

Morphing mit Beier-Neely

Digitale Bildverarbeitung WS2023/24

Michael Eggers, Johann Rittenschober

8. 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

München, 8. Januar 2024

Matrikelnummer: 00322614

Studiengruppe: Master Informatik VZ/TZ



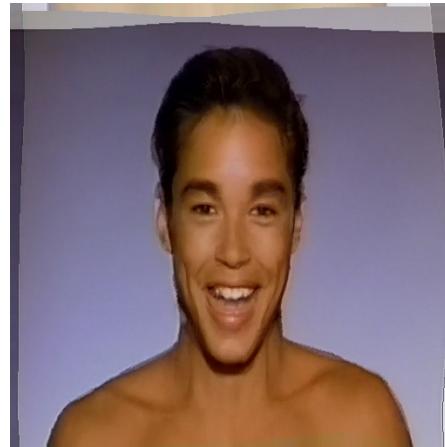
(a) Quellbild



(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

1. Festlegung der Features	6
2. Warping der Bilder	7
3. Komposition	10
A. Software	12
A.1. Verwenden von MagicMorph	12
A.2. Ausblick	12
Literaturverzeichnis	15

Abbildungsverzeichnis

1.	Gegenüberstellung: einfache Kreuzblende und Beier-Neely Morphing	3
1.1.	Setzen eines Linienpaars	6
1.2.	Finale Menge an Linienpaaren	6
2.1.	Lineare Interpolation zwischen Ziellinie \overline{PQ} und Quelllinie $\overline{PQ'}$	8
2.2.	Transformation des Pixels \mathbf{X} nach \mathbf{X}'	8
3.1.	Quell- zu Ziel warps	10
3.2.	Ziel- zu Quell warps	10
3.3.	Finale Komposition	11
A.1.	MagicMorph	13

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 Linienpaaren: Einer Linie im Quellbild wird genau eine Linie im Zielbild zugeordnet. Dabei werden die Linienpaare 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 Linienpaar wird dementsprechend gesetzt, siehe 1.1.

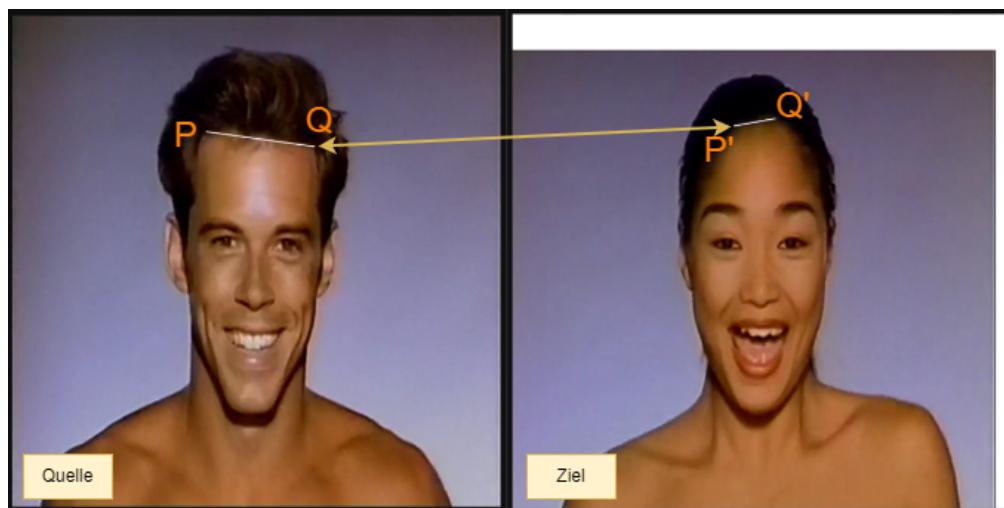


Abbildung 1.1.: Setzen eines Linienpaars

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

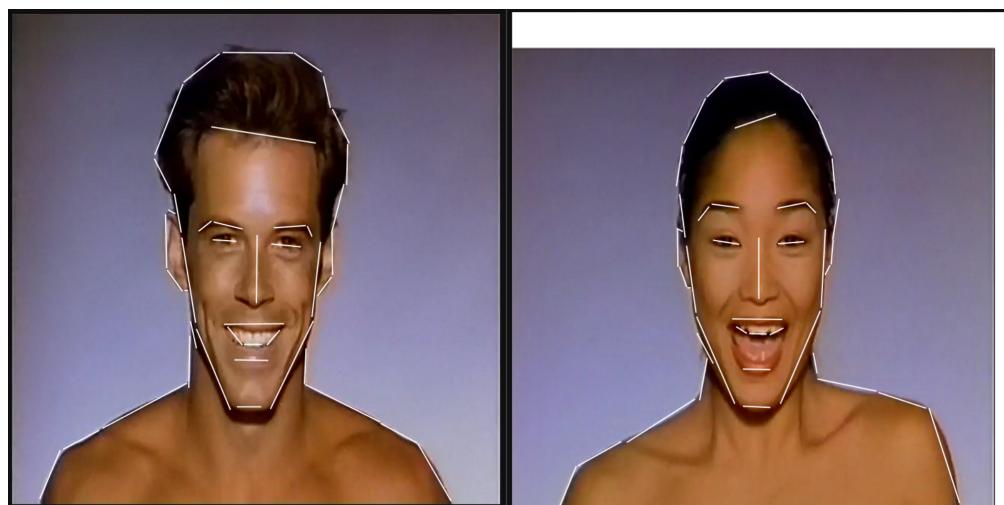


Abbildung 1.2.: Finale Menge an Linienpaaren

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. Für jeden Pixel im Bild wird jede Linie in Betracht gezogen. Alle Linien haben also einen gewissen Einfluss auf das Warping eines Pixels. Die Laufzeitkomplexität des Beier-Neely Algorithmus errechnet sich aus $\mathcal{O}(numPixels \cdot numLines)$. Der Einfluss einer Linie auf einen Pixel kann mit einem von Beier und 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:** Länge der Linie
- **dist:** senkrechter Abstand des Pixels zur Linie (der kürzeste Abstand von Punkt zu 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 ∞ beeinflusst.
- **b:** Kontrolliert den Fall-Off der Linien. Ist b nahe 0, tragen alle Linien, gleich ihrer senkrechten Distanz zum Pixel, gleichmäßig zur Verzerrung des jenes Pixels bei. Große Werte (es wird nicht erwähnt, was groß ist), führen zu einem höheren Gewicht, je näher der Pixel sich an der jeweiligen Linie befindet.
- **p:** Regelt, ob sich die Liniengröße auf das Gewicht auswirkt. Bei einem Wert von 0 wird $length^p$ offensichtlich 1 und das Gewicht ist unabhängig von der Liniengröße.

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 $a = 0$, $b = 3.045$ und $p = 0$ für das Bilderset aus Abbildung 1.2 in unseren Tests bewährt. Die Linien können linear interpoliert werden, wie Abbildung 2.1 zeigt. Durch diese Art der Interpolation entsteht jedoch bei Rotationen ein Fehler, wie man in 2.1 sehen kann: Die linear interpolierten Linien nehmen fälschlicherweise an Länge ab. In unseren praktischen Tests hat sich dieses Problem als nicht gravierend herausgestellt. Durch die Linienpaare lässt sich ein Pixel \mathbf{X} im Zielbild einem Pixel \mathbf{X}' zuordnen (2.2), da durch das Paar eine Transformation festgelegt werden kann [1]. Für unsere Implementierung nutzen wir den von Beier und Neely mitgelieferten Pseudocode.

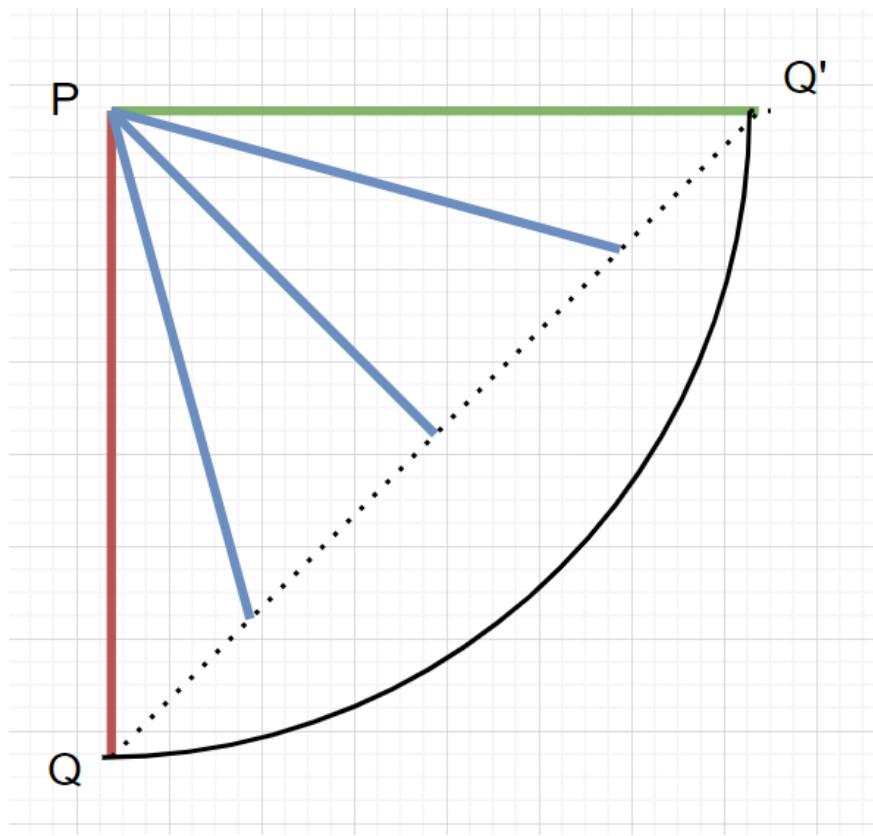


Abbildung 2.1.: Lineare Interpolation zwischen Ziellinie \overline{PQ} und Quelllinie $\overline{PQ'}$

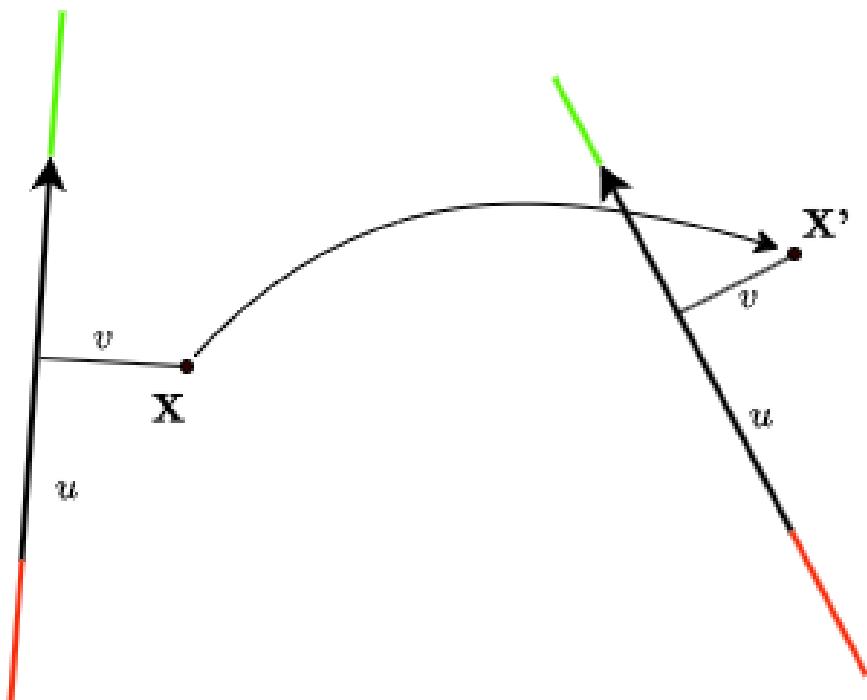


Abbildung 2.2.: Transformation des Pixels X nach X'

```

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
8              Line& destLine = destLines[i];
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:

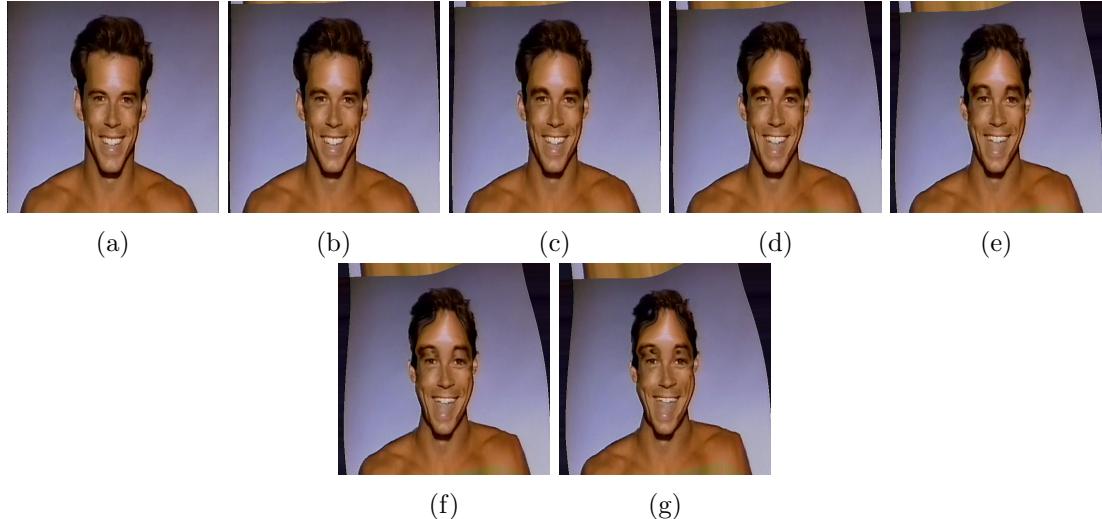


Abbildung 3.1.: Quell- zu Ziel warps

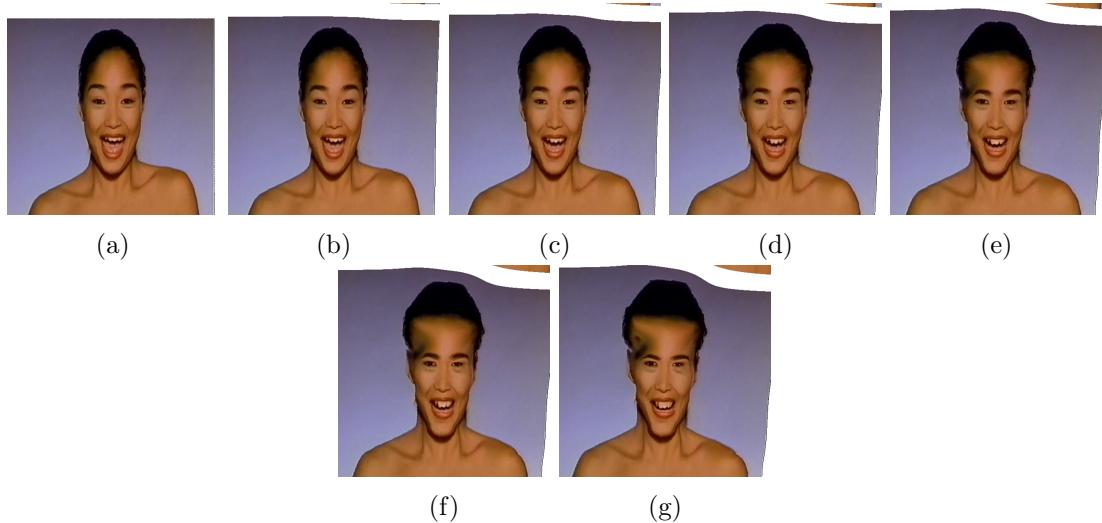
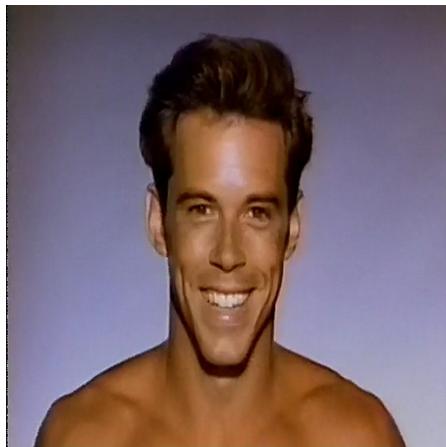
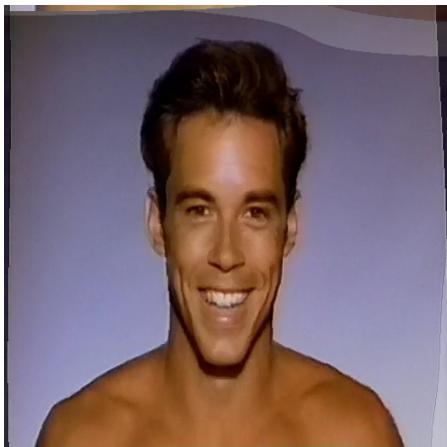


Abbildung 3.2.: Ziel- zu Quell warps

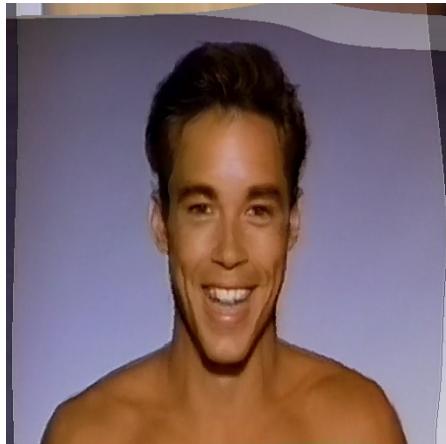
Die Sequenz in Abbildung 3.2 muss zunächst noch in ihrer Reihenfolge geändert werden bevor die Kreuzblende angewendet wird.



(a)



(b)



(c)



(d)



(e)



(f)



(g)

Abbildung 3.3.: Finale Komposition

A. Software

Für den Kurs haben wir eine Beispielanwendung, MagicMorph, erstellt. Ein CMake file bereitgestellt, um Projektdateien für die jeweilige IDE bzw. ein Makefile zu erzeugen. Die Software basiert auf OpenGL 3.0 und ist somit auch auf MacOS lauffähig (zwar hat Apple den Support von OpenGL eingestellt, allerdings werden Anwendungen, welche bis einschl. maximal Version 4.1 von OpenGL verwenden, nach wie vor unterstützt). Außerdem nutzen wir folgende externe Bibliotheken:

- **SDL2** als Abstraktion zum Betriebssystem für 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 für den Editor.
<https://github.com/ocornut/imgui>
- **tinyfiledialogs**: Betriebssystemunabhängige Bibliothek für Dialogfenster (Nachrichten an Anwender, öffnen, speichern, etc.)
<https://sourceforge.net/projects/tinyfiledialogs/>
- **GIF writer by Charlie Tangora**: Speichern der gerenderten Sequenzen als GIF.
<https://github.com/charlietangora/gif-h>

A.1. Verwenden von MagicMorph

Die Bedienung ist weitestgehend selbsterklärend. Im **Source**-Fenster wird eine Linie für ein Feature gezogen. Per Mausklick wird der Fußpunkt der Linie gesetzt, ein weiterer Mausklick vervollständigt diese. Dabei ist die Richtung der Linie durch den letzten Klick gegeben. Danach muss im **Destination**-Fenster eine weitere Linie erzeugt werden, um das Paar zu komplettieren. Dieses Vorgehen kann so lange wiederholt werden, wie gewünscht. Eine bereits gesetzte Linie kann immer mit der Tastenkombination CTRL+Z rückgängig gemacht werden. Ist man mit der Definition der Features zufrieden, können mit einem Klick auf den **MAGIC!**-Button die Morphs von Quell- und Zielbild generiert und danach überblendet werden. Je nach Auflösung des Bilderpaars und der Anzahl gesetzter Linien kann dieser Vorgang ein wenig dauern. Ist die Berechnung abgeschlossen, so wird ein neues **Result**-Fenster geöffnet. Dort lässt sich das Resultat begutachten. Die Parameter **a**, **b** und **p**, um das Gewicht einer Linie zu bestimmen, lassen sich mit den Schiebereglern im **Control Panel** festlegen.

A.2. Ausblick

Die Interpolation der Linien erfolgt linear für die Start- und Endpunkte. Wie beschrieben werden Rotationen durch diese Weise skaliert, auch wenn die Länge des Linienpa-

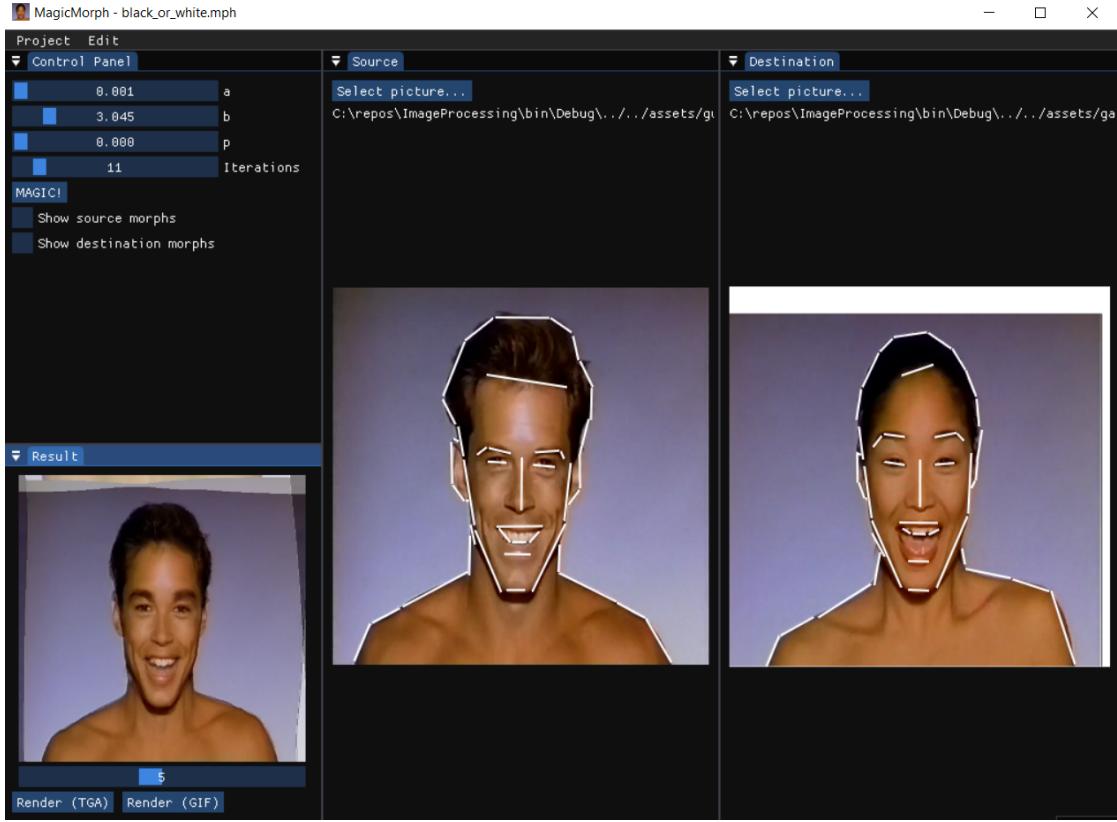


Abbildung A.1.: MagicMorph

res jeweils gleich bleibt. Beier und Neely beschreiben [1], dass eine weitere Möglichkeit zur Interpolation folgendermaßen aussieht: Die Mittelpunkte der beiden Linien und deren Orientierung werden interpoliert. Darauf wird die Länge der resultierenden Linie ebenfalls interpoliert. Denkbar wäre hierbei die Nutzung von Quaternions, welche die Orientierungen der beiden Linien repräsentieren. Das Erzeugen aus Winkel und Vektor wird durch **GLM** unterstützt:

```
1 glm::quat qSource = glm::angleAxis(theta, glm::vec3(0, 0, 1));
```

Listing A.1: Quaternions in GLM

q_{Source} repräsentiert nun eine Drehung um die Z-Achse (unsere Linien werden im R2 platziert, wodurch die 3. Dimension hinzugezogen wird, um einen weiteren Freiheitsgrad zu erlangen).

Den Winkel θ erhält man durch:

$$\theta = \arcsin \left(\frac{y}{\|\mathbf{v}\|} \right)$$

wobei y die 2. Komponente des Vektors \mathbf{v} ist, welcher die Linie in Ziel- bzw. Quellbild repräsentiert. Die Interpolation zwischen zwei Quaternions erfolgt schließlich durch:

```
1 glm::quat qInterpolated = glm::slerp(qSource, qDest, 0.5);
```

Listing A.2: Spherical interpolation zwischen zwei Quaternions

In Listing A.2 wird die Sphärische Interpolation zwischen zweier Quaternions bei 50% berechnet. `glm::slerp` sorgt dafür, dass der kürzeste Pfad zwischen den beiden Orientierungen genommen wird. Um nun die rotierte (interpolierte) Linie zu bekommen, wendet man den Quaternion auf die Start- und Endpunkte der Linie an:

```
1     glm::vec3 aInterpolated = qInterpolated * source.a;
2     glm::vec3 bInterpolated = qInterpolated * source.b;
3     Line interpolatedLine = Line(
4         aInterpolated.x, aInterpolated.y,
5         bInterpolated.x, bInterpolated.y
6     );
7 
```

Listing A.3: Rotation der Quelllinie durch einen Quaternion

Nun kann der Pixel **X** (Pixelposition relativ zur vom Anwender gezogenen Linie) nach **X'** (Pixelposition relativ zur interpolierten) Linie transformiert werden.

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.