

# **Philipps-Universität Marburg**

Fachbereich 12 - Mathematik und Informatik

**Philipps**



**Universität  
Marburg**

Bachelorarbeit

## **Pop-Art Effekte mit OpenGL Shadern**

von  
Keita  
27. September 2025

Betreuer:  
Prof. Dr. Thorsten Thormahlen

Arbeitsgruppe Grafik und Multimedia Programmierung

## **Erklärung**

Ich, Keita (Informatikstudent an der Philipps-Universität Marburg, Matrikelnummer 3588329), versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die hier vorliegende Bachelorarbeit wurde weder in ihrer jetzigen noch in einer ähnlichen Form einer Prüfungskommission vorgelegt.

Ich bin damit einverstanden, dass der Titel der Arbeit zusammen mit meinem Namen und dem Abgabedatum auf der Webseite der Arbeitsgruppe veröffentlicht wird.

Marburg, 27. September 2025

Keita

# Zusammenfassung

Diese Bachelorarbeit untersucht die Anwendung von GLSL-Shadern zur Erzeugung künstlerischer und physikalisch inspirierter Bildverfremdungen in Echtzeit. Durch die Kombination eines modularen Shader-Managements mit einer Pipeline-Struktur wird eine flexible und erweiterbare Effektverarbeitung ermöglicht.

Die Implementierung umfasst zwanzig Effekte, aufgeteilt in zwei Kategorien: zehn Pop-Art Effekte (z.B. Posterisierung, Halftone) und zehn physikalische Effekte (z.B. Verzerrung, Bloom). Jeder Effekt besteht aus jeweils eigenen Shader-Dateien und einer zugehörigen Uniform-Klasse zur Parametrierung. Alle Uniform-Setter-Klassen implementieren ein gemeinsames Interface, das für das einheitliche Setzen von Shader-Parametern verwendet wird.

Das System basiert auf einem Java-Framework mit JOGL und folgt dem MVC-Architekturmuster. Die Benutzeroberfläche ermöglicht die interaktive Auswahl, Kombination und Konfiguration der Effekte sowie das Zurücksetzen und Speichern des aktuellen Bildzustands. Die technische Umsetzung umfasst Shader-Komposition, Texturladung, Framebuffer-Verwaltung und Integrationstests.

Der modulare Aufbau erlaubt eine effiziente Verarbeitung und hohe Gestaltungsfreiheit. Die aktuelle Implementierung eignet sich besonders für visuelle Experimente und kreative Bildmanipulation. Zukünftige Erweiterungen könnten GPU-Optimierungen, animierte Effekte oder eine adaptive Steuerung der Effektpараметer in Echtzeit ermöglichen.

# Abstract

This bachelor thesis explores the use of GLSL shaders to create artistic and physically inspired image transformations in real time. By combining a modular shader management system with a pipeline structure, the implementation enables flexible and extensible effect processing.

The system comprises twenty visual effects, divided into two categories: ten pop-art effects (e.g., posterization, halftone) and ten physical effects (e.g., distortion, bloom). Each effect is implemented using dedicated shader files and an associated uniform class for parameter control. All uniform setter classes implement a shared interface that standardizes the configuration of shader parameters.

The software is built on a Java framework using JOGL and follows the Model-View-Controller (MVC) architectural pattern. The graphical user interface allows interactive selection, combination, and configuration of effects, as well as resetting and saving the current image state. The technical implementation includes shader composition, texture loading, framebuffer management, and integration testing.

The modular design enables efficient processing and offers a high degree of creative flexibility. The current implementation is particularly suitable for visual experimentation and artistic image manipulation. Future extensions may include GPU optimizations, animated effects, or adaptive real-time parameter control.

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Ziel der Arbeit . . . . .	1
1.3. Aufbau der Arbeit . . . . .	2
<b>2. Grundlagen</b>	<b>3</b>
2.1. Einführung in GLSL und Shader-Programmierung . . . . .	3
2.1.1. Die OpenGL-Pipeline . . . . .	3
2.1.2. Vertex- und Fragment-Shader . . . . .	4
2.1.3. Shader-Komposition und Modularität . . . . .	4
2.2. Bildverarbeitung mit Shadern . . . . .	5
2.2.1. Texturierung und Shader Manipulation . . . . .	5
2.2.2. Framebuffer und Render-to-Texture . . . . .	6
2.2.3. Uniforms und Parametrisierung . . . . .	7
2.3. Künstlerische Effekte und Pop-Art in der Computergrafik . . . . .	7
2.3.1. Pop-Art als Stilrichtung . . . . .	7
2.3.2. Digitale Umsetzung grafischer Pop-Art-Elemente . . . . .	8
2.3.3. Beispiele aus Film, Werbung und Games . . . . .	9
2.4. Physikalische Effekte in der Computergrafik . . . . .	9
2.4.1. Lichtbasierte Effekte . . . . .	10
2.4.2. Verzerrungsbasierte Effekte . . . . .	10
2.4.3. Bildflächenbasierte Effekte . . . . .	11
<b>3. Implementierung der Architektur</b>	<b>13</b>
3.1. MVC-Architektur . . . . .	13
3.2. Projektstruktur & Technologien . . . . .	14
3.3. Textur- und Shaderverwaltung . . . . .	16
3.3.1. Texturladen und OpenGL-Bindung . . . . .	16
3.3.2. Shader-Laden und Kompilieren . . . . .	16
3.4. Rendering-Pipeline: Framebuffer, Effektketten . . . . .	17
3.5. Uniform-Verwaltung über Interface . . . . .	17
3.6. Benutzeroberfläche und Interaktion (GUI) . . . . .	18
3.6.1. Effekt-Dialoge zur Parametersteuerung . . . . .	19
3.7. Testklassen und Qualitätssicherung . . . . .	20
<b>4. Implementierung der Effekte</b>	<b>21</b>
4.1. Pop-Art-Effekte . . . . .	21
4.1.1. Posterisierung . . . . .	21
4.1.2. Duotone . . . . .	22
4.1.3. Tritone . . . . .	22
4.1.4. Selective Color Boost . . . . .	22
4.1.5. Halftone . . . . .	23

4.1.6. Benday Dots . . . . .	23
4.1.7. Out-of-Register Print . . . . .	23
4.1.8. Comic Look . . . . .	24
4.1.9. Intelligent Bold Outlines . . . . .	24
4.1.10. Seriality . . . . .	24
4.2. Physikalische Effekte . . . . .	25
4.2.1. Bloom . . . . .	25
4.2.2. Chromatic Aberration . . . . .	26
4.2.3. Distortion . . . . .	26
4.2.4. Chromatic Wave Distortion . . . . .	26
4.2.5. Heat Distortion . . . . .	27
4.2.6. Water Ripple . . . . .	27
4.2.7. Refraction . . . . .	27
4.2.8. Vignette . . . . .	28
4.2.9. Noise . . . . .	28
4.2.10. Scanline . . . . .	28
<b>5. Ergebnisse und Evaluation</b>	<b>29</b>
5.1. Auswahl und Beschreibung der Ergebnisbilder . . . . .	29
5.1.1. Kombinationsbeispiel 1: Analog Glitch-Look . . . . .	29
5.1.2. Kombinationsbeispiel 2: Nostalgischer Zeitungsdruck . . . . .	30
5.1.3. Kombinationsbeispiel 3: Dramatische Verzerrung . . . . .	31
5.1.4. Kombinationsbeispiel 4: Surrealistische Wasserlandschaft . . . . .	31
5.2. Analyse und Vergleich der Ergebnisbilder . . . . .	32
5.3. Performance-Analyse und zeitlicher Vergleich . . . . .	34
<b>6. Zusammenfassung und Ausblick</b>	<b>36</b>
<b>Literaturverzeichnis</b>	<b>37</b>
<b>Abkürzungsverzeichnis</b>	<b>39</b>
<b>Abbildungsverzeichnis</b>	<b>40</b>
<b>Tabellenverzeichnis</b>	<b>42</b>
<b>A. Anhang</b>	<b>43</b>
<b>Danksagung</b>	<b>53</b>

# 1. Einleitung

Diese Bachelorarbeit beschäftigt sich mit der Echtzeitanwendung kombinierbarer visueller Effekte durch OpenGL-Shader in GLSL. Im Mittelpunkt steht die Entwicklung eines modularen Systems, das sowohl künstlerisch inspirierte Pop-Art-Effekte als auch physikalisch inspirierte Effekte umfasst. Die Effekte können in frei definierbaren Pipelines kombiniert und parametergesteuert angewendet werden. Die technische Umsetzung erfolgt in Java unter Verwendung der JOGL-Bibliothek, ergänzt durch eine Benutzeroberfläche zur interaktiven Steuerung und Vorschau der Effekte.

## 1.1. Motivation

Die Entwicklung der Computergrafik wird zunehmend durch die fortschreitende Leistungsfähigkeit moderner Grafikhardware geprägt. Besonders in kreativen Anwendungen, etwa in der Bildbearbeitung, steigt der Anspruch an Flexibilität, Echtzeitfähigkeit und visuelle Ausdrucksstärke. Shaderprogramme, die direkt auf der GPU ausgeführt werden, ermöglichen es, Bilder dynamisch zu verändern, wobei nicht nur realistische, sondern auch stilisierte und künstlerisch verfremdete Effekte erzielt werden können. Gerade Pop-Art-Effekte stellen ein spannendes Anwendungsfeld dar, das technische Innovation mit ästhetischer Vielfalt verbindet.

Moderne Rendering-Architekturen bieten die Möglichkeit, mehrere visuelle Transformationen in modular aufgebauten Pipelines zu kombinieren. Dabei ist eine performante Ausführung in Echtzeit unerlässlich, um Interaktivität und flüssige Rückmeldung zu gewährleisten. Die Fähigkeit, mehrere Effekte flexibel miteinander zu verknüpfen und gleichzeitig individuell zu steuern, eröffnet neue kreative und technische Potenziale.

Die Gestaltung eines solchen Systems erfordert sowohl ein fundiertes Verständnis von Echtzeit-Rendering als auch die effiziente Nutzung der verfügbaren Hardware-Ressourcen. Aufbauend auf etablierten Methoden aus der Echtzeitgrafik und unter Verwendung der Programmiersprache Java in Kombination mit der OpenGL-Bindung sowie der Shadersprache GLSL wird in dieser Arbeit ein Framework entwickelt, das sowohl Pop-Art-Transformationen als auch physikalisch motivierte Bildverfremdungen integriert. Die konzeptionelle Grundlage dieser Arbeit orientiert sich an zentralen Prinzipien der modernen Echtzeitdarstellung, wie sie in der Literatur zur interaktiven Computergrafik beschrieben werden. [1]

## 1.2. Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung eines interaktiven Frameworks zur modularen Echtzeit-Bildverarbeitung, das sowohl Pop-Art inspirierte als auch physikalisch inspirierte Shader-Effekte kombiniert. Dabei liegt der Schwerpunkt auf einem künstlerischen Umgang

mit dem Erscheinungsbild digitaler Bilder, wie er in modernen Konzepten des Appearance Design beschrieben wird [2].

### Echtzeit

Die Bildmanipulation soll auf handelsüblicher Hardware in Echtzeit erfolgen. Nutzeraktionen wie das Aktivieren, Deaktivieren oder Kombinieren von Effekten sollen ohne spürbare Verzögerung direkt visuell umgesetzt werden. Auch bei hoher Bildauflösung oder mehreren aktiven Shadern soll die Interaktion flüssig bleiben.

### Kosten

Das Framework soll auf regulären Desktop-Systemen mit OpenGL-fähiger Grafikkarte lauffähig sein. Es wird keine Spezialhardware benötigt, sodass auch Geräte im Consumer-Bereich ausreichen.

### Abgrenzung

Besonders im Vergleich zu statischen Bildfiltern bietet die vorgeschlagene Lösung eine modulare Architektur, die eine dynamische Kombination visueller Effekte ermöglicht und damit den kreativen Spielraum der Nutzenden erweitert.

## 1.3. Aufbau der Arbeit

Kapitel 2 vermittelt die theoretischen Grundlagen dieser Arbeit. Es beginnt mit einer Einführung in die Shader-Programmierung mit GLSL, erläutert die OpenGL-Pipeline und geht auf den Aufbau sowie die Modularität von Shadern ein. Danach werden Konzepte der Bildverarbeitung vorgestellt, etwa Texturierung, Framebuffer-Nutzung und Parametrisierung über Uniforms. Den Abschluss bildet ein Überblick über künstlerische (insbesondere Pop-Art) und physikalische Effekte in der Computergrafik.

Kapitel 3 beschreibt die Architektur des entwickelten Frameworks. Es wird zunächst die gewählte MVC-Struktur erläutert, anschließend folgen Details zur Projektorganisation, Shader- und Texturverwaltung sowie zur Rendering-Pipeline. Danach wird die Uniform-Verwaltung über das Interface behandelt. Auch die grafische Benutzeroberfläche (GUI) und die Interaktionsmöglichkeiten werden in diesem Kapitel beschrieben. Abschließend werden die implementierten Testklassen vorgestellt, die zur Gewährleistung der Qualitätssicherung und Stabilität des Frameworks dienen.

Kapitel 4 behandelt die Implementierung der konkreten Shader-Effekte. Die Pop-Art-inspirierten und physikalischen Effekte werden dabei jeweils mit Parametern und Besonderheiten einzeln vorgestellt.

Kapitel 5 stellt die Ergebnisse dar. Die entwickelten Effekte werden anhand ausgewählter Testbilder bewertet und verglichen.

Kapitel 6 fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen und zukünftige Arbeiten.

## 2. Grundlagen

In diesem Kapitel werden die theoretischen und technischen Grundlagen erläutert, die für das Verständnis der späteren Implementierung notwendig sind. Dazu zählen die Prinzipien der Shader-Programmierung in GLSL, relevante Aspekte der OpenGL-Pipeline sowie Konzepte wie Texturierung, Framebuffer-Nutzung und Parametrisierung über Uniforms. Weiterhin werden künstlerisch inspirierte Bildbearbeitungstechniken sowie physikalische Effekte in der Computergrafik behandelt. Ziel ist es, ein fundiertes Verständnis für die gestalterischen und technischen Möglichkeiten zu schaffen, die dem modularen Aufbau des entwickelten Frameworks zugrunde liegen.

### 2.1. Einführung in GLSL und Shader-Programmierung

Die OpenGL Shading Language (GLSL) ist eine Programmiersprache zur Erstellung von Shadern, welche Grafikeffekte direkt auf der GPU ermöglichen. Shader werden eingesetzt, um Beleuchtung, Effekte und Bildmanipulationen effizient auf der GPU umzusetzen.

Im Gegensatz zum klassischen CPU-basierten Rendering mit starrer Fix-Function-Pipeline bietet GLSL eine programmierbare Pipeline, die mehr Flexibilität und Kontrolle erlaubt. Diese Entwicklung stellte einen Meilenstein in der Computergrafik dar.

Die wichtigsten Shader-Typen sind Vertex- und Fragment-Shader, die in Abschnitt 2.1.2 näher erläutert werden, während weitere Stufen wie Tessellation-, Geometry- und Compute-Shader zusätzliche Funktionen bieten. [3]

#### 2.1.1. Die OpenGL-Pipeline

Um die Rolle der verschiedenen Shader-Typen besser zu verstehen, betrachten wir zunächst die OpenGL-Rendering-Pipeline, die den Ablauf der Grafikverarbeitung beschreibt. Die Pipeline gliedert sich in mehrere Stufen, die nacheinander durchlaufen werden: Zunächst gelangen die Eingabedaten in Form von Vertex-Informationen in die Pipeline. Diese werden vom Vertex-Shader verarbeitet, der unter anderem geometrische Transformationen und Beleuchtungsberechnungen durchführt. Anschließend erfolgt die Rasterisierung, bei der die geometrischen Daten in Fragmente (potenzielle Bildpixel) umgewandelt werden. Im nächsten Schritt übernimmt der Fragment-Shader die Farb- und Texturinformationen und berechnet das endgültige Erscheinungsbild der Fragmente. Abschließend werden die berechneten Fragmente im Framebuffer, einem Zwischenspeicher für Bilddaten, gespeichert und auf dem Bildschirm ausgegeben. . [3]

Zusätzlich zu Vertex- und Fragment-Shadern können weitere Shader-Stufen wie Geometry-, Tessellation- und Compute-Shader in der Pipeline integriert werden, um spezielle Effekte und komplexe Geometrieverarbeitung zu ermöglichen. Die Daten, etwa Positionen, Farben oder Texturkoordinaten, fließen dabei kontinuierlich durch die Pipeline und werden an

den jeweiligen Stufen von den Shadern angepasst. Ein visuelles Diagramm dieser Pipeline erleichtert das Verständnis dieses komplexen Prozesses.

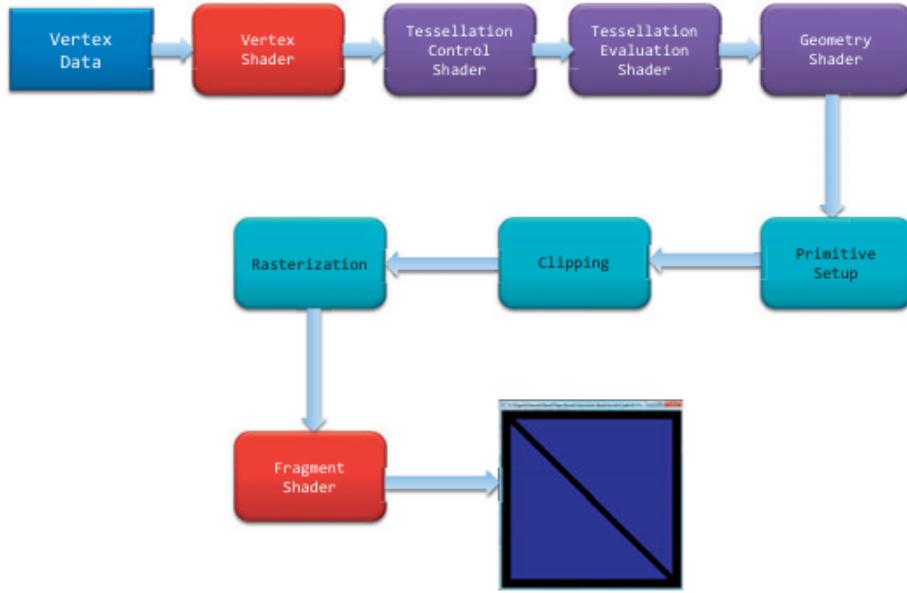


Abbildung 2.1.: Übersicht der OpenGL-Rendering-Pipeline. Quelle: [3]

### 2.1.2. Vertex- und Fragment-Shader

Die beiden wichtigsten Shader-Typen in der OpenGL-Pipeline sind der Vertex-Shader und der Fragment-Shader. Der Vertex-Shader verarbeitet pro Scheitelpunkt eingehende Daten wie Positionen, Normalen oder Texturkoordinaten. Er ist verantwortlich für geometrische Transformationen, etwa Modell-, Sicht- und Projektionsmatrizen, die den 3D-Raum in den 2D-Bildraum abbilden. Zudem kann er zusätzliche Attribute berechnen oder weiterreichen, die für die spätere Verarbeitung benötigt werden.

Der Fragment-Shader hingegen arbeitet auf der Ebene der einzelnen Bildfragmente, also potenziellen Pixeln, und bestimmt deren endgültige Farbe. Hier werden Beleuchtungsmodelle, Texturzugriffe und Farbmischungen berechnet, um das visuelle Erscheinungsbild zu erzeugen.

Wichtig ist dabei der Unterschied zwischen den Eingabe- und Ausgabeveriablen der Shadern: „in“ bezeichnet Daten, die ein Shader empfängt, „out“ Daten, die er an die nächste Pipeline-Stufe weitergibt. Uniform-Variablen sind konstant während eines Render-Durchlaufs und dienen zur Übergabe externer Parameter wie Lichtposition oder Zeit. [4]

### 2.1.3. Shader-Komposition und Modularität

Um Vertex- und Fragment-Shader in einer Anwendung zu nutzen, müssen sie zu einem gemeinsamen Shaderprogramm zusammengeführt werden. Zunächst werden mit `glCreateShader` separate Shader-Objekte für jede Stufe (z.B. Vertex und Fragment) erstellt. Der jeweilige Quellcode wird den Shadern mit `glShaderSource` zugewiesen und anschließend mit `glCompileShader` kompiliert. Danach werden die kompilierten Shader mit `glAttachShader` an ein Shaderprogramm gebunden, das zuvor mit `glCreateProgram` erstellt wurde. Sobald

alle benötigten Shader angehängt sind, erfolgt der finale Linkvorgang mit `glLinkProgram`, wodurch ein ausführbares Shaderprogramm entsteht, das in der Rendering-Pipeline verwendet werden kann.

Während der Ausführung kommunizieren die Shader-Stufen über `out`- und `in`-Variablen, die interpolierte Daten von der Vertex- zur Fragment-Stufe übergeben. Uniform-Variablen dienen dazu, konstante Parameter wie Lichtquellen oder Transformationen von der CPU an alle Shader-Stufen zu übermitteln. [4]

Für komplexere Effekte ist ein modularer Aufbau von Shadern sinnvoll. In GLSL lassen sich eigene Funktionen definieren, um wiederkehrende Berechnungen wie Beleuchtungsmodelle oder Texturoperationen effizient zu strukturieren. Zudem besteht die Möglichkeit, Shader-Code in externe Dateien auszulagern und mittels Präprozessor-Direktiven wie `#include` in den Haupt-Shader zu integrieren. Dies fördert die Wartbarkeit und Wiederverwendbarkeit des Codes. Ein modulares Shadersystem erlaubt es beispielsweise, verschiedene Effekt-Shader wie Spiegelungen oder Schatten gezielt miteinander zu kombinieren und flexibel auszutauschen. [4]

## 2.2. Bildverarbeitung mit Shadern

Nachdem Shader in der Pipeline modular aufgebaut und kombiniert wurden, kommt ihre Stärke insbesondere in der Bildverarbeitung zum Tragen. Image-based Rendering beschreibt Techniken, bei denen bereits gerenderte Szenen oder Texturen als Grundlage für weitere Bildmanipulationen dienen. Shader werden dabei genutzt, um gezielt visuelle Effekte auf diese Bilddaten anzuwenden, ohne die Geometrie erneut zu berechnen. Typische Anwendungen sind Postprocessing-Effekte wie Farbkorrekturen, Kontrastanpassungen oder Unschärfe (Blur). Diese Effekte werden meist im Fragment-Shader realisiert, indem das Bild als Textur in einen Fullscreen-Quad gemappt und anschließend modifiziert wird. Durch solche Nachbearbeitungsschritte lässt sich das visuelle Erscheinungsbild flexibel und effizient optimieren. [5]

### 2.2.1. Texturierung und Shader Manipulation

Ein wichtiger Aspekt der Bildverarbeitung mit Shadern ist die gezielte Manipulation von Texturen, die im Folgenden näher erläutert wird. Ein zentrales Element der Bildverarbeitung mit Shadern ist der Zugriff auf Texturen und deren gezielte Manipulation. Texturen sind 2D-Bilddaten, die mithilfe von UV-Koordinaten auf geometrische Flächen projiziert werden. Diese UV-Koordinaten definieren, wie eine Textur über die Oberfläche eines 3D-Objekts gelegt wird. In GLSL erfolgt der Zugriff auf Texturen über sogenannte `sampler2D`-Uniforms, die im Fragment-Shader mit der `texture()`-Funktion ausgelesen werden. Dabei können pro Fragment Farbwerte aus der Textur abgerufen und weiterverarbeitet werden.

Typische Manipulationen umfassen Farbanpassungen, etwa durch das Modifizieren von Helligkeit, Kontrast oder Farbtönen. Ebenso lassen sich Maskierungen realisieren, bei denen nur bestimmte Bildbereiche beeinflusst werden.

Neben farblichen Anpassungen lassen sich mit Shadern auch geometrische Effekte auf Texturen anwenden. Beispielsweise können durch Verzerrungen (Distortion) UV-Koordinaten

dynamisch verändert werden, um Wellen- oder Spiegelungseffekte zu erzeugen. [5]



Abbildung 2.2.: Darstellung der Shader Manipulation. Quelle: Eigene Darstellung

### 2.2.2. Framebuffer und Render-to-Texture

Damit die in Abschnitt 2.2.1 beschriebenen Manipulationen effizient umgesetzt werden können, müssen Shader-Ausgaben zunächst in Texturen geschrieben werden. Dies geschieht mithilfe von Framebuffer-Objekten (FBOs). Ein FBO erlaubt es, die Ausgabe eines Renderprozesses nicht direkt auf den Bildschirm, sondern in eine Textur zu schreiben. Dieses Verfahren wird als Render-to-Texture bezeichnet und bildet die Grundlage für viele Postprocessing- und Mehrpass-Effekte.

Der typische Ablauf umfasst dabei vier Schritte: Zunächst wird das gewünschte Framebuffer-Objekt gebunden (*Bind FBO*). Anschließend wird die Szene ganz normal gerendert, wobei die Ausgabe nun in die gebundene Textur fließt (*Render Szene in Textur*). Danach wird der Framebuffer wieder entkoppelt und auf den Standard-FBO zurückgeschaltet (*Unbind FBO*). Schließlich kann die erzeugte Textur als Eingabe in einem weiteren Rendering-Schritt verwendet werden, etwa zur Nachbearbeitung (*Textur als Input nutzen*).

Dieses Prinzip ist zentral im sogenannten Multi-pass Rendering, bei dem mehrere Verarbeitungsschritte nacheinander auf das Bild angewendet werden. Dadurch lassen sich komplexe Effekte wie Bloom, Blur oder Heat Distortion effizient umsetzen.

Im praktischen Einsatz bedeutet das, dass die Szene beispielsweise in einem ersten Pass weichgezeichnet wird, bevor im zweiten Pass eine Blend-Operation mit dem Originalbild erfolgt. Die Kombination mehrerer FBOs ermöglicht zudem flexible Effektketten und eine hohe Kontrolle über das visuelle Endergebnis. [5]

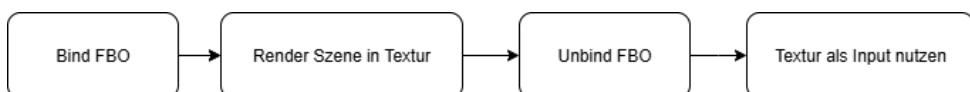


Abbildung 2.3.: Workflow des Render-to-Texture-Verfahrens mit FBO.

Quelle : in Anlehnung an [5]

### 2.2.3. Uniforms und Parametrisierung

Um die in den vorherigen Abschnitten beschriebenen Shader-Manipulationen dynamisch und flexibel steuern zu können, kommen `uniform`-Variablen in GLSL zum Einsatz. Diese globalen Variablen werden vom CPU-Code (OpenGL) gesetzt und bleiben innerhalb eines Shaders konstant, bis sie explizit neu geschrieben werden. Im Gegensatz zu `varying`- oder `attribute`-Variablen werden `uniforms` nicht pro Fragment oder Vertex interpoliert, sondern gelten global für alle Shader-Ausführungen innerhalb eines Draw-Calls.

Über diese `uniform`-Variablen lassen sich externe Parameter wie Zeit, Farbe oder Intensität eines Effekts an den Shader übergeben. Dadurch kann das Verhalten des Shaders flexibel angepasst werden, ohne dass der Shader-Code selbst verändert werden muss. Typische Anwendungen sind etwa die Steuerung der Stärke eines Blur-Effekts, das Einfärben von Objekten oder Animationen auf Basis einer Zeitvariable.

Besonders in interaktiven Anwendungen ist die Parametrisierung über `uniforms` essenziell. So kann beispielsweise die Anzahl der Scanlines in Echtzeit über einen GUI-Slider angepasst werden. Diese Trennung zwischen Logik (CPU) und Darstellung (GPU) ermöglicht eine hohe Flexibilität und Benutzerinteraktion im Rendering-Prozess. [5]

## 2.3. Künstlerische Effekte und Pop-Art in der Computergrafik

Die zuvor beschriebenen Möglichkeiten zur Parametrisierung von Shadern bilden die Grundlage, um komplexe und interaktive künstlerische Effekte zu realisieren. Der Einsatz künstlerischer Effekte in der Computergrafik ermöglicht eine gezielte ästhetische Gestaltung, die über rein technische Bildverarbeitung hinausgeht. Ein besonders markanter Stil ist die Pop-Art, die durch kräftige, kontrastreiche Farben, Rasterpunkte, klare Konturen und die Betonung alltäglicher Objekte geprägt ist. Diese visuellen Merkmale lassen sich mit modernen Shader-Techniken digital umsetzen, etwa durch Farbposterisierung, simulierte Benday-Dots oder stilisierte Kantenfilter. Die Verbindung traditioneller Pop-Art-Elemente mit Echtzeitgrafik eröffnet vielfältige Möglichkeiten für interaktive Medien, Film, Werbung und Spiele, wobei der Wiedererkennungswert und die starke visuelle Wirkung den künstlerischen Stil für digitale Anwendungen besonders attraktiv machen.

### 2.3.1. Pop-Art als Stilrichtung

Um die technische Umsetzung der Pop-Art in der Computergrafik zu verstehen, ist zunächst ein Blick auf ihre historische Entwicklung und stilistischen Merkmale notwendig. Die Pop-Art entstand in den 1950er- und 1960er-Jahren als Gegenbewegung zu den damals dominierenden abstrakten Kunstströmungen und orientierte sich stark an der visuellen Sprache der Massenmedien und Konsumkultur. Sie griff Motive aus Werbung, Comics und Alltagsgegenständen auf und überführte sie in einen künstlerischen Kontext. Dabei wurden kräftige Farben, klare Konturen und serielle Muster zu zentralen Gestaltungsmitteln, die den unmittelbaren Wiedererkennungswert der Werke steigerten.

Die Bewegung wurde maßgeblich von Künstlern wie Andy Warhol und Roy Lichtenstein geprägt, die alltägliche Objekte und Medienbilder in ihren Arbeiten neu inszenierten.

Andy Warhol nutzte serielle Wiederholungen von Konsumprodukten und Prominentenporträts, um die Mechanismen der Massenproduktion und des Starkults zu hinterfragen. Roy Lichtenstein hingegen übertrug den Stil von Comic-Zeichnungen, inklusive großformatiger Rasterpunkte und klarer Konturen, in die Kunstwelt. Beide Künstler setzten auf Wiederholung und Variation als zentrale gestalterische Mittel, um Konsum- und Medienphänomene kritisch zu reflektieren. Diese Ansätze führten zu einer neuen visuellen Sprache, die sowohl zugänglich als auch provokativ wirkte.

Die Pop-Art beeinflusste nachhaltig die zeitgenössische Mediengestaltung und findet bis heute Anwendung in Grafikdesign, Werbung und digitaler Kunst. Ihre klaren Formen und kontrastreichen Farben eignen sich besonders für die Umsetzung in digitalen Medien, da sie eine hohe visuelle Wirkung entfalten und zugleich technische Reproduzierbarkeit begünstigen. In der Computergrafik lassen sich diese Merkmale gezielt in Shader-Programmen nachbilden, um den charakteristischen Stil für interaktive und visuelle Medienprojekte nutzbar zu machen. [6]



Andy Warhol



Roy Lichtenstein

Abbildung 2.4.: Typische Stilmerkmale in der Pop-Art.

Quelle : [6]

### 2.3.2. Digitale Umsetzung grafischer Pop-Art-Elemente

Um den einzigartigen visuellen Stil der Pop-Art auch in digitalen Medien zu realisieren, werden ihre typischen grafischen Elemente mithilfe moderner Computergrafiktechniken umgesetzt. Die charakteristischen Merkmale der Pop-Art, wie Rasterpunkte (Ben Day Dots), kräftige Farben und klare Konturen, lassen sich in der Computergrafik mithilfe moderner Shader-Technologien wirkungsvoll digital umsetzen. Shader bieten dabei die Möglichkeit, einzelne Pixel und Bildbereiche gezielt zu manipulieren und so typische Pop-Art-Effekte in Echtzeit zu erzeugen.

Ein zentrales Element ist die Simulation von Rasterpunkten, die durch spezielle Muster im Shader nachgebildet werden. Diese sogenannten Ben Day Dots ahmen die Drucktechnik der klassischen Pop-Art nach und erzeugen den charakteristischen gepunkteten Look. Farbposterisierung und Duotone-Techniken reduzieren die Farbpalette auf wenige, kräftige Farbtöne, was den kontrastreichen Stil der Pop-Art unterstützt und gleichzeitig die Bildwirkung intensiviert.

Zusätzlich lassen sich Überdruck-Fehler, sogenannte Out-of-Register-Effekte, simulieren, um den typischen Druckfehler-Look der Pop-Art zu imitieren. Eine weitere wichtige Technik ist die Kantenerkennung, die verwendet wird, um Comic-ähnliche Umrisse zu erzeugen. Diese Konturen heben Objekte hervor und verstärken den grafischen Stil.

Durch die Kombination dieser Shader-Effekte können komplexe und dynamische Pop-Art-Darstellungen erzeugt werden, die sowohl in interaktiven Medien als auch in filmischen und werblichen Anwendungen Einsatz finden. Moderne Grafikpipelines ermöglichen es, diese Techniken effizient zu implementieren und in Echtzeit anzuwenden. [7].

### 2.3.3. Beispiele aus Film, Werbung und Games

Die zuvor beschriebenen digitalen Techniken zur Umsetzung von Pop-Art-Elementen finden in der Praxis vielfältige Anwendung und prägen die visuelle Gestaltung zahlreicher Medienproduktionen. Pop-Art-Elemente finden in zahlreichen zeitgenössischen Medienformen Anwendung und prägen die visuelle Sprache in Film, Werbung und Games. Im Film werden Pop-Art-inspirierte Gestaltungsmittel häufig eingesetzt, um eine stilisierte, grafisch reduzierte Ästhetik zu erzeugen. Der Einsatz von kräftigen Farben, klaren Konturen und Rastermustern in Animationsfilmen schafft nicht nur einen hohen Wiedererkennungswert, sondern verstärkt auch die emotionale Wirkung von Szenen [8].

In der Werbebranche wird der Pop-Art-Stil gezielt genutzt, um Aufmerksamkeit zu generieren und Markenbotschaften klar zu kommunizieren. Die visuelle Direktheit und der kulturelle Bezug der Pop-Art ermöglichen es, Inhalte schnell und einprägsam zu transportieren, was insbesondere in Kampagnen mit hohem Wiedererkennungsdruck wirksam ist [9].

Auch in der Spieleindustrie hat Pop-Art eine besondere Rolle, vor allem bei der Entwicklung einzigartiger visueller Stile. Pop-Art-inspirierte Grafik in Games bietet nicht nur ästhetischen Mehrwert, sondern ist auch generationenübergreifend zugänglich, da sie kulturelle Referenzen mit moderner Interaktivität verbindet [10].

Diese Beispiele verdeutlichen, dass Pop-Art-Ästhetik in digitalen Medien nicht nur als Stilmittel fungiert, sondern auch strategische Funktionen in der Gestaltung von Erzählung, Markenidentität und Nutzererlebnis übernimmt.

## 2.4. Physikalische Effekte in der Computergrafik

Nachdem zuvor Beispiele für Pop-Art-Elemente in verschiedenen Medien vorgestellt wurden, richtet sich der Fokus nun auf die technischen Grundlagen und Umsetzungsmöglichkeiten physikalischer Effekte in der Computergrafik. Physikalische Effekte in der Computergrafik basieren auf der Simulation realer optischer Phänomene, um Szenen glaubwürdiger und immersiver zu gestalten. Sie werden häufig mithilfe von GLSL-Shadern umgesetzt, die Bildinhalte in Echtzeit manipulieren. Solche Effekte können Lichtverhalten, Bildverzerrungen oder bildflächenweite Veränderungen nachbilden und so den Realismus einer Szene deutlich erhöhen. Eine Übersicht typischer Kategorien und Beispiele physikalischer Effekte zeigt Tabelle 2.1.

Kategorie	Beispiele
Lichtbasiert	Bloom, Chromatic Aberration
Verzerrungsbasiert	Distortion, Chromatic Wave Distortion, Heat Distortion, Water Ripple, Refraction
Bildflächenbasiert	Vignette, Noise, Scanline

Tabelle 2.1.: Kategorien physikalischer Effekte in der Computergrafik

Neben ihrer technischen Funktion haben sie auch eine gestalterische Rolle, da sie gezielt eingesetzt werden, um Atmosphäre, Tiefe und visuelle Betonungen zu schaffen.

### 2.4.1. Lichtbasierte Effekte

Lichtbasierte Effekte in der Echtzeitgrafik dienen der realistischen oder stilisierten Darstellung lichtbedingter Phänomene und tragen entscheidend zur visuellen Qualität einer Szene bei. Ein häufig genutztes Verfahren ist der *Bloom*-Effekt, der helle Bildbereiche leicht überstrahlen lässt. Dadurch entsteht der Eindruck einer intensiven Lichtquelle, wie sie auch in realen optischen Systemen durch Streuung im Linsenmaterial oder in der Atmosphäre auftritt. Dieser Effekt wird in der Post-Processing-Pipeline umgesetzt, indem besonders helle Pixel extrahiert, weichgezeichnet und anschließend wieder in das Bild integriert werden.

Ein weiteres Beispiel ist die Simulation der *Chromatic Aberration*. Hierbei handelt es sich um eine farbliche Verschiebung an Kontrastkanten, verursacht durch die wellenlängenabhängige Brechung des Lichts in Linsen. In der Computergrafik wird dieses Phänomen oft leicht übertrieben eingesetzt, um einen filmischen Look oder den Eindruck einer Kameralinse zu erzeugen.

Die physikalischen Grundlagen dieser Effekte liegen in der Streuung und Brechung von Licht, die sich mit vereinfachten Modellen in Echtzeit simulieren lassen. Durch den gezielten Einsatz dieser Effekte können Szenen nicht nur realistischer wirken, sondern auch bestimmte Stimmungen oder stilistische Akzente erhalten [1].

### 2.4.2. Verzerrungsbasierte Effekte

Neben lichtbasierten Phänomenen spielen auch Verzerrungseffekte eine zentrale Rolle in der Echtzeitgrafik, da sie gezielt Abweichungen von einer idealen Abbildung erzeugen können. Verzerrungseffekte in der Echtzeitgrafik dienen dazu, optische Abweichungen von einer idealen Abbildung zu erzeugen. Sie werden sowohl in realistischen Szenen als auch in stilisierten Darstellungen eingesetzt, um besondere Atmosphären, Hitze- oder Flüssigkeitseffekte oder surreale Bildeindrücke zu schaffen. Grundlegend basieren diese Effekte auf der Manipulation von Texturkoordinaten in Fragment-Shadern, wodurch einzelne Bildbereiche gezielt verschoben oder verformt werden. [1]

Ein einfaches Beispiel ist der allgemeine *Distortion*-Effekt, bei dem ein zusätzliches Stör-muster oder eine Vektormap genutzt wird, um Bildbereiche unregelmäßig zu verschieben. Dadurch lassen sich z.B. Glaseffekte, Energieschilder oder abstrakte visuelle Störungen erzeugen.

Eine spezielle Form stellt die *Heat Distortion* dar, die das Flimmern heißer Luft simuliert. Hierbei wird oft ein perlinrauschbasiertes Störmuster verwendet, das kontinuierlich animiert und über den Hintergrund gelegt wird, um die für Hitze typischen, zufällig oszillierenden Brechungen nachzubilden. [11]

Ebenfalls weit verbreitet ist der *Water Ripple*-Effekt, der sich durch konzentrische Wellenstrukturen auszeichnet. Er wird häufig durch mathematische Funktionen wie Sinus- und Kosinustransformationen in den Texturkoordinaten realisiert, um die Oberflächenverformung von Wasser nachzuahmen. [12]

Bei der *Refraction* hingegen wird gezielt die Brechung von Licht simuliert, wie sie beim Durchgang durch transparente Materialien auftritt. Hierbei kommen Normalmaps oder Displacement-Maps zum Einsatz, um die Lichtstrahlen scheinbar abzulenken und ein physikalisch plausibles Bild zu erzeugen. [1]

Eine kreative Variante ist die *Chromatic Wave Distortion*, bei der neben der geometrischen Verzerrung auch eine Farbtrennung für die einzelnen RGB-Kanäle erfolgt. Dies kann den Eindruck von Energiestrahlen, digitalen Störungen oder surrealen Übergängen verstärken. [13]

Technisch werden Verzerrungseffekte in Fragment-Shadern durch die Berechnung verschiedener Texturkoordinaten umgesetzt. Ausgangspunkt ist in der Regel eine Distortion-Map, deren Werte zur Modifikation der Sampling-Positionen verwendet werden. Bei statischen Effekten bleibt diese Map unverändert, sodass die Verzerrung konstant im Bild vorhanden ist. Dadurch lassen sich stabile, klar definierte optische Abweichungen erzeugen, die unabhängig von zeitbasierten Animationen wirken. Da Verzerrungen die Wahrnehmung einer Szene stark beeinflussen, ist ihr gezielter Einsatz entscheidend, um die gewünschte Atmosphäre zu unterstützen, ohne die Lesbarkeit oder Orientierung im Bild zu beeinträchtigen. [1]

### 2.4.3. Bildflächenbasierte Effekte

Neben verzerrungsbasierten Effekten gibt es auch Post-Processing-Techniken, die direkt auf dem fertigen Bildframe arbeiten und die gesamte Bildfläche beeinflussen. Bildflächenbasierte Effekte in der Echtzeitgrafik wirken direkt auf der zweidimensionalen Projektion einer Szene, also dem fertig gerenderten Frame, und beeinflussen dadurch das gesamte Bild gleichzeitig. Diese Post-Processing-Techniken sind oft besonders effizient, da sie unabhängig von der zugrunde liegenden 3D-Geometrie arbeiten und nur pro Pixel der Bildfläche berechnet werden. [1]

Ein typisches Beispiel ist die *Vignette*, bei der die Bildränder abgedunkelt werden, um den Blick des Betrachters auf die Bildmitte zu lenken. Dieser Effekt hat seinen Ursprung in optischen Eigenschaften realer Linsen und wird häufig zur Bildfokussierung oder zur Erzeugung einer bestimmten Stimmung eingesetzt. [14]

Der *Scanline* simuliert die horizontale Zeilenstruktur alter Röhrenbildschirme (CRT). Hierbei wird ein Muster aus dunklen Linien oder Helligkeitsschwankungen über das Bild gelegt, um einen retroähnlichen Darstellungsstil zu erzeugen. [15]

*Noise*-Effekte fügen dem Bild zufällige Helligkeits- oder Farbabweichungen hinzu, um das Bild körniger und weniger klinisch glatt erscheinen zu lassen. In moderaten Mengen

steigert dies die wahrgenommene Natürlichkeit, da viele reale Aufnahmegeräte ähnliche Artefakte aufweisen. [16]

Für die Umsetzung solcher Effekte werden häufig einfache Shader-Operationen eingesetzt, die Textur-Sampling und Farbmanipulation kombinieren. Dadurch lassen sich auch mehrere Effekte miteinander verknüpfen, ohne die Bildrate stark zu beeinträchtigen. Insgesamt ermöglichen bildflächenbasierte Post-Processing-Techniken sowohl subtile Verbesserungen der Bildqualität als auch markante stilistische Veränderungen und stellen damit ein effizientes Werkzeug für die visuelle Gestaltung in Echtzeitanwendungen dar. [1]

# 3. Implementierung der Architektur

Nach der theoretischen Betrachtung der Grundlagen folgt nun die praktische Umsetzung in Form einer modularen Softwarearchitektur. Die Implementierung der Architektur folgt einem modularen Aufbau, um Erweiterbarkeit und Wartbarkeit sicherzustellen. Grundlage bildet das *MVC-Architekturmödell*, das eine klare Trennung zwischen Datenhaltung (Model), Darstellung (View) und Steuerung (Controller) gewährleistet. Die *Projektstruktur* ist so organisiert, dass Shader, Texturen und GUI-Elemente effizient verwaltet werden können. Zentrale Komponenten sind die *Textur- und Shaderverwaltung*, welche das Laden, Binden und Kompilieren von Ressourcen übernimmt. Über die *Rendering-Pipeline* werden Effekte in Ketten auf Framebuffer angewendet. Zusätzlich ermöglicht eine *Uniform-Verwaltung über ein Interface* die dynamische Parameteranpassung. Die *GUI* bietet schließlich eine intuitive Interaktion, inklusive spezieller Effekt-Dialoge zur Steuerung einzelner Parameter.

## 3.1. MVC-Architektur

Ein zentraler Bestandteil der gewählten Softwarearchitektur ist das *Model–View–Controller* (MVC) Muster, das als Grundlage für die klare Trennung von Datenhaltung, Darstellung und Steuerung dient. Dieses Architekturprinzip ist seit langem etabliert und wird insbesondere für modulare und erweiterbare Anwendungen empfohlen. [17]

Im vorliegenden Projekt übernimmt das **Model** die Verwaltung der zentralen Daten und Berechnungen. Dazu gehören Shader, Texturen und die Logik zur Realisierung der verschiedenen Effekte. Das **View** ist für die Ausgabe zuständig, also für die Darstellung des gerenderten Frames sowie die grafische Benutzeroberfläche, die eine Interaktion ermöglicht. Der **Controller** bildet die Verbindung zwischen beiden Schichten: Er verarbeitet Benutzereingaben, steuert die Parameter und leitet diese an die Shader und die Rendering-Pipeline weiter.

Die Wahl der MVC-Architektur bringt mehrere Vorteile mit sich. Durch die strikte Trennung der Verantwortlichkeiten wird die Modularität der Anwendung erhöht, was spätere Anpassungen oder Erweiterungen erleichtert. Neue Effekte können in das Model integriert werden, ohne Änderungen am View oder Controller vorzunehmen. Zudem fördert MVC die Wiederverwendbarkeit von Komponenten und sorgt für eine verbesserte Wartbarkeit. Für die Benutzer entsteht durch die klare Struktur eine konsistente und nachvollziehbare Interaktion mit der Software. Insgesamt bietet MVC somit eine robuste Grundlage für die Realisierung einer flexiblen und erweiterbaren Rendering-Architektur.

Die grundlegende Struktur des eingesetzten MVC-Modells ist in Abbildung 3.1 dargestellt.

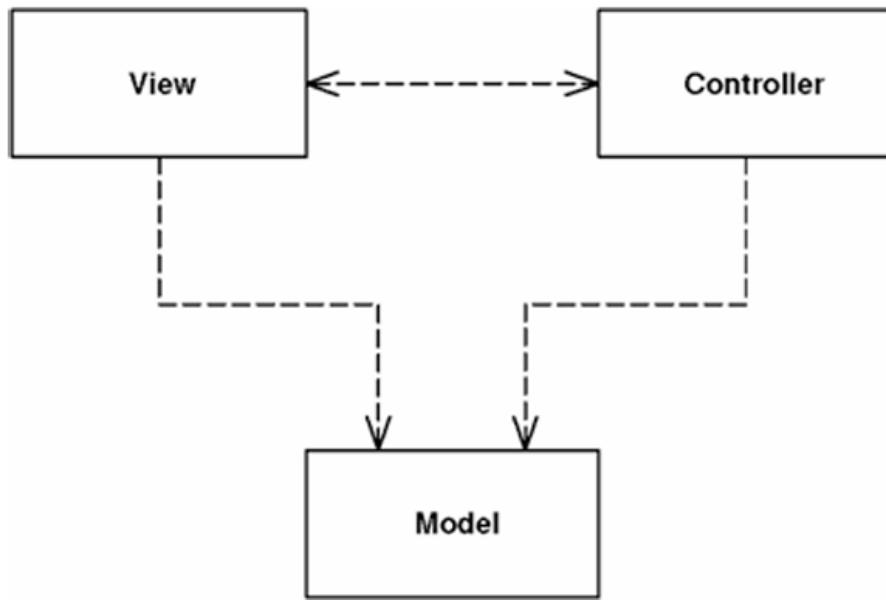


Abbildung 3.1.: Übersicht der MVC-Architektur. Quelle: [17]

## 3.2. Projektstruktur & Technologien

Aufbauend auf der in Abschnitt 3.1 beschriebenen MVC-Architektur wird die konkrete Projektstruktur sowie die eingesetzten Technologien vorgestellt. Die Implementierung des Projekts erfolgt in **Java**, das durch Plattformunabhängigkeit, Stabilität und eine klare Modularisierung überzeugt [18]. Für die Grafikprogrammierung wird **OpenGL** als Application Programming Interface (API) verwendet, da es plattformübergreifend Hardwarebeschleunigtes Rendering und effiziente Shader- sowie Texturverwaltung ermöglicht.

Die **Projektstruktur** ist hierarchisch aufgebaut und folgt dem MVC-Prinzip. Im `src/main/java` Verzeichnis liegen die Kernklassen, die in `model`, `view` und `controller` gegliedert sind. Ein zusätzliches `utils`-Paket stellt Hilfsklassen bereit, während das Model insbesondere Shader, Pipelines und Uniform-Setter verwaltet, der Controller die Ablaufsteuerung übernimmt und die View für GUI-Elemente zuständig ist.

Im `resources`-Ordner befinden sich die **Shader** und **Texturen**, wodurch diese unabhängig vom Quellcode gepflegt werden können. Ergänzend enthält `src/test/java` eine Sammlung von Integrations-Tests, die das Zusammenspiel zentraler Komponenten validieren. Zur Unterstützung des Entwicklungsprozesses wird **Maven** als Build-Automatisierungstool eingesetzt, womit Abhängigkeiten konsistent verwaltet und wiederholbare Builds gewährleistet werden.

Die in Abbildung 3.2 dargestellte Organisation der Ordner erleichtert die Orientierung im Code und ermöglicht eine effiziente Erweiterung um neue Funktionen.

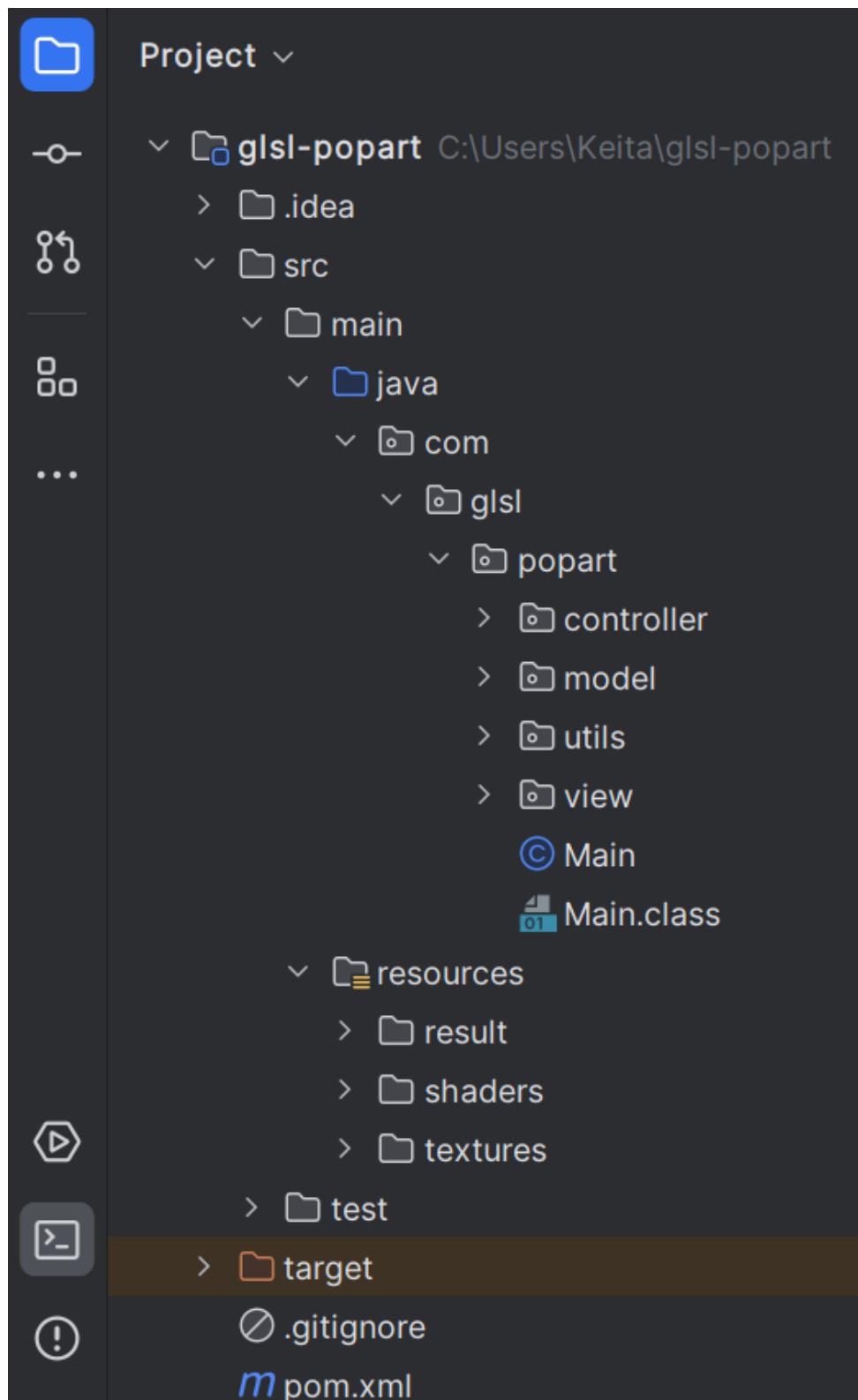


Abbildung 3.2.: Projektstruktur. Quelle: Eigene Darstellung

### 3.3. Textur- und Shaderverwaltung

Innerhalb dieser modularen Architektur wird nun die zentrale Komponente für die Verwaltung von Texturen und Shadern im Detail erläutert. Ein zentrales Element der Rendering-Architektur ist die effiziente Verwaltung von Texturen und Shadern. Sie ermöglicht, Effekte flexibel zu kombinieren und erweitert die Modularität der Anwendung. Während Texturen die visuellen Grundlagen liefern, steuern Shader die Berechnung und Darstellung, sodass beide Komponenten eng verzahnt zusammenarbeiten.

#### 3.3.1. Texturladen und OpenGL-Bindung

Aufbauend auf der zentralen Textur- und Shaderverwaltung ist das effiziente Laden und Binden von Texturen in den GPU-Speicher ein entscheidender Schritt für die Rendering-Pipeline. Zunächst werden Bilddateien, typischerweise im JPG-Format, geladen und auf ihre Kompatibilität mit OpenGL geprüft. Anschließend werden die Texturdaten in Texturobjekte übertragen, die im GPU-Speicher abgelegt werden, um eine schnelle Verarbeitung zu gewährleisten. Dabei werden Speicherformate wie RGB und Filtermethoden wie Linear Filtering verwendet, um eine korrekte Darstellung und Glättung der Texturen sicherzustellen. Schließlich erfolgt die Zuordnung der Texturkoordinaten, sodass die Shader die Texturen korrekt ansprechen und in der Szene darstellen können. [19]

Abbildung 3.3 veranschaulicht den gesamten Workflow.

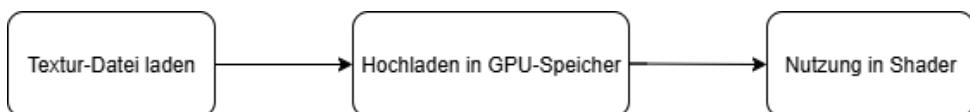


Abbildung 3.3.: Workflow zum Laden von Texturen. Quelle : Eigene Darstellung

#### 3.3.2. Shader-Laden und Kompilieren

Aufbauend auf der Texturverwaltung bildet das Shader-Management einen zentralen Bestandteil der Rendering-Architektur. Als kleine Programme, die auf der GPU laufen, steuern Shader die Berechnung von visuellen Elementen wie Pixeln und Vertices. Die Implementierung setzt auf die Nutzung externer GLSL-Dateien, die im `resources`-Ordner abgelegt werden. Durch die Auslagerung der Shader in separate Dateien wird die Modularität der Anwendung deutlich erhöht, da Shader unabhängig vom Hauptcode entwickelt, getestet und ausgetauscht werden können.

Beim Laden der Shader werden zunächst die entsprechenden Dateien aus dem Ressourcenordner gelesen und deren Inhalt als Strings in den Speicher geladen. Danach erfolgt die Kompilierung der Shader auf der GPU. Während dieses Schritts prüft OpenGL die Syntax und Semantik des Codes, wobei im Falle von Fehlern detaillierte Shader-Logs erzeugt werden. Diese Logs liefern Entwicklern wichtige Informationen zur Fehlerbehebung und erlauben eine schnelle iterative Anpassung. Nach erfolgreicher Kompilierung werden Vertex- und Fragment-Shader zu einem Shader-Programm verknüpft, das anschließend von der Rendering-Pipeline verwendet werden kann. [19]

Abbildung 3.4 veranschaulicht den gesamten Workflow.



Abbildung 3.4.: Workflow zum Laden von Shadern. Quelle : Eigene Darstellung

### 3.4. Rendering-Pipeline: Framebuffer, Effektketten

Die Verwaltung von Shadern und Texturen bildet die Grundlage für die effektive Nutzung der Rendering-Pipeline, in der Framebuffer und Effektketten zentral sind. In dieser Architektur dienen **Framebuffer Objects** (FBOs) dazu, Zwischenbilder während des Renderings zu speichern. Dadurch können die Ergebnisse einzelner Shader-Effekte nicht nur angezeigt, sondern auch als Eingabe für nachfolgende Effekte genutzt werden, ein Prinzip, das als *Render-to-Texture* bekannt ist [5].

Die Pipeline ermöglicht die Erstellung komplexer Effektketten, bei denen mehrere Post-Processing-Schritte sequenziell angewendet werden. Ein Beispiel wäre die Anwendung eines Duotone-Filters, gefolgt von einem Noise-Effekt, wobei das Ergebnis des ersten Effekts als Eingabe für den zweiten dient. Diese Herangehensweise erlaubt eine flexible Kombination und Anpassung der Reihenfolge von Effekten, ohne dass grundlegende Änderungen an der Rendering-Logik notwendig sind. Die modulare Struktur der Pipeline trägt dazu bei, dass neue Effekte unkompliziert integriert werden können, was die Erweiterbarkeit der Anwendung unterstützt.

Darüber hinaus ermöglicht die Nutzung von FBOs die Trennung der Renderausgabe von der eigentlichen Bildanzeige. So kann die GPU mehrere Renderziele gleichzeitig verarbeiten, was die Performance in Echtzeitanwendungen deutlich verbessert. Die visuelle Konsistenz wird durch die zentrale Steuerung der Effektketten gewährleistet, und Entwickler können effizient experimentieren, indem sie einzelne Stufen der Pipeline austauschen oder umordnen. Insgesamt stellt die Rendering-Pipeline somit einen zentralen Baustein für die flexible und leistungsfähige Umsetzung von Echtzeit-Effekten dar.

Abbildung 3.5 veranschaulicht den gesamten Workflow.

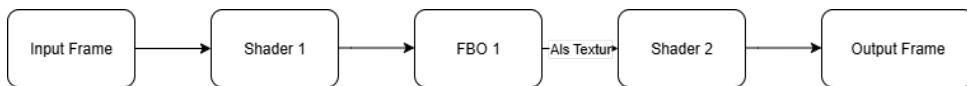


Abbildung 3.5.: Rendering-Pipeline: Aufbau und Datenfluss. Quelle : Eigene Darstellung

### 3.5. Uniform-Verwaltung über Interface

Ein zentrales Element bei der Steuerung der Rendering-Pipeline ist die Verwaltung von **Uniforms**. Dabei handelt es sich um Parameter, die von der CPU an die Shader übergeben werden und deren Verhalten während der Laufzeit beeinflussen. [5] Typische Beispiele sind Farbwerte für das Duotone oder Stärke des Rauschens, die es ermöglichen, das visuelle Ergebnis flexibel anzupassen.

Um diese Parameter effizient zu organisieren, wird eine **zentrale Schnittstelle** eingeführt, die den Zugriff auf Uniforms vereinheitlicht. Anstatt dass jeder Shader individuell

verwaltet werden muss, erfolgt die Übergabe über ein Interface, das die relevanten Parameter kapselt und abstrahiert. Dies reduziert Code-Duplikation erheblich, da gemeinsame Aufgaben wie das Setzen von Farbwerten oder die Bindung von Texturen nicht mehrfach implementiert werden müssen.

Ein weiterer Vorteil dieser Struktur liegt in der dynamischen Steuerung während der Laufzeit. Entwickler oder Benutzer können Parameter in Echtzeit verändern, wodurch Effekte unmittelbar sichtbar werden. Zudem fördert die einheitliche Verwaltung die Erweiterbarkeit: Neue Shader können leicht integriert werden, da sie lediglich das definierte Interface implementieren müssen. Damit stellt die Uniform-Verwaltung über eine zentrale Schnittstelle eine robuste Grundlage dar, um sowohl die Konsistenz als auch die Flexibilität der Rendering-Architektur sicherzustellen.

### 3.6. Benutzeroberfläche und Interaktion (GUI)

Nachdem die interne Verwaltung von Uniforms und Parametern beschrieben wurde, wird nun die Benutzeroberfläche vorgestellt, über die diese Parameter und Effekte interaktiv gesteuert werden können. Die Interaktion des Nutzers mit der Rendering-Architektur erfolgt über eine grafische Benutzeroberfläche, die mit dem Framework **Java Swing** realisiert wurde. Swing bietet eine Vielzahl flexibler Komponenten und eignet sich besonders für modulare Desktop-Anwendungen, da es eine klare Trennung zwischen Darstellung und Logik unterstützt.

Das **Komponenten-Design** umfasst mehrere zentrale Elemente. Zwei **JList**-Komponenten ermöglichen die multiselektive Auswahl von Effekten, die in Pop Art- und physikalische Kategorien gegliedert sind. Damit lassen sich unterschiedliche Effektkombinationen intuitiv zusammenstellen. Ergänzend stehen Kontrollschaftflächen bereit, mit denen Nutzer eine Textur laden, das gerenderte Bild speichern oder alle angewendeten Effekte zurücksetzen können. [20] Für die visuelle Rückmeldung integriert die Oberfläche eine **GLCanvas**-Komponente, die das gerenderte Ergebnis in Echtzeit darstellt und so ein klares Feedback nach Bestätigung der gewählten Einstellungen bietet.

Der **Interaktionsmechanismus** basiert auf der dynamischen Aktualisierung der Rendering-Pipeline. Sobald der Benutzer in den Effektlisten eine Auswahl trifft oder Parameter über Parameter-Kontrollschaftflächen verändert, wird die zugrunde liegende Effektkette automatisch neu konfiguriert. Dadurch entsteht ein interaktiver Workflow: Änderungen werden sofort im Vorschaufenster sichtbar, ohne dass ein Neustart oder manuelles Neuladen erforderlich ist.

Diese Architektur ermöglicht eine effiziente und benutzerfreundliche Steuerung der Effekte. Gleichzeitig unterstützt sie die Erweiterbarkeit, da neue Komponenten oder Effekte problemlos in die bestehende GUI-Struktur integriert werden können.

Abbildung 3.6 veranschaulicht den Aufbau der Benutzeroberfläche sowie die Interaktionsmöglichkeiten.

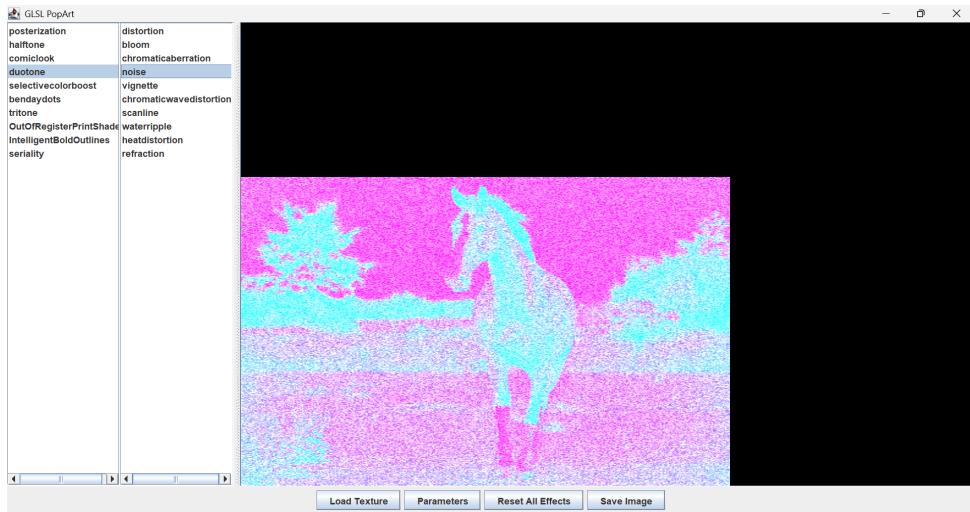


Abbildung 3.6.: GUI. Quelle: Eigene Darstellung

### 3.6.1. Effekt-Dialoge zur Parametersteuerung

Neben der allgemeinen Steuerung über die GUI bieten spezielle **Effekt-Dialoge** eine feinere Kontrolle einzelner Parameter. Diese Dialogfenster enthalten typischerweise Schieberegler, mit denen Benutzer die relevanten Parameter direkt anpassen können. Durch die enge Kopplung an die zugrunde liegende Uniform-Verwaltung wird die Kontrolle über Parameter von der GUI an die jeweiligen Dialoge übergeben. Dies ermöglicht eine klare Trennung zwischen allgemeiner Pipeline-Steuerung und der individuellen Feinkonfiguration einzelner Effekte. [20]

Ein typisches Beispiel ist die **Noise-Stärke**, die über einen Schieberegler angepasst werden kann. Änderungen an diesem Regler wirken sich unmittelbar auf die Rendering-Pipeline aus, sodass der Nutzer das Resultat in Echtzeit im Vorschaufenster nachvollziehen kann. Diese Live-Änderungen machen die Bedienung besonders intuitiv und unterstützen ein experimentelles Arbeiten mit verschiedenen Effekteinstellungen.

Abbildung 3.7 zeigt exemplarisch den Dialog für die Steuerung der Noise-Parameter.

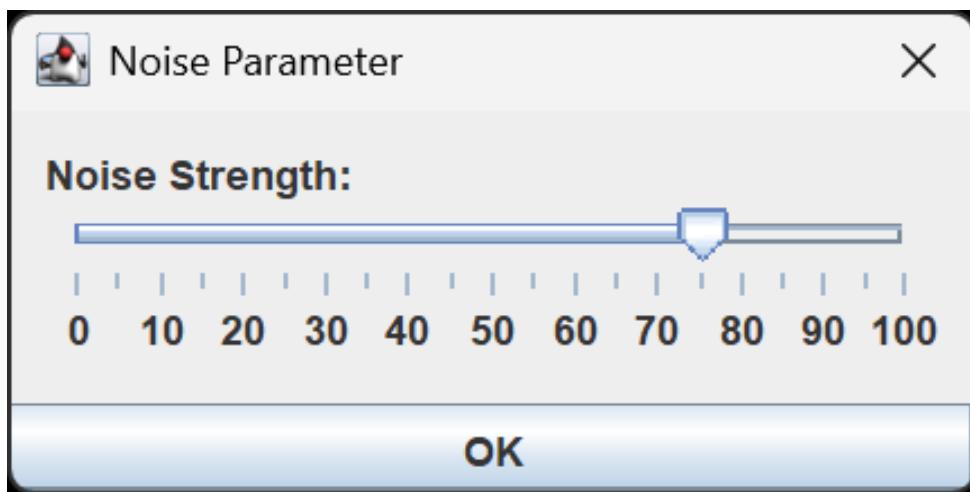


Abbildung 3.7.: Noise Dialog. Quelle: Eigene Darstellung

## 3.7. Testklassen und Qualitätssicherung

Die Implementierung von Testklassen stellt einen integralen Bestandteil einer robusten Softwareentwicklung dar und war daher ein wichtiger Aspekt bei der Konzeption des Frameworks. Sie dienen der Qualitätssicherung, indem sie eine automatisierte Verifizierung der Kernfunktionalitäten ermöglichen und eine stabile Basis für die Weiterentwicklung schaffen. Im Rahmen dieser Arbeit wurden umfassende Integrationstests mit dem JUnit-Framework entwickelt, um die korrekte Funktionsweise der zentralen Komponenten zu gewährleisten.

Dazu zählt der `PipelineManagerTest`, der die logische Reihenfolge der Effekte und die korrekte Datenübergabe in der Render-Pipeline verifiziert. Der `ShaderManagerTest` stellt sicher, dass Shader-Programme fehlerfrei geladen, kompiliert und verwaltet werden. Ebenso entscheidend ist der `ShaderPipelineTest`, der die korrekte Abarbeitung der gesamten Shader-Kette überprüft.

Die Funktionalität der für die Effektketten unerlässlichen Framebuffer-Objekte wird durch den `FramebufferObjectTest` sichergestellt, der die korrekte Erstellung und Nutzung dieser Objekte verifiziert. Darüber hinaus überprüft der `RendererTest` den Haupt-Rendering-Prozess der Anwendung, während Hilfsklassen für das Laden von Shadern und Texturen durch den `ShaderUtilsTest` und den `TextureUtilsTest` getestet werden.

Diese systematische Testung ermöglichte nicht nur eine frühzeitige Fehlererkennung und -behebung während der Entwicklung, sondern schafft auch eine solide Grundlage für zukünftige Erweiterungen [21]. Sie sorgt dafür, dass neue Effekte oder Funktionen hinzugefügt werden können, ohne die Stabilität der bereits bestehenden Architektur zu gefährden.

# 4. Implementierung der Effekte

Aufbauend auf der in Kapitel 3 beschriebenen Architektur, die Textur- und Shaderverwaltung, die Rendering-Pipeline, die Uniform-Steuerung sowie die Benutzeroberfläche umfasst, werden nun die konkreten Effekte implementiert. Die Implementierung der Effekte bildet den praktischen Kern der Arbeit. Ziel ist es, die zuvor konzipierte Rendering-Architektur in OpenGL und GLSL umzusetzen und dabei eine flexible, modulare Struktur zu schaffen. Grundlage bildet eine erweiterbare Rendering-Pipeline, in der einzelne Effekte als Shader implementiert und frei miteinander kombinierbar sind. Die Steuerung erfolgt über eine zentrale Uniform-Verwaltung sowie die angebundene Benutzeroberfläche, wodurch Parameteränderungen in Echtzeit sichtbar werden. Die enge Verzahnung von GUI, Shadern und Pipeline ermöglicht einen interaktiven Workflow. Das Kapitel gliedert sich in Pop-Art-Effekte (Abschnitt 4.1) und physikalische Effekte (Abschnitt 4.2).

## 4.1. Pop-Art-Effekte

Im Rahmen der in diesem Kapitel vorgestellten Implementierungsstrategie werden im Folgenden die Pop-Art-Effekte umgesetzt. Ziel ist es, typische Pop-Art-Ästhetiken wie vereinfachte Farbflächen, kontrastreiche Konturen und expressive Muster direkt im Rendering-Prozess zu realisieren. Jeder Effekt wird als eigenständiger GLSL-Fragment-Shader implementiert, wobei einige Effekte, wie der *Comic Look*, durch die Kombination mehrerer Shader realisiert werden, um komplexe visuelle Ergebnisse zu erzielen.

Die Steuerung der Effekte erfolgt über die zentrale Uniform-Verwaltung, die eine dynamische Anpassung von Parametern während der Laufzeit ermöglicht. Hierzu zählen beispielsweise Farbwerte. Nutzer können diese Parameter individuelle Effekt-Dialoge verändern, wobei Änderungen sofort im GLCanvas sichtbar werden, was einen interaktiven Workflow gewährleistet.

Die Effekte können sowohl einzeln als auch in beliebiger Kombination angewendet werden, was die kreative Gestaltung erleichtert. Durch die modulare Pipeline-Struktur ist die Einbindung neuer Effekte effizient möglich, da jeder Shader die standardisierte Schnittstelle nutzt. Damit bilden die Pop-Art-Effekte ein anschauliches Beispiel für die praktische Umsetzung der zuvor konzipierten Architektur in OpenGL und GLSL.

### 4.1.1. Posterisierung

Der Posterisierungs-Effekt ist ein grundlegendes Element der Pop-Art, das die Farbanzahl eines Bildes drastisch reduziert, indem es Farbtöne zu definierten Stufen zusammenfasst [22]. Die Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der die RGB-Farbwerthe jedes Pixels quantisiert. Dies wird mithilfe der `floor()`-Operation realisiert, bei der jeder Farbkanal auf eine vordefinierte Anzahl von diskreten Werten abgebildet wird [4]. Die Anzahl der Farbstufen wird als anpassbarer Parameter (`uniform float u_levels`)

über die GUI zur Verfügung gestellt, wodurch Nutzer die Intensität des Effekts dynamisch steuern können. Dieses Vorgehen ermöglicht eine flexible Anpassung, um von einer subtilen Farbkorrektur bis zu einer stark vereinfachten, flächigen Ästhetik zu gelangen.

Das visuelle Ergebnis ist in Abbildung A.1 im Anhang dargestellt.

#### 4.1.2. Duotone

Der Duotone-Effekt ist ein klassisches Stilmittel, das die visuelle Komplexität eines Bildes reduziert, indem es dessen Graustufen auf eine Palette aus zwei definierten Farbtönen abbildet. Die technische Umsetzung erfolgt mithilfe eines **GLSL-Fragment-Shaders**. Dieser berechnet zunächst den Luminanzwert jedes Pixels durch den **dot**-Produkt des RGB-Wertes mit einer gewichteten Vektorkonstante. Anschließend wird dieser Luminanzwert als Interpolationsfaktor genutzt, um eine Mischung zwischen zwei **Uniform-Farben** zu erzeugen. Die **Besonderheit** dieses Ansatzes liegt in der dynamischen Steuerung: Die beiden Farbtöne können über die grafische Benutzeroberfläche in Echtzeit verändert werden, was eine flexible Anpassung des visuellen Ausdrucks ermöglicht. [23]

Das visuelle Ergebnis ist in Abbildung A.2 im Anhang dargestellt.

#### 4.1.3. Tritone

Der Tritone-Effekt stellt eine Erweiterung des Duotone-Ansatzes dar und dient dazu, einem Bild eine stärkere visuelle Tiefe und Nuancierung zu verleihen, indem die Graustufen auf drei Farbbereiche abgebildet werden: Schatten, Mitteltöne und Highlights. Die Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der für jedes Pixel die Luminanz berechnet. Dieser einzige Helligkeitswert wird anschließend in eine von drei vordefinierten Farbtönen übersetzt. Technisch wird dies durch bedingte Anweisungen (**if/else-Statements**) realisiert, welche die Luminanz mit festen Schwellenwerten vergleichen, um zu entscheiden, welche der drei Farben für das Pixel verwendet wird. Die größte **Besonderheit** dieses Implementierungsansatzes liegt in seiner Einfachheit und hohen Performance. Da auf komplexe Interpolationsberechnungen verzichtet wird, ermöglicht der Shader eine ressourcenschonende und direkte Anwendung des Effekts. Die drei Farbtöne werden über **Uniform-Variablen** gesteuert, wodurch der Effekt in Echtzeit anpassbar ist und eine schnelle, interaktive Gestaltung ermöglicht. [23]

Das visuelle Ergebnis ist in Abbildung A.3 im Anhang dargestellt.

#### 4.1.4. Selective Color Boost

Der „Selective Color Boost“-Effekt dient der gezielten Verstärkung bestimmter Farbbereiche eines Bildes, um eine erhöhte visuelle Prägnanz zu erreichen, die an die oft leuchtenden Farben der Pop-Art erinnert. Die Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der für jedes Pixel dessen Farbwert analysiert. Der Shader berechnet den **euklidischen Abstand im RGB-Farbraum** zwischen der aktuellen Pixelfarbe und einer vordefinierten Zielfarbe (**targetColor**), die über eine Uniform-Variablen gesteuert wird. Liegt dieser Abstand unter einem bestimmten Schwellenwert (**threshold**), wird die Sättigung und Helligkeit der Pixelfarbe mithilfe eines Multiplikators (**boostAmount**) verstärkt, während alle anderen Farben entsättigt und in Graustufen umgewandelt werden. Diese **Besonderheit** in der Handhabung der Uniform-Variablen ermöglicht eine dynamische

Anpassung in Echtzeit, wodurch Nutzer flexibel entscheiden können, welcher Farbbereich hervorgehoben und wie stark die Verstärkung ausfallen soll. [23]

Das visuelle Ergebnis ist in Abbildung A.4 im Anhang dargestellt.

#### 4.1.5. Halftone

Der Halftone-Effekt ist ein klassisches Stilmittel der Druckgrafik und simuliert das charakteristische Raster von Zeitungen oder Comics. Er wandelt ein kontinuierliches Bild in ein Muster aus Punkten um, deren Größe oder Dichte die Helligkeit des Quellbildes repräsentiert. Die technische Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der ein regelmäßiges, gitterbasiertes Punktraster generiert. Innerhalb jeder Zelle des Rasters wird der Radius des zu zeichnenden Punktes dynamisch an die Luminanz des entsprechenden Bildbereichs angepasst. Dunkle Bereiche werden dabei durch größere Punkte, helle Bereiche durch kleinere oder fehlende Punkte repräsentiert. Die **Besonderheit** dieses Ansatzes liegt in der dynamischen Steuerung der Raster- und Punktgröße über Uniform-Variablen, was eine Echtzeit-Anpassung ermöglicht. Zudem sorgt die `smoothstep()`-Funktion für weiche, fließende Kanten der Rasterpunkte, wodurch der visuelle Effekt präziser und ästhetisch ansprechender wirkt. [24]

Das visuelle Ergebnis ist in Abbildung A.5 im Anhang dargestellt.

#### 4.1.6. Benday Dots

Der Benday Dots-Effekt ist eine Hommage an die klassische Comic-Drucktechnik, bei der Farben durch die Rasterung mit farbigen Punkten erzeugt werden. Ziel ist es, dem Bild ein charakteristisches, aus der Nähe sichtbares Punktmuster zu verleihen, das aus der Ferne als eine durchgehende Farbfläche wahrgenommen wird. Die Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der ein präzises, gitterförmiges Muster erzeugt. Innerhalb jeder Zelle wird der Punkt-Radius basierend auf der Helligkeit des Quellbildes berechnet. Die größte **Besonderheit** dieses Effekts liegt in der Verwendung der originalen Bildfarben für die Punkte, was einen entscheidenden Unterschied zum Halftone-Effekt darstellt, bei dem Punkte üblicherweise nur in Graustufen gerendert werden. Die Möglichkeit, die Rastergröße über eine **Uniform-Variable** dynamisch anzupassen, unterstreicht die Flexibilität der Architektur und ermöglicht eine genaue Nachbildung verschiedener Druckstile. [25]

Das visuelle Ergebnis ist in Abbildung A.6 im Anhang dargestellt.

#### 4.1.7. Out-of-Register Print

Der „Out-of-Register Print“-Effekt simuliert einen analogen Druckfehler, bei dem die einzelnen Farbplatten leicht versetzt zueinander gedruckt werden. Dieses Verfahren verleiht einem digitalen Bild eine charakteristische Ästhetik, die an alte Comic-Bücher oder Zeitschriften erinnert. Die technische Umsetzung erfolgt durch einen **GLSL-Fragment-Shader**, der jeden Farbkanal separat verarbeitet. Anstatt die Farbe eines Pixels nur einmal abzufragen, greift der Shader an mehreren Stellen auf die Textur zu: Er verschiebt den Rot- und den Blau-Kanal um einen bestimmten Vektor, während der Grün-Kanal in der Originalposition bleibt. Die **Besonderheit** dieses Effekts liegt in der Verwendung eines **Uniforms** für den Versatz (`u_offset`), was die dynamische Steuerung der Intensität

ermöglicht. So kann der Nutzer den Grad des Versatzes in Echtzeit anpassen und eine Bandbreite von subtilen Farbsäumen bis zu stark separierten Kanälen erzeugen. [23]

Das visuelle Ergebnis ist in Abbildung A.7 im Anhang dargestellt.

#### 4.1.8. Comic Look

Der Comic Look-Effekt ist eine Kombination aus zwei klassischen Bildbearbeitungs-Methoden, die zusammen die charakteristische Ästhetik von gedruckten Comics simulieren: die Vereinfachung von Farben und die Hervorhebung von Konturen. Anstatt diese Effekte nacheinander in einer Rendering-Pipeline zu verarbeiten, werden hier beide Schritte innerhalb eines einzigen, komplexen **GLSL-Fragment-Shaders** ausgeführt. Zuerst wird die Farbe des Bildes durch Posterisierung auf eine begrenzte Anzahl von Farbstufen reduziert, wodurch die flächigen, unschattierten Bereiche entstehen. Parallel dazu wird eine Kantenerkennung, die auf einem Algorithmus wie dem **Sobel-Filter** basiert, angewendet, um signifikante Helligkeitsänderungen zu identifizieren. Diese Konturen werden anschließend als schwarze Linien über das posterisierte Bild gelegt. Die **Besonderheit** dieses Ansatzes liegt in der direkten und performanten Verarbeitung: Da beide Operationen in einem einzigen Shader-Durchlauf stattfinden, entfällt der Overhead, der bei der Übergabe von Daten zwischen mehreren Shadern entstehen würde. Dies ermöglicht eine ressourcenschonende Umsetzung, während Parameter wie die Anzahl der Farbstufen (**u\_levels**) dynamisch in Echtzeit angepasst werden können. [25]

Das visuelle Ergebnis ist in Abbildung A.8 im Anhang dargestellt.

#### 4.1.9. Intelligent Bold Outlines

Der Effekt „Intelligent Bold Outlines“ dient dazu, eine Art von Kanten zu extrahieren, die über die einfachen Konturen des Bildes hinausgeht und die visuelle Ästhetik einer handgezeichneten Skizze imitiert. Die als „intelligent“ bezeichnete Qualität des Effekts beruht auf seiner Fähigkeit, die stärksten Linien dynamisch auf Basis des Bildinhalts hervorzuheben. Die technische Umsetzung wird durch einen **GLSL-Fragment-Shader** realisiert, der für jedes Pixel die Luminanz seiner Umgebung in einem 3x3-Muster analysiert. Hierfür wird ein Algorithmus wie der **Sobel-Filter** eingesetzt, der die Helligkeitsgradienten horizontal und vertikal berechnet. Die Ergebnisse dieser Berechnungen geben Aufschluss über die Stärke einer Kante an der jeweiligen Pixelposition. Diese **Besonderheit** in der Handhabung der Uniform-Variablen ermöglicht eine feingranulare Steuerung: Über einen Uniform für die **Kantendicke** (**u\_thickness**) kann die Stärke der Linie angepasst werden. Zudem erlaubt ein variabler **Schwellenwert** (**u\_edgeThreshold**) eine dynamische Entscheidung darüber, welche Kanten als relevant genug erachtet werden, um gezeichnet zu werden. Durch diese Parameter kann der Nutzer die Linienführung interaktiv steuern und den Grad der Detailtreue flexibel anpassen. [25]

Das visuelle Ergebnis ist in Abbildung A.9 im Anhang dargestellt.

#### 4.1.10. Seriality

Der Seriality-Effekt ist eine Hommage an die Pop-Art, insbesondere an die Mehrfachdrucke von Künstlern wie Andy Warhol, bei denen dasselbe Bild in einem Raster wiederholt wird [22]. Die technische Umsetzung erfolgt mithilfe eines **GLSL-Fragment-Shaders**,

der die Texturkoordinaten neu berechnet. Durch die Verwendung der `fract()`-Funktion werden die Koordinaten in regelmäßigen Abständen auf den Bereich von 0 bis 1 zurückgesetzt, wodurch die Textur mehrfach als Kachel nebeneinander wiederholt wird [4]. Die **Besonderheit** liegt in der dynamischen Steuerung der Wiederholungen. Über eine **Uniform-Variablen** (`u_repeat`) kann der Nutzer die Anzahl der Kacheln in X- und Y-Richtung in Echtzeit anpassen.

Das visuelle Ergebnis ist in Abbildung A.10 im Anhang dargestellt.

## 4.2. Physikalische Effekte

Im Rahmen der in diesem Kapitel vorgestellten Implementierungsstrategie werden die Physikalischen Effekte umgesetzt. Der zugrunde liegende Ansatz ist modular: Jeder Effekt wird als eigener, abgeschlossener **GLSL-Shader** realisiert, der darauf abzielt, reale, physikalische Phänomene zu simulieren, die die Lichtwahrnehmung beeinflussen. Diese Shader manipulieren entweder **Vertex-** oder **Fragmente-Daten** oder beides, um die gewünschten visuellen Transformationen zu erzielen. Viele Effekte basieren dabei auf komplexen mathematischen Modellen, die eine realistische Simulation ermöglichen.

Ein Hauptmerkmal dieser Architektur ist die hohe Flexibilität: Durch die modulare Umsetzung können die Effekte einzeln angewendet oder in Echtzeit dynamisch miteinander kombiniert werden. Über einen Parameter Dialog können die Einstellungen der Effekte mithilfe von **Uniforms** gesteuert werden, was eine interaktive Echtzeit-Vorschau erlaubt. Die Effekte sind breit anwendbar, von der Simulation natürlicher Umgebungen (z. B. Wasser, Hitze) bis hin zur Imitation von Kamerafehlern (z. B. Chromatic Aberration).

### 4.2.1. Bloom

Der Bloom-Effekt ist ein zentraler Bestandteil der physikalischen Effekte und dient dazu, die visuelle Überstrahlung von Lichtquellen zu simulieren, wie sie typischerweise bei der Aufnahme mit einer Kamera entsteht. In dieser Implementierung wird ein spezieller Ansatz verfolgt, um das Überstrahlen auf effiziente Weise zu realisieren.

Die Umsetzung erfolgt in einem **Single-Pass-Rendering-Verfahren**, bei dem ein einziger **Fragmente-Shader** das gesamte Bild verarbeitet. Dieser Shader wendet eine Faltungsmethode an, um die Textur zu verwischen, wobei das gesamte Bild in die Berechnung einbezogen wird. Das Ergebnis dieser Glättung wird anschließend mit den ursprünglichen Bilddaten kombiniert, um den gewünschten Überlagerungseffekt zu erzeugen. [1]

Für die Steuerung der Effekteigenschaften nutzt der Shader spezifische **Uniforms**. Die **Blur-Größe** der Faltung kann über den Parameter `u_texelSize` angepasst werden, während die Intensität des Überstrahlens durch `u_intensity` modifiziert wird. Ein `u_threshold` Parameter zur Isolierung heller Bereiche wird in diesem spezifischen Implementierungsansatz bewusst nicht verwendet, da das gesamte Bild in die Faltungsberechnung einbezogen wird.

Trotz der Implementierung in einem einzigen Durchgang simuliert dieser Ansatz das physikalische Phänomen des Überbelichtens effektiv durch eine einfache Überlagerung von Originalbild und Weichzeichner. Dieser Ansatz bietet eine sehr performante Methode, um einen ästhetisch ansprechenden Effekt zu erzielen.

Das visuelle Ergebnis ist in Abbildung A.11 im Anhang dargestellt.

### 4.2.2. Chromatic Aberration

Der Effekt der **Chromatischen Aberration** zielt darauf ab, die optische Unvollkommenheit einer Linse zu simulieren, die zu Farbsäumen an den Rändern von Objekten führt [19]. Die Umsetzung erfolgt in einem **Fragment-Shader**, wobei die Farbkanäle Rot, Grün und Blau eines Pixels um einen festen, horizontalen Betrag verschoben werden, um das Farbverschieben zu erzeugen. Die Stärke der Verschiebung wird über die **Uniform u\_offset** gesteuert. Im Gegensatz zu komplexeren, radialen Implementierungen wird bei diesem Ansatz die Verschiebung der Farbkanäle ohne Berücksichtigung des Abstands zum Bildzentrum berechnet. Durch diese einfache, aber effektive Transformation imitiert der Shader einen charakteristischen Linsenfehler, was dem Bild eine realistische, kamerabasierte Ästhetik verleiht.

Das visuelle Ergebnis ist in Abbildung A.12 im Anhang dargestellt.

### 4.2.3. Distortion

Der Effekt **Distortion** zielt darauf ab, die Texturkoordinaten wellenförmig zu verzerrn, um ein Bild zu erzeugen, das sich wie eine wogende Welle bewegt [26]. Die Umsetzung erfolgt in einem **Fragment-Shader**, der jeden Pixel basierend auf seiner horizontalen Position und der Zeit manipuliert. Die Stärke der Welle wird über die **Uniform u\_amplitude** gesteuert, während die Häufigkeit der Wellen mithilfe von **u\_frequency** angepasst werden kann. Durch diese prozedurale, mathematische Berechnung wird eine **stilisierte optische Wirkung** erzielt, die keine optische Illusion, sondern eine gezielte visuelle Transformation ist. Das Ergebnis erzeugt einen dynamischen, flüssigen Effekt, der oft für die Erzeugung abstrakter Welleneffekte oder surrealer Transformationen genutzt wird.

Das visuelle Ergebnis ist in Abbildung A.13 im Anhang dargestellt.

### 4.2.4. Chromatic Wave Distortion

Der Effekt der **Chromatic Wave Distortion** stellt eine komplexe visuelle Transformation dar, indem er zwei separate, aber miteinander verbundene Shader-Techniken kombiniert. Die Umsetzung dieses Effekts ist statisch und nicht zeitbasiert, wodurch er eine dauerhafte wellenartige Verformung des Objekts erzeugt. Eine **wellenartige Verformung der Geometrie** wird im **Vertex-Shader** realisiert, wo die Positionen jedes Vertices mathematisch manipuliert werden. Auf dieser Grundlage wird im **Fragment-Shader** eine **chromatische Trennung der Farben** angewendet, indem die einzelnen Farbkanäle des Bildes leicht gegeneinander verschoben werden. [26]

Diese Effektsimulation wird durch das Zusammenspiel der verschiedenen Shader-Typen ermöglicht. Uniforms wie **u\_amplitude** und **u\_frequency** werden verwendet, um die statische Form und die Dichte der Welle zu definieren. Im Vertex-Shader wird die Stärke der chromatischen Verschiebung auf einen fixierten Wert gesetzt und anschließend als veränderte Texturkoordinate an den Fragment-Shader weitergegeben. Durch diese Kommunikation und Arbeitsteilung zwischen den Shadern wird eine stilisierte optische Verzerrung geschaffen, die visuell komplexer ist als Effekte, die nur in einem einzigen Shader-Typ umgesetzt werden. Diese Methode demonstriert somit eine fortgeschrittene Nutzung der

Grafikpipeline, um ein physikalisches Phänomen durch die geschickte Kombination von Geometrie- und Pixelfiltern zu simulieren.

Das visuelle Ergebnis ist in Abbildung A.14 im Anhang dargestellt.

#### 4.2.5. Heat Distortion

Der Effekt **Heat Distortion** simuliert die visuelle Verzerrung, wie sie über heißen Oberflächen auftritt. Die Umsetzung erfolgt in einem **Fragment-Shader**, der jeden Pixel des Bildes basierend auf einer komplexen Wellenfunktion verschiebt [26]. Statt einer einfachen Sinuswelle werden hier verschachtelte Sinusfunktionen genutzt, um ein organisches, flüssigeres Muster zu erzeugen. Die Stärke der Verzerrung kann durch die **Uniform u\_strength** und die Dichte der Wellen durch **u\_frequency** angepasst werden. Das Ergebnis ist eine realistische, Fata-Morgana-artige Verzerrung, die die Hitze als eine visuell spürbare Kraft im Bild darstellt und dem Betrachter einen Eindruck von extremer Hitze vermittelt.

Das visuelle Ergebnis ist in Abbildung A.15 im Anhang dargestellt.

#### 4.2.6. Water Ripple

Der Effekt **Water Ripple** erzeugt eine statische, wellenartige Verformung der Geometrie eines Objekts [19]. Die Umsetzung erfolgt in einem **Vertex-Shader**, der jeden Vertex des Gitters mathematisch neu positioniert, um die Wellenform zu erzeugen. Hierbei wird die Position der Vertices ausschließlich basierend auf ihrer horizontalen Position neu berechnet. Die statische Form der Welle wird durch die **Uniforms u\_amplitude** gesteuert, welche die Stärke der Verformung definiert, und **u\_frequency**, welche die Dichte der Wellen bestimmt. Eine Besonderheit dieses Ansatzes liegt darin, dass es sich um einen rein **geometrischen Effekt** handelt, der das Objektgitter direkt manipuliert, anstatt nur die Farbe der Pixel zu verändern.

Das visuelle Ergebnis ist in Abbildung A.16 im Anhang dargestellt.

#### 4.2.7. Refraction

Der **Refraction**-Effekt bietet eine vereinfachte, statische Simulation der Lichtbrechung, wie sie bei der Durchsicht durch eine unebene Wasseroberfläche oder ein verzogenes Glas auftritt. Die Umsetzung dieses visuellen Phänomens erfolgt in einem **Fragment-Shader**, der jeden Pixel des Bildes prozedural manipuliert. Anstatt physikbasierte Berechnungen durchzuführen, verschiebt der Shader die Texturkoordinaten wellenförmig mithilfe einer mathematischen Funktion wie dem Sinus [26]. Die Stärke dieser Verzerrung lässt sich über die **Uniform u\_refractionStrength** anpassen. Als Besonderheit dieser Implementierung ist die grundlegende Form der Welle, einschließlich ihrer Frequenz und Amplitude, im Code fest definiert. Dies macht den Effekt effizient und vorhersehbar, indem eine visuell ansprechende Annäherung an ein physikalisches Prinzip geschaffen wird.

Das visuelle Ergebnis ist in Abbildung A.17 im Anhang dargestellt.

#### 4.2.8. Vignette

Der **Vignette**-Effekt stellt eine weit verbreitete Technik in der Bildverarbeitung dar, bei der die Ränder eines Bildes progressiv abgedunkelt werden. Die Stärke der Abdunklung hängt dabei direkt vom Abstand jedes Pixels zum Bildzentrum ab. Diese Art der visuellen Transformation wird in einem **Fragment-Shader** implementiert. Der Shader berechnet für jeden Pixel seinen Abstand vom Zentrum und nutzt diese Information, um den entsprechenden Abdunklungsfaktor zu bestimmen. Für die Feinsteuerung des Effekts werden die Uniforms `u_radius` und `u_softness` verwendet: Ersteres definiert den inneren Bereich des Bildes, der unbeeinflusst bleibt, während Letzteres die Weichheit und den Übergang der Abdunklung von der Mitte zu den Rändern steuert. Technisch wird hierfür oft die `smoothstep()`-Funktion eingesetzt, um einen weichen Übergang zu erzeugen [26]. Die Besonderheit der Vignette liegt in ihrer Eigenschaft, einen natürlichen Linsenfehler zu imitieren und damit die Aufmerksamkeit des Betrachters gezielt auf den zentralen Bildbereich zu lenken, wodurch eine fokussierte und oft dramatische Wirkung erzielt wird.

Das visuelle Ergebnis ist in Abbildung A.18 im Anhang dargestellt.

#### 4.2.9. Noise

Der Effekt **Noise** simuliert die visuelle Textur, die bei älteren analogen Medien wie Film oder bei digitalen Sensoren bei schlechten Lichtverhältnissen entsteht. Die Umsetzung erfolgt in einem **Fragment-Shader**, der jedem Pixel einen festen, zufälligen Wert hinzufügt, um ein feines Rauschen zu erzeugen. Die Rauschwerte werden dabei nicht aus einer statischen Textur entnommen, sondern prozedural berechnet. Hierfür wird ein **Zufallsgenerator** verwendet, dessen Werte von den Texturkoordinaten abhängen [26]. Die Intensität des Effekts kann über die Uniform `u_strength` angepasst werden. Die Besonderheit des Noise-Effekts liegt in seiner Fähigkeit, eine analoge oder retro-Ästhetik zu erzeugen. Durch das Hinzufügen dieser körnigen Textur wird die visuelle Qualität eines älteren Mediums nachgeahmt, was dem Bild eine authentische, unperfekte Optik verleiht und seinen visuellen Stil prägt.

Das visuelle Ergebnis ist in Abbildung A.19 im Anhang dargestellt.

#### 4.2.10. Scanline

Der Effekt **Scanline** ist ein visueller Filter, der das Bild mit dunklen, horizontalen Linien überlagert, um das Zeilenraster alter Röhrenmonitore (CRTs) nachzuahmen. Die Umsetzung erfolgt in einem **Fragment-Shader**, der für jeden Pixel eine Berechnung auf der y-Koordinate anwendet. Hierfür werden die Funktionen `fract()` und `step()` kombiniert, um ein sich wiederholendes, binäres Muster zu erzeugen [26]. Die Breite der Scanlines lässt sich über die Uniform `u_scanlineWidth` steuern. Die Besonderheit dieses Effekts liegt in seiner Fähigkeit, die nostalgische Ästhetik von „Old-School“-Computergrafiken nachzubilden und damit eine charakteristische retro-visuelle Wirkung zu erzielen.

Das visuelle Ergebnis ist in Abbildung A.20 im Anhang dargestellt.

# 5. Ergebnisse und Evaluation

Dieses Kapitel dient der systematischen Evaluation und Analyse der im Projekt implementierten Shader. Anstatt jeden Effekt einzeln und isoliert zu betrachten, liegt der Fokus hier auf der Demonstration ihrer visuellen Wirkung und ihrer Synergie in der Kombination. Das Vorgehen ermöglicht eine ganzheitliche Beurteilung der Effizienz und des kreativen Potenzials des entwickelten Frameworks. Die Analyse gliedert sich in zwei Sektionen: In der ersten Sektion, „**Auswahl und Beschreibung der Ergebnisbilder**“, werden vier repräsentative Bilder ausgewählt und deren zugrundeliegende Testbilder begründet. Anschließend folgt in der zweiten Sektion, „**Analyse und Vergleich der Ergebnisbilder**“, eine detaillierte Untersuchung dieser Ergebnisse, um ihre spezifischen Eigenschaften und ihre gegenseitige Wirkung zu analysieren.

## 5.1. Auswahl und Beschreibung der Ergebnisbilder

Die folgenden vier Bilder wurden als repräsentative Beispiele für die vielfältigen Einsatzmöglichkeiten der entwickelten Shader ausgewählt. Jedes Bild kombiniert einen Pop-Art-Effekt mit mindestens einem physikalischen Effekt, um die Brücke zwischen künstlerischer Stilisierung und technischer Simulation zu schlagen. Die Absicht hinter jeder Kombination wird im Folgenden erläutert, um die strategische Grundlage für die darauffolgende Evaluation zu legen.

### 5.1.1. Kombinationsbeispiel 1: Analoger Glitch-Look

Der erste Anwendungsfall zielt auf die Erzeugung eines analogen Glitch-Looks ab und kombiniert dafür die Shader „**Out-of-Register Print**“ und „**Noise**“. Für diese Effektkombination wurde das Testbild „**Pferd im Feld**“ gezielt ausgewählt. Die reichhaltige Farbpalette, die markanten Hell-Dunkel-Kontraste und die detaillierten Texturen des Originalbildes boten eine ideale Grundlage, um die spezifischen Eigenheiten der beiden Effekte optimal sichtbar zu machen. Die visuelle Absicht dieser Kombination war es, die künstliche Verzerrung des **Out-of-Register Print**-Shaders mit den zufälligen, körnigen Texturen des **Noise**-Shaders zu verbinden. Die Wahl eines Bildes mit klaren Übergängen und Strukturen ermöglichte es zudem, die Interaktion der Effekte an diesen kritischen Bereichen zu testen und zu evaluieren, wie die druckähnlichen Fehler durch das analoge Rauschen verstärkt und geerdet werden. Das Ergebnis sollte die visuelle Ästhetik eines kaputten, alten Bildschirms imitieren und so die Brücke zwischen einem künstlichen Druckfehler und einem physikalischen Rauschen schlagen.

Die Gegenüberstellung des Originalbildes und des visuellen Ergebnisses ist in Abbildung 5.1 dargestellt.



Abbildung 5.1.: Gegenüberstellung des Originalbildes und des analogen Glitch-Looks durch die Kombination der Shader Out-of-Register Print und Noise.

### 5.1.2. Kombinationsbeispiel 2: Nostalgischer Zeitungsdruck

Für die zweite Kombinationsanalyse wurde der nostalgische Zeitungsdruck-Stil gewählt, der durch das Zusammenspiel der Shader „**Halftone**“ und „**Vignette**“ erzeugt wird. Auch hier diente das Testbild des Pferdes als ideale Grundlage. Die Detailvielfalt und die breite Palette an Grauwerten des Bildes boten eine hervorragende Möglichkeit, die Funktionsweise des **Halftone**-Effekts zu demonstrieren. Dieser Effekt wandelt die Helligkeitsinformationen des Originals in ein Punktraster um, dessen Dichte die Graustufen repräsentiert. Die visuelle Absicht dieser Kombination war es, diese Punktrasterung mit der atmosphärischen Rahmung der **Vignette** zu verbinden. Das Ergebnis ist ein kohärenter Stil, der die Ästhetik alter, gedruckter Fotos nachahmt. Das Testbild war aufgrund seiner komplexen Helligkeitsbereiche und feinen Texturen besonders geeignet, um zu zeigen, wie der **Halftone**-Effekt auch subtile Farbübergänge überzeugend in ein Schwarz-Weiß-Punktmuster übersetzt.

Die Gegenüberstellung des Originalbildes und des visuellen Ergebnisses ist in Abbildung 5.2 dargestellt.

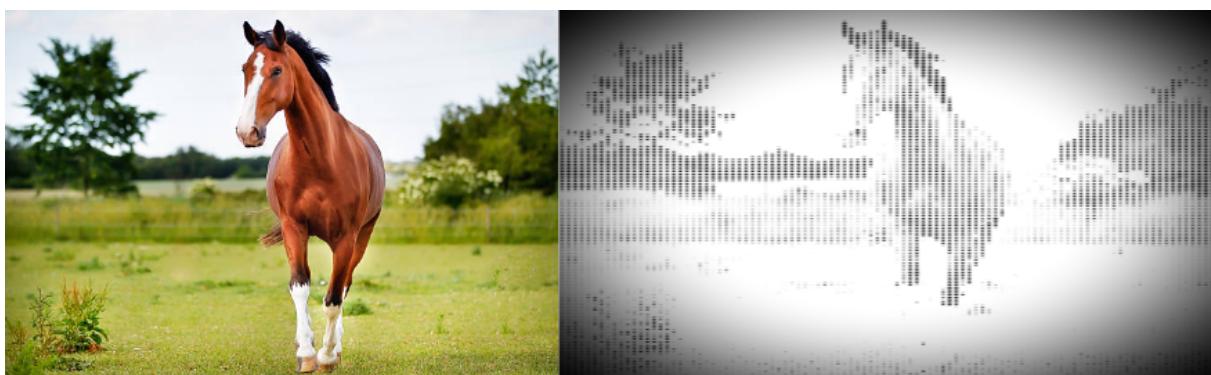


Abbildung 5.2.: Gegenüberstellung des Originalbildes und des nostalgischen Zeitungsdrucks durch die Kombination der Shader Halftone und Vignette.

### 5.1.3. Kombinationsbeispiel 3: Dramatische Verzerrung

Für die dritte Kombinationsanalyse, „**Dramatische Verzerrung**“, wurden die Shader **Heat Distortion**, **Refraction** und **Posterisierung** miteinander kombiniert. Die Wahl des Testbildes eines Waldbrandes war hierbei gezielt und thematisch motiviert. Die extremen Hell-Dunkel-Kontraste, die leuchtenden Farben der Flammen und die inhärente Thematik der Hitze boten die idealen Voraussetzungen, um die physikalischen Effekte der Lichtbrechung und thermischen Verzerrung überzeugend zu simulieren. Das übergeordnete Ziel war es, die visuelle Kraft der physikalischen Shader zu demonstrieren. Die Kombination von **Heat Distortion** und **Refraction** wurde gewählt, um die physikalischen Eigenschaften von Hitze und Lichtbrechung zu imitieren und so eine authentische simulierte Verformung zu erzeugen. Parallel dazu ermöglichte die **Posterisierung** eine künstlerische Abstraktion der Szene, indem sie die fließenden Farbübergänge in harte, stilisierte Flächen auflöste. Die Kombination zielt somit auf eine effektvolle Überhöhung der Realität ab, die sowohl physikalisch plausibel als auch künstlerisch expressiv ist.

Die Gegenüberstellung des Originalbildes und des visuellen Ergebnisses ist in Abbildung 5.3 dargestellt.

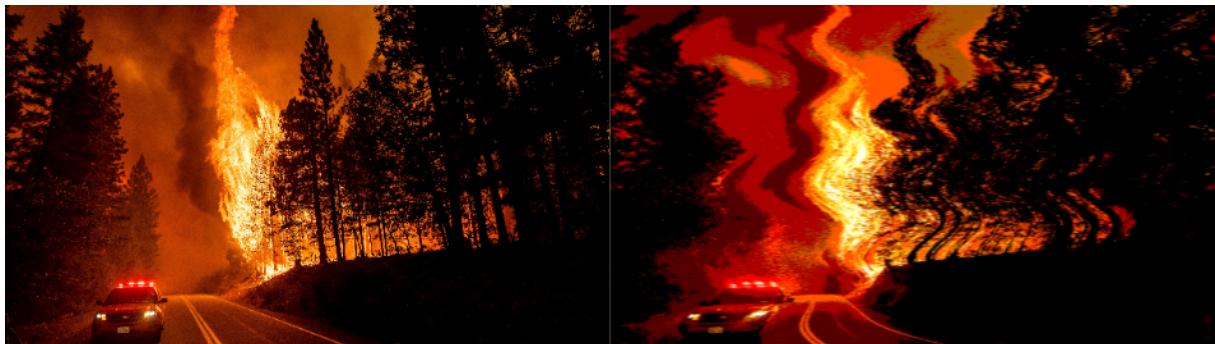


Abbildung 5.3.: Gegenüberstellung des Originalbildes und der dramatischen Verzerrung durch die Kombination der Shader Heat Distortion, Refraction und Posterisierung.

### 5.1.4. Kombinationsbeispiel 4: Surrealistische Wasserlandschaft

Für die letzte Kombinationsanalyse, „**Surrealistische Wasserlandschaft**“, wurden die Shader **Tritone** und **Water Ripple** gewählt. Die gezielte Entscheidung für das Testbild einer Flusslandschaft war hierbei von entscheidender Bedeutung, da seine flache und weitläufige Wasseroberfläche die perfekte Leinwand bot. Diese Grundlage ermöglichte es, die physikalisch korrekte Wellenbewegung des **Water Ripple**-Shaders visuell an einer idealen Oberfläche zu demonstrieren. Das übergeordnete Ziel dieser Kombination war es, die Interaktion und den synergetischen Effekt eines künstlerischen Pop-Art-Farbilters mit einer physikalischen Simulation zu demonstrieren. Während der **Tritone**-Shader die Farbpalette der gesamten Landschaft auf einen stilisierten Farbbereich reduzierte, nutzte der **Water Ripple**-Shader diese veränderte Bildinformation, um eine realistische, visuelle Simulation einer Wellenbewegung zu erzeugen. Das Testbild bot die ideale Grundlage, um diese Verbindung zwischen künstlerischer Abstraktion und physikalischer Präzision in einem kohärenten und ästhetisch ansprechenden Ergebnis zu vereinen.

Die Gegenüberstellung des Originalbildes und des visuellen Ergebnisses ist in Abbildung 5.4 dargestellt.



Abbildung 5.4.: Gegenüberstellung des Originalbildes und der surrealistischen Wasserlandschaft durch die Kombination der Shader Tritone und Water Ripple.

## 5.2. Analyse und Vergleich der Ergebnisbilder

Aufbauend auf der Auswahl und Beschreibung der jeweiligen Effektkombinationen in der vorherigen Sektion, liegt der Fokus dieses Abschnitts auf einer detaillierten Analyse und dem direkten Vergleich der Ergebnisse. Dabei werden die visuelle Wirkung, die Synergie der Effekte sowie die Stärken und Schwächen jeder Kombination evaluiert, um eine ganzheitliche Bewertung der vielfältigen Anwendungsmöglichkeiten des Frameworks zu ermöglichen.

Die visuelle Wirkung der ersten Effektkombination, die in Abbildung 5.1 dargestellt ist, ist ein markanter Glitch-Look, der dem Endergebnis ein kaputtes und fehlerhaftes Aussehen verleiht, das an das Bild eines alten, analogen Bildschirms erinnert. Die **Synergie** der beiden Effekte ist dabei von zentraler Bedeutung: Während der **Out-of-Register Print**-Shader die künstlichen, druckähnlichen Fehler erzeugt, wird diese Verzerrung durch das körnige Rauschen des **Noise**-Shaders verstärkt und geerdet. Letzterer fügt eine unvorhersehbare, organische Textur hinzu, die dem statischen Druckfehler eine dynamische Anmutung verleiht und das Gesamtbild weniger steril wirken lässt. Zu den **Stärken** dieser Kombination zählt die Fähigkeit, einen authentischen, analogen Look zu erzeugen, der insbesondere bei Bildern mit hohen Kontrasten seine volle Wirkung entfaltet. Die Zufälligkeit des Rauschens macht jedes Ergebnis zudem einzigartig. Eine wesentliche **Schwäche** liegt jedoch darin, dass die Effekte das Bild leicht zu chaotisch wirken lassen und feine Details des Originalmotivs verloren gehen können. Das Aussehen kann gezielt über die **Parameter** gesteuert werden, indem beispielsweise die Stärke des **Noise**-Effekts oder der Offset des **Out-of-Register Print**-Shaders angepasst wird.

Das Ergebnis der zweiten Kombination, die in Abbildung 5.2 dargestellt ist, ist ein Bild im Stil eines Schwarz-Weiß-Drucks aus einer alten Zeitung, das einen gealterten und nostalgischen Eindruck vermittelt. Die **Synergie** der Effekte **Halftone** und **Vignette** ist hierbei besonders minimalistisch, aber hochwirksam. Der **Halftone**-Shader wandelt die Helligkeitsinformationen des Originalbildes präzise in ein Punktraster um, wodurch der

typische Zeitungsdruck-Look entsteht. Die **Vignette** ergänzt dies durch eine dunkle, atmosphärische Rahmung, die den Effekt eines alten, ausgeblichenen Fotos unterstreicht. Die **Stärke** dieser Kombination liegt in der Effizienz, mit der sie einen sofort erkennbaren und überzeugenden Stil erzeugt. Ihre Einfachheit ist dabei ein großer Vorteil. Allerdings ist eine wesentliche **Schwäche**, dass der Stil sehr spezifisch ist und sich kaum für andere visuelle Ziele anwenden lässt. Besonders bei sehr hellen oder kontrastarmen Bildern kann die Überzeugungskraft des Punktrasters verloren gehen. Über die **Parameter** kann die Wirkung gezielt beeinflusst werden: Die Punktdichte des **Halftone**-Shaders bestimmt die wahrgenommene Auflösung, während die Intensität der **Vignette** die Stimmung und Dramatik des Bildes steuert.

Das visuelle Ergebnis der dritten Kombination, die in Abbildung 5.3 dargestellt ist, ist eine dramatisch verzerrte und farblich stilisierte Szene, die die Hitze und Bewegung der Flammen visuell überhöht. Die **Synergie** dieser Effekte ist besonders wirkungsvoll, da sie die physikalische Simulation mit künstlerischer Abstraktion verbindet. Während **Heat Distortion** und **Refraction** das Bild verformen, um eine simulierte Verformung durch Hitze und Lichtbrechung zu erzeugen, übersetzt die **Posterisierung** diesen physischen Prozess in einen stilisierten, fast surrealen Look, indem sie die Farbverläufe in harte Stufen auflöst. Eine der größten **Stärken** dieser Kombination ist die Fähigkeit, eine starke visuelle Wirkung zu erzielen und die Leistungsfähigkeit der physikalischen Shader eindrucksvoll zu demonstrieren. Sie eignet sich hervorragend für die Darstellung von Hitze, Rauch oder anderen Formen von Verzerrung. Die wesentliche **Schwäche** ist jedoch ihre thematische Spezifität, da sie primär auf Szenen mit extremer Licht- oder Hitzeeinwirkung anwendbar ist. Zudem können zu extreme Parametereinstellungen das Ergebnis unnatürlich wirken lassen. Die endgültige Wirkung kann präzise über die **Parameter** gesteuert werden, indem die Stärke der simulierten Verzerrung oder die Anzahl der Farbstufen angepasst wird.

Das Ergebnis der vierten Kombination, die in Abbildung 5.4 dargestellt ist, ist eine surrealistische Wasserlandschaft, die das Bild in eine fast flüssige, künstlerisch eingefärbte Szene mit realistischen Wellen verwandelt. Die **Synergie** dieser Effekte ist ein exzellentes Beispiel für die Verbindung von Kunst und Physik. Der **Tritone**-Shader stylisiert die gesamte Farbpalette der Landschaft und schafft so eine abstrakte, aber kohärente visuelle Grundlage. Darauf aufbauend nutzt der **Water Ripple**-Shader diese veränderte Bildinformation, um eine physikalisch korrekte, statische Simulation einer Wellenbewegung vorzunehmen. Die **Stärken** dieser Kombination sind ihre Eleganz und Einzigartigkeit, die sie zu einem vielseitigen Werkzeug für die Erzeugung verschiedener Stimmungen machen. Sie demonstriert zudem sehr eindrucksvoll die Fähigkeit des Frameworks, physikalische und künstlerische Effekte zu verbinden. Eine wesentliche **Schwäche** ist jedoch, dass die Wirkung der Effekte am besten bei Bildern mit großen, ebenen Flächen wie Gewässern zur Geltung kommt und das Ergebnis für einige Betrachter zu abstrakt sein könnte. Über die **Parameter** können die Farben des **Tritone**-Effekts und die Wellenfrequenz des **Water Ripple**-Effekts angepasst werden, um das Erscheinungsbild zu verändern.

Die individuellen Analysen zeigen, dass die vier Kombinationsbeispiele sich in zwei klare thematische Gruppen einteilen lassen. Die Kombinationen in **Abschnitt 5.1.1 (Analoger Glitch-Look)** und **Abschnitt 5.1.2 (Nostalgischer Zeitungsdruck)** verfolgen das übergeordnete Ziel, die Ästhetik und die Fehler analoger Medien zu simulieren. Der Glitch-Look ahmt die elektronischen Störungen eines alten Bildschirms nach, während der Zeitungsdruck die spezifischen Unregelmäßigkeiten eines mechanischen Druckverfahrens

repliziert. Demgegenüber stehen die Kombinationen in **Abschnitt 5.1.3 (Dramatische Verzerrung)** und **Abschnitt 5.1.4 (Surrealistische Wasserlandschaft)**, die sich auf die Anwendung von Shadern auf physikalische Phänomene konzentrieren. Hier wird die visuelle Wirkung von Hitze und Lichtbrechung sowie die Bewegung von Wasserwellen simuliert, um die Leistungsfähigkeit des Frameworks bei der Abbildung der realen Welt zu demonstrieren. Diese thematische Trennung unterstreicht die Vielseitigkeit der entwickelten Shader und zeigt ihre Eignung sowohl für künstlerische als auch für physikalische Simulationszwecke.

Neben der thematischen Einteilung lassen sich die Kombinationen auch nach der Art und Weise ihrer visuellen Wirkung und Synergie vergleichen. So nutzen die Kombinationen in **Abschnitt 5.1.1 (Analoger Glitch-Look)** und **Abschnitt 5.1.3 (Dramatische Verzerrung)** Effekte, die thematisch kohärent sind und zur Verstärkung von **Dramatik oder Chaos** dienen. Der **Noise**-Effekt im Glitch-Look verstärkt die kaputte Ästhetik des **Out-of-Register Print**-Effekts. Analog dazu nutzen die physikalischen **Heat Distortion**- und **Refraction**-Shader das Thema des Waldbrandes, um eine visuell überwältigende Verzerrung zu erzeugen, die anschließend durch den **Posterisierung**-Shader künstlerisch abstrahiert wird. Demgegenüber stehen die Kombinationen in **Abschnitt 5.1.2 (Nostalgischer Zeitungsdruck)** und **Abschnitt 5.1.4 (Surrealistische Wasserlandschaft)**, die eine andere Form der Synergie aufweisen. Sie verbinden **stilisierende Pop-Art-Effekte (Halftone und Tritone)** mit **kontrollierten, subtilen physischen Simulationen (Vignette und Water Ripple)**. Das Zusammenspiel dieser kontrastierenden Effekte schafft jeweils einen einzigartigen Stil, der zeigt, wie künstlerische Abstraktion und physikalische Realität harmonisch verschmelzen können.

### 5.3. Performance-Analyse und zeitlicher Vergleich

Nachdem die visuelle und synergetische Wirkung der Effektkombinationen in den vorherigen Abschnitten detailliert analysiert wurde, fokussiert dieser Abschnitt auf die quantitative Bewertung der technischen Performance, um die Echtzeit-Fähigkeit des Frameworks empirisch zu belegen. Die Performance-Analyse erfolgte durch die Messung der durchschnittlichen Renderzeit über **N=1000** aufeinanderfolgende Frames pro Effektkombination. Die Zeit wurde CPU-seitig in der `display()`-Methode des `GLEventListener` mittels der hochauflösenden Funktion `System.nanoTime()` erfasst und gemittelt. Dieses Vorgehen gewährleistet eine statistisch stabile und reproduzierbare Datenbasis zur Validierung der Echtzeit-Fähigkeit. Die Messungen wurden bei einer Auflösung von **1920×1080** auf einem System mit einer **AMD Radeon Graphics (Device ID: 0x164C)** durchgeführt.

Die gesammelten Messwerte, die die Effizienz der Shader-Pipeline quantifizieren, sind in der folgenden Tabelle 5.1 zusammengefasst.

Kombi.	Shader-Anzahl	Renderzeit (ms/Frame)	Framerate (FPS)
1	2	0,561	1782,07
2	2	0,333	3001,67
3	3	0,526	1900,11
4	2	0,523	1911,54

Tabelle 5.1.: Quantitative Performance-Messungen der Effektkombinationen (Mittelwert über  $N = 1000$  Frames)

Die Ergebnisse der quantitativen Analyse belegen eindrücklich die **hohe Performance und Echtzeit-Fähigkeit** des Frameworks. Mit Frameraten, die durchgängig über **1700 FPS** liegen, wird die übliche Anforderung an Echtzeitanwendungen ( $\geq 60 \text{ FPS}$ ) um ein Vielfaches übertroffen. Dies validiert den gewählten **GPU-basierten Ansatz** als hochgradig effizient.

Die **Kombination 2** (Halftone und Vignette) erweist sich mit **3001,67 FPS** als die schnellste. Ihre Effizienz resultiert aus der relativ **geringen Komplexität** der beteiligten Shader, die einfache, lokale Operationen (Punktraster, Abdunkelung) ohne komplexe Textur-Lookups durchführen.

Im Vergleich dazu ist die Renderzeit von **Kombination 1 (0,561 ms/Frame)** die längste, was auf die intensiveren Berechnungen des Out-of-Register Print-Shaders zurückzuführen ist. Die nahezu identische Leistung der **Kombinationen 3 und 4** ( $\approx 1900 \text{ FPS}$ ) zeigt ferner, dass die **Art der GLSL-Operation** (komplexe Texturkoordinaten-Verschiebungen bei Verzerrungs-Shadern) einen größeren Einfluss auf die Renderzeit hat als die reine Anzahl der hintereinandergeschalteten Effekte. Die Stabilität der hohen **FPS**-Werte bestätigt zudem die Robustheit der FBO-basierten Pipeline-Architektur.

# 6. Zusammenfassung und Ausblick

Diese Arbeit hatte zum Ziel, ein interaktives Framework zur modularen Echtzeit Bildverarbeitung zu entwickeln, das künstlerisch inspirierte Pop-Art-Effekte mit physikalisch motivierten Shader-Effekten kombiniert. Aufbauend auf den theoretischen Grundlagen der Echtzeitgrafik und unter Verwendung von Java, JOGL und der Shader-Sprache GLSL wurde ein System implementiert, das die flexible Anwendung und Kombination dieser Effekte ermöglicht. Die im Rahmen der Arbeit entwickelten Shader-Effekte wurden in vier demonstrativen Kombinationen evaluiert, die erfolgreich die Vielseitigkeit des Frameworks unter Beweis stellten. Durch eine detaillierte Analyse und einen direkten Vergleich wurde gezeigt, dass die erstellten Effekte sich in zwei thematische Gruppen einteilen lassen: die **Simulation analoger Medien und Ästhetiken** sowie die **Anwendung auf physikalische Phänomene**. Die quantitative Performance-Analyse bestätigte die hohe Effizienz des gewählten GPU-basierten Ansatzes: Alle Effektkombinationen übertrafen die Anforderung an die Echtzeitverarbeitung ( $>= 60 \text{ FPS}$ ) um ein Vielfaches und erreichten Spitzenwerte von über **3000 FPS**. Das Framework hat somit sein Potenzial als flexibles Werkzeug für eine Vielzahl kreativer und technischer Anwendungen unterstrichen.

Trotz der erfolgreichen Umsetzung sind die Limitationen der Arbeit zu beachten. Eine wesentliche Einschränkung liegt in der primär **statischen Natur** der implementierten Effekte, da die dynamische Komponente des Frameworks nicht Teil des Projektumfangs war. Dies schränkt die Darstellung physikalischer Phänomene, die per se dynamisch sind, auf eine einzelne Momentaufnahme ein. Zudem fokussiert sich die Arbeit ausschließlich auf die **2D-Bildverarbeitung** und die serielle Anwendung von Shadern in einer linearen Pipeline. Das System ist zwar funktional, könnte in einem größeren Projekt jedoch durch eine optimierte Benutzeroberfläche und erweiterte Interaktionsmöglichkeiten nutzerfreundlicher gestaltet werden.

Die erkannten Limitationen bieten zugleich eine klare Grundlage für zukünftige Entwicklungen. Ein wesentlicher Ausblick ist die Implementierung einer **dynamischen Komponente**, um die Effekte zu animieren und so eine lebendige Darstellung von Phänomenen wie Wasserwellen oder Hitzeblitzen zu ermöglichen. Eine weitere Erweiterung könnte in der Entwicklung **komplexerer, physikalischer Shader** liegen, die etwa die Simulation von Wasserströmungen oder die Darstellung von Lichtstreuung im Nebel umfassen. Auch die Schaffung eines **nutzerfreundlichen Interfaces**, möglicherweise in Form eines visuellen Editors zur freien Anordnung der Shader in komplexen Pipelines, würde die Interaktivität und den kreativen Spielraum für zukünftige Anwender deutlich erhöhen.

# Literaturverzeichnis

- [1] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 4th ed. Boca Raton: CRC Press, 2018.
- [2] T.-W. Schmidt, F. Pellacini, D. Nowrouzezahrai, W. Jarosz, and C. Dachsbaecher, “State of the Art in Artistic Editing of Appearance, Lighting, and Material,” in *Eurographics 2014 - State of the Art Reports*. Strasbourg, France: Eurographics Association, 2014.
- [3] D. Shreiner, G. Sellers, J. M. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*, 8th ed. Boston: Addison-Wesley, 2013.
- [4] R. J. Rost and B. Licea-Kane, *OpenGL Shading Language*, 3rd ed. Boston: Addison-Wesley, 2009.
- [5] D. Wolff, *OpenGL 4.0 Shading Language Cookbook*. Birmingham: Packt Publishing, 2011.
- [6] L. Alloway, *American Pop Art*. New York: Macmillan, 1974.
- [7] W. F. Engel, Ed., *GPU Pro 7: Advanced Rendering Techniques*. Boca Raton: CRC Press, 2016.
- [8] M. A. E.-M. A. El-Majed, A. E.-R. A. E.-H. El-Gendy, and G. G. Fawze, “The effect of the artistic features of the pop art’s works on the animation films,” *International Journal of Social Science and Humanities Research*, vol. 8, no. 3, pp. 229–237, 2020.
- [9] C. Yang, “Analysis on the commercialization of pop art,” in *Proceedings of the 2023 5th International Conference on Literature, Art and Human Development (ICLAHD 2023)*. Atlantis Press, 2023, pp. 725–730.
- [10] R. I. Tosi, “Pop culture art as educational bridge: Connecting generations through games and animation movies in classroom environment,” *Open Journal of Social Sciences*, vol. 11, no. 5, pp. 18–32, 2023.
- [11] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y.-C. Kuo, A. Kaufman, and K. Mueller, “Visual simulation of heat shimmering and mirage,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 1, pp. 179–189, 2007.
- [12] S. Premoze and M. Ashikmin, “Rendering Natural Waters,” *Computer Graphics Forum*, vol. 20, no. 4, pp. 189–199, 2001.
- [13] D. Blanchette and E. Agu, “Adaptive spectral mapping for real-time dispersive refraction,” in *Advances in Visual Computing (ISVC 2012)*. Springer, 2012, pp. 81–91.

- [14] M. Makarenko, “Towards Real-Time 3D VR Simulation of Stained Glass Windows,” Master’s Thesis, Trinity College Dublin, Dublin, Ireland, 2024.
- [15] K. Kallio, “Scanline edge-flag algorithm for antialiasing,” in *Theory and Practice of Computer Graphics*. Bangor, Wales: Eurographics Association, 2007.
- [16] J. C. Hart, “Perlin noise pixel shaders,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. Los Angeles, USA: Association for Computing Machinery, 2001.
- [17] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2002.
- [18] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2005.
- [19] G. Sellers, R. S. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. Boston, MA: Addison-Wesley, 2015.
- [20] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole, *Java Swing*, 2nd ed. Sebastopol, CA: O’Reilly Media, 2002.
- [21] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Professional, 1999.
- [22] L. R. Lippard, *Pop Art*. New York: Praeger Publishers, 1966.
- [23] G. Sharma, Ed., *Digital Color Imaging Handbook*. Boca Raton, FL: CRC Press, 2003.
- [24] R. A. Ulichney, *Digital Halftoning*. Cambridge, MA: MIT Press, 1987.
- [25] B. Gooch and A. Gooch, *Non-Photorealistic Rendering*. Wellesley, MA, USA: A K Peters Ltd., 2001.
- [26] P. G. Vivo and J. Lowe. (2015) The book of shaders. [Online]. Available: <https://thebookofshaders.com/>

# Abkürzungsverzeichnis

**2D** - Zwei-Dimensional

**3D** - Drei-Dimensional

**N** - Anzahl der Frames

**API** - Application Programming Interface

**CPU** - Central Processing Unit

**CRT** - Cathode Ray Tube

**FBO** - Framebuffer Object

**FPS** - Frames per Second

**GLSL** - OpenGL Shading Language

**GPU** - Graphics Processing Unit

**GUI** - Graphical User Interface

**JOGL** - Java OpenGL

**JPG** - Joint Photographic Experts Group

**JUnit** - Ein Test-Framework für Java

**MVC** - Model-View-Controller

**OpenGL** - Open Graphics Library

**RGB** - Rot, Grün, Blau

**UV** - Texturkoordinaten

**ms/Frame** - Millisekunden pro Frame

# Abbildungsverzeichnis

2.1.	Übersicht der OpenGL-Rendering-Pipeline. Quelle: [3] . . . . .	4
2.2.	Darstellung der Shader Manipulation. Quelle: Eigene Darstellung . . . . .	6
2.3.	Workflow des Render-to-Texture-Verfahrens mit FBO. Quelle : in Anlehnung an [5] . . . . .	6
2.4.	Typische Stilmerkmale in der Pop-Art. Quelle : [6] . . . . .	8
3.1.	Übersicht der MVC-Architektur. Quelle: [17] . . . . .	14
3.2.	Projektstruktur. Quelle: Eigene Darstellung . . . . .	15
3.3.	Workflow zum Laden von Texturen. Quelle : Eigene Darstellung . . . . .	16
3.4.	Workflow zum Laden von Shadern. Quelle : Eigene Darstellung . . . . .	17
3.5.	Rendering-Pipeline: Aufbau und Datenfluss. Quelle : Eigene Darstellung . .	17
3.6.	GUI. Quelle: Eigene Darstellung . . . . .	19
3.7.	Noise Dialog. Quelle: Eigene Darstellung . . . . .	19
5.1.	Gegenüberstellung des Originalbildes und des analogen Glitch-Looks durch die Kombination der Shader Out-of-Register Print und Noise. . . . .	30
5.2.	Gegenüberstellung des Originalbildes und des nostalgischen Zeitungsdrucks durch die Kombination der Shader Halftone und Vignette. . . . .	30
5.3.	Gegenüberstellung des Originalbildes und der dramatischen Verzerrung durch die Kombination der Shader Heat Distortion, Refraction und Posterisierung. . . . .	31
5.4.	Gegenüberstellung des Originalbildes und der surrealistischen Wasserschaft durch die Kombination der Shader Tritone und Water Ripple. . . . .	32
A.1.	Der Posterisierungs-Effekt reduziert die Anzahl der Farbstufen im Bild. . . . .	43
A.2.	Der Duotone-Effekt ordnet Graustufen auf eine zweifarbige Palette ab. . . . .	43
A.3.	Der Tritone-Effekt erweitert den Duotone, indem er drei Farbtöne für Schatten, Mitteltöne und Highlights verwendet. . . . .	44
A.4.	Der Selective Color Boost-Effekt verstärkt gezielt die Sättigung bestimmter Farbbereiche. . . . .	44
A.5.	Der Halftone-Effekt simuliert Druckrasterpunkte, deren Dichte von der Helligkeit des Bildes abhängt. . . . .	45
A.6.	Der Benday Dots-Effekt simuliert farbige Rasterpunkte, deren Größe abhängig von der Helligkeit des Bildes ist, um Flächen und Schatten darzustellen. . . . .	45
A.7.	Der Out-of-Register Print-Effekt imitiert einen fehlerhaften Druck, indem die Farbkanäle leicht gegeneinander verschoben werden. . . . .	46
A.8.	Der Comic Look-Effekt kombiniert Posterisierung mit einer Kantenerkennung, um das Bild wie eine Illustration wirken zu lassen. . . . .	46
A.9.	Der Intelligent Bold Outlines-Effekt erkennt Kanten im Bild und erzeugt dynamisch dicke Umrisslinien. . . . .	47

A.10.Der Seriality-Effekt repliziert das Bild in einer Kachelstruktur und imitiert so die serielle Ästhetik der Pop-Art. . . . .	47
A.11.Der Bloom-Effekt simuliert das Überstrahlen heller Lichtquellen. . . . .	48
A.12.Der Chromatic Aberration-Effekt imitiert einen Linsenfehler, bei dem Farbsäume an den Rändern von Objekten auftreten. . . . .	48
A.13.Der Distortion-Effekt verzerrt die Bilddarstellung. . . . .	49
A.14.Der Chromatic Wave Distortion-Effekt kombiniert Wellenverzerrung mit einer Separation der Farbkanäle. . . . .	49
A.15.Der Heat Distortion-Effekt simuliert die visuelle Verzerrung über heißen Oberflächen. . . . .	50
A.16.Der Water Ripple-Effekt erzeugt das visuelle Muster von Wellen auf einer Wasseroberfläche. . . . .	50
A.17.Der Refraction-Effekt simuliert die Lichtbrechung, indem er das Bild mit einer einfachen Wellenverzerrung abbildet. . . . .	51
A.18.Der Vignette-Effekt verdunkelt die Ränder des Bildes, um den Fokus auf die Mitte zu lenken. . . . .	51
A.19.Der Noise-Effekt fügt dem Bild zufälliges Rauschen hinzu, um eine analoge oder Retro-Ästhetik zu erzeugen. . . . .	52
A.20.Der Scanline-Effekt imitiert die horizontalen Zeilenraster von alten Röhrenfernsehern. . . . .	52

# Tabellenverzeichnis

2.1. Kategorien physikalischer Effekte in der Computergrafik . . . . .	10
5.1. Quantitative Performance-Messungen der Effektkombinationen (Mittelwert über $N = 1000$ Frames) . . . . .	34

# A. Anhang

## Galerie der Pop-Art-Effekte



Abbildung A.1.: Der Posterisierungs-Effekt reduziert die Anzahl der Farbstufen im Bild.



Abbildung A.2.: Der Duotone-Effekt ordnet Graustufen auf eine zweifarbig Palette ab.



Abbildung A.3.: Der Tritone-Effekt erweitert den Duotone, indem er drei Farbtöne für Schatten, Mitteltöne und Highlights verwendet.



Abbildung A.4.: Der Selective Color Boost-Effekt verstärkt gezielt die Sättigung bestimmter Farbbereiche.



Abbildung A.5.: Der Halftone-Effekt simuliert Druckrasterpunkte, deren Dichte von der Helligkeit des Bildes abhängt.

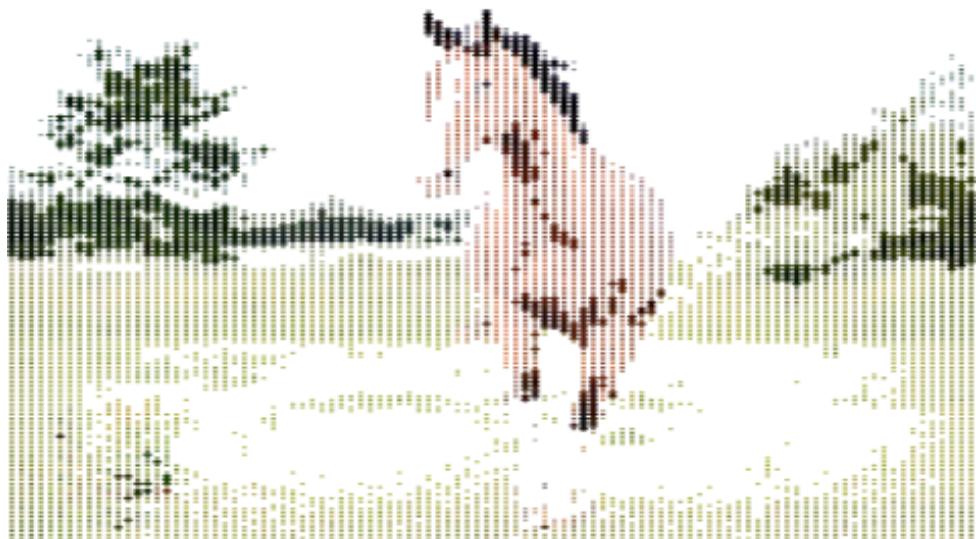


Abbildung A.6.: Der Benday Dots-Effekt simuliert farbige Rasterpunkte, deren Größe abhängig von der Helligkeit des Bildes ist, um Flächen und Schatten darzustellen.



Abbildung A.7.: Der Out-of-Register Print-Effekt imitiert einen fehlerhaften Druck, indem die Farbkanäle leicht gegeneinander verschoben werden.



Abbildung A.8.: Der Comic Look-Effekt kombiniert Posterisierung mit einer Kantenerkennung, um das Bild wie eine Illustration wirken zu lassen.



Abbildung A.9.: Der Intelligent Bold Outlines-Effekt erkennt Kanten im Bild und erzeugt dynamisch dicke Umrisslinien.

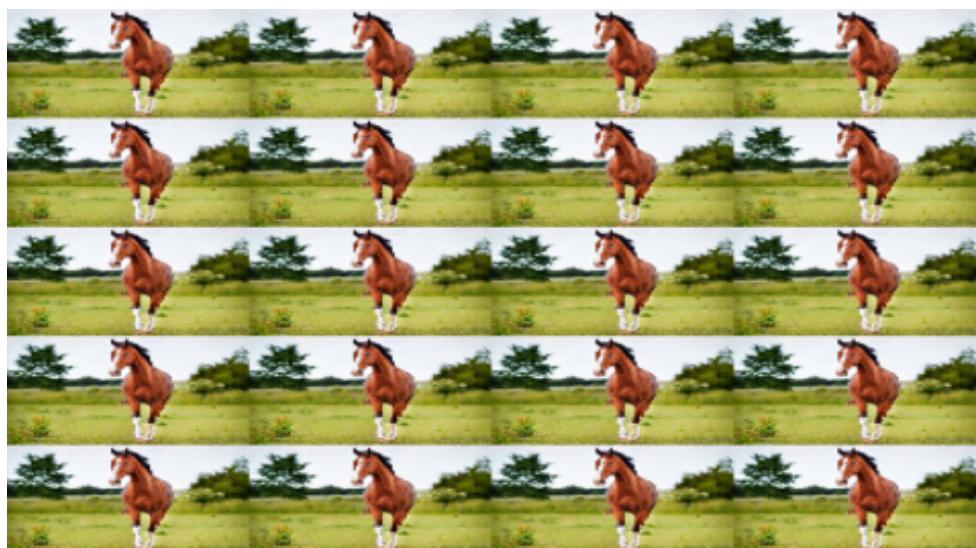


Abbildung A.10.: Der Seriality-Effekt repliziert das Bild in einer Kachelstruktur und imitiert so die serielle Ästhetik der Pop-Art.

## Galerie der Physikalischen Effekte



Abbildung A.11.: Der Bloom-Effekt simuliert das Überstrahlen heller Lichtquellen.



Abbildung A.12.: Der Chromatic Aberration-Effekt imitiert einen Linsenfehler, bei dem Farbsäume an den Rändern von Objekten auftreten.



Abbildung A.13.: Der Distortion-Effekt verzerrt die Bilddarstellung.



Abbildung A.14.: Der Chromatic Wave Distortion-Effekt kombiniert Wellenverzerrung mit einer Separation der Farbkanäle.



Abbildung A.15.: Der Heat Distortion-Effekt simuliert die visuelle Verzerrung über heißen Oberflächen.



Abbildung A.16.: Der Water Ripple-Effekt erzeugt das visuelle Muster von Wellen auf einer Wasseroberfläche.



Abbildung A.17.: Der Refraction-Effekt simuliert die Lichtbrechung, indem er das Bild mit einer einfachen Wellenverzerrung abbildet.



Abbildung A.18.: Der Vignette-Effekt verdunkelt die Ränder des Bildes, um den Fokus auf die Mitte zu lenken.



Abbildung A.19.: Der Noise-Effekt fügt dem Bild zufälliges Rauschen hinzu, um eine analoge oder Retro-Ästhetik zu erzeugen.



Abbildung A.20.: Der Scanline-Effekt imitiert die horizontalen Zeilenraster von alten Röhrenfernsehern.

# Danksagung

Hiermit möchte ich mich besonders bei Prof. Dr. Thormählen für die Betreuung meiner Arbeit, hilfreiche Diskussionen und viel Geduld bei zahlreichen Fragen bedanken.