bilderset aus Abbildung 1.2 in unseren Tests bewahrt. Die Linien koennen linear interpoliert werden, wie Abbildung ?? zeigt.

Durch diese Art der Interpolation entsteht jedoch bei Rotationen ein Fehler, wie man in ?? sehen kann: Die linear interpolierten Linien nehmen faelschlicherweise an Laenge ab. In unseren praktischen Tests hat sich dieses Problem als nicht gravierend herausgestellt.

Fuer unsere Implementierung nutzen wir den von Beier und Neely mitgelieferten Pseudocode. Unsere Implementierung:

```
1 for (uint32_t y = 0; y < destImage.m_Height; y++) {  
2     for (uint32_t x = 0; x < destImage.m_Width; x++) {  
3         glm::vec2 X = glm::vec2(x, y);  
4         glm::vec2 DSUM = glm::vec2(0.0, 0.0);  
5         float weightsum = 0;  
6         for (uint32_t i = 0; i < destLines.size(); i++) {  
7             Line& destLine = destLines[i];  
8             ...  
9         }  
10    }  
11}
```

```

9     Line& srcLine = sourceLines[i];
10    Line interpolatedLine = InterpolateLinesLinear(destLine, srcLine, pct);
11
12    glm::vec2 P = destLine.a.pos;
13    glm::vec2 Q = destLine.b.pos;
14    glm::vec2 srcP = interpolatedLine.a.pos;
15    glm::vec2 srcQ = interpolatedLine.b.pos;
16    glm::vec2 PX = X - P;
17    glm::vec2 PQ = Q - P;
18    float PQlength = glm::length(PQ);
19    float u = glm::dot(PX, PQ) / (PQlength * PQlength);
20    float v = glm::dot(PX, Perpendicular(PQ)) / PQlength;
21    glm::vec2 srcPQ = srcQ - srcP;
22    glm::vec2 srcX = srcP + u * srcPQ + (v * Perpendicular(srcPQ) / glm::
23        length(srcPQ));
24    glm::vec2 D = srcX - X;
25    float dist = Distance(u, v, P, Q, X);
26    float weight = glm::pow(glm::pow(PQlength, p) / (a + dist), b);
27    DSUM += D * weight;
28    weightsum += weight;
29 }
30 glm::vec2 srcX = X + DSUM / weightsum;
31 if ((uint32_t)srcX.x > destImage.m_Width - 1) srcX.x = float(destImage.
32     m_Width - 1);
33 if ((uint32_t)srcX.y > destImage.m_Height - 1) srcX.y = float(destImage.
34     m_Height - 1);
35 if ((uint32_t)srcX.x < 0) srcX.x = 0.0f;
36 if ((uint32_t)srcX.y < 0) srcX.y = 0.0f;
37
38     glm::ivec3 sourcePixel = sourceImage((uint32_t)srcX.x, (uint32_t)srcX.y);
39
40     unsigned char* newPixel = image.m_Data + (image.m_Channels * (y * image.
41         m_Width + x));
42     newPixel[0] = sourcePixel.r;
43     newPixel[1] = sourcePixel.g;
44     newPixel[2] = sourcePixel.b;
45
46 } // ! pixel row
47 } // ! pixel col

```

Listing 2.1: Beier-Neely in C++

### 3. Komposition

Sind die beiden Bilder erst einmal fertig verzerrt worden, so werden sie nun, wie eingangs beschrieben, durch eine Kreuzblende zusammengefügt. Zu beachten ist, dass das Zielbild in Richtung des Quellbildes gewarpt wurde. Die Bildsequenzen für die beiden Bilder sehen dementsprechend folgendermassen aus:

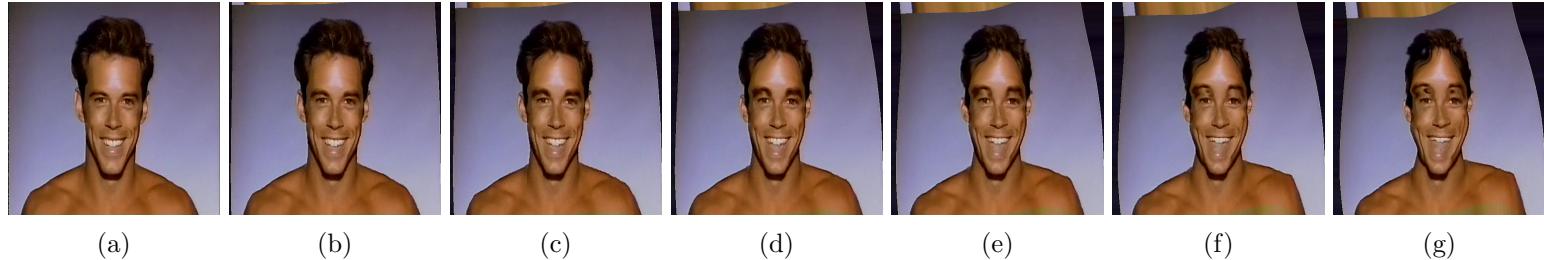


Abbildung 3.1.: Quell- zu Ziel warps

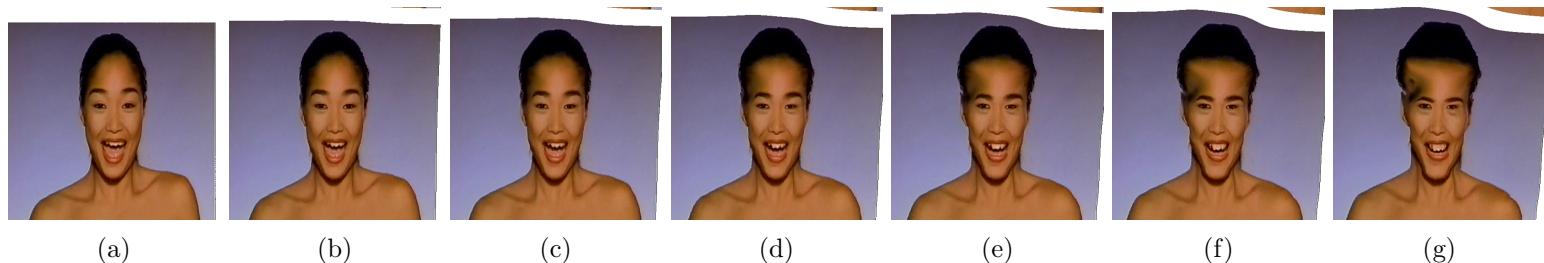


Abbildung 3.2.: Ziel- zu Quell warps

Die Sequenz in Abbildung 3.2 muss zunaechst noch in ihrer Reihenfolge geändert werden bevor die Kreuzblende angewendet wird.

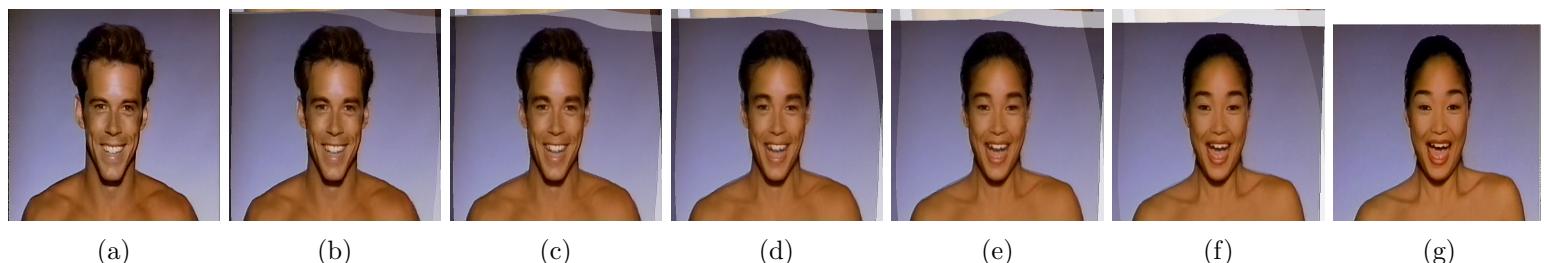


Abbildung 3.3.: Finale Komposition

# A. Software

Fuer den Kurs haben wir eine Beispielanwendung, PowerMorph, erstellt. Fuer das Bauen wird ein CMake file bereitgestellt. Die Software basiert auf OpenGL 3.0 und ist somit auch auf MacOS lauffaehig (zwar hat Apple den Support von OpenGL eingestellt, allerdings werden Anwendungen, welche bis einschl. maximal Version 4.1 von OpenGL verwenden, nach wie vor unterstuetzt). Ausserdem nutzen wir folgende externe Bibliotheken:

- **SDL2** als Abstraktion zum Betriebssystem fuer Fenster und OpenGL context Erstellung. <https://github.com/libsdl-org/SDL>
- **STB image/image write**: Lesen/Schreiben von Bilddateien. <https://github.com/nothings/stb>
- **GLM**: Mathematik Bibliothek, die gut mit OpenGL zusammenarbeitet. <https://github.com/g-truc/glm>
- **Dear ImGUI**: Immediate Mode GUI Bibliothek fuer den Editor.
- **tinyfiledialogs**: Betriebssystemunabhaengige Bibliothek fuer Window-Messages, Oeffnen/Speichern Dialoge.
- **GIF writer by Charlie Tangora**: Speichern der gerenderten Sequenzen als GIF. <https://github.com/charlietangora/gif-h>

## A.1. Installation

# Literaturverzeichnis

- [1] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *SIGGRAPH Comput. Graph.*, 26(2), 1992.
- [2] Ian Failes. Cartoon brew. <https://www.cartoonbrew.com/vfx/oral-history-morphing-michael-jacksons-black-white-144015.html>. Accessed: 11 02, 2023.