Darstellung von Motion Blur durch Non-Photorealistic Rendering Verfahren

Diplomarbeit

zur Erlangung des Grades Diplominformatiker
im Studiengang Computervisualistik

vorgelegt von

Eckhard Großmann

Erstgutachter:	Prof. Dr. Stefan Müller Institut für Computervisualistik, Fachbereich 4
Zweitgutachter:	Dipl. Inf. Martin Schumann Institut für Computervisualistik, Fachbereich 4

Koblenz, im 7. Januar 2011

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Diplomarbeit
Eckhard Großmann
(Matr.-Nr. 205 210 121)

Thema: **Darstellung von Motion Blur durch Non-Photorealistic Rendering Verfahren**

Non-Photorealistic Rendering (NPR) abstrahiert Objekte und Szenen. Das Teilgebiet der Computergraphik wird bei Spielen, Filmen und Computer Aided Design (CAD) verwendet. Kanten von Objekten werden betont und Farben reduziert. Die Abstraktion kann soweit gehen, dass anstatt der exakten Silhouetten vereinfachte Umrisse gezeichnet werden. Diese Umrisse können auch durch Stilisierung mittels starker oder schwacher Striche gezeichnet werden. Statt flächigen Farben werden schwache oder starke Schraffuren dargestellt, eventuell sind diese farbig. Objekte, die mittels NPR gerendert sind, können mit Comic-Zeichnungen verglichen werden. Bei genauerer Betrachtung solcher Zeichnungen fällt auf, dass sie Bewegungen andeuten. Bewegungen von schnellen Objekten werden beispielsweise durch Linien, so genannte Speed Lines, dargestellt. Schnelle Objekte können aber deformiert sein. Vibrationen von Objekten können durch mehrere, zueinander verschobenen, vereinfachten Silhouetten angedeutet werden. Bei existierenden Verfahren wird die Bewegung nicht durch Bewegungsunschärfe dargestellt, sondern nur mittels zusätzlicher Linien angedeutet.

Ziel dieser Arbeit ist die Darstellung von Bewegungsunschärfe mittels Techniken aus dem Teilgebiet NPR. Hierzu werden bestehende Darstellungen von Bewegungen in Comic-Illustrationen ausfindig gemacht und in verschiedene Kategorien unterteilt. Vorhandene Verfahren zur Andeutung von Bewegung bzw. zur Darstellung von Bewegungsunschärfe werden recherchiert und miteinander verglichen. Der Hauptaspekt ist es ein neues, eigenes Verfahren zu entwickeln, welches auf vorhandenen Motion Blur-Verfahren basiert. Die Bewegungsunschärfe soll zusätzlich die Farbe der Objekte andeuten.

Schwerpunkte dieser Arbeit sind:

1. Recherche und Klassifizierung von Bewegungsarten im Bereich von Comic-Darstellungen
2. Recherche und Untersuchung vorhandener Verfahren zur Darstellung von Bewegung bzw. Bewegungsunschärfe
3. Eigenständige Entwicklung eines Verfahrens zur Darstellung von Bewegung, basierend auf bestehenden Verfahren wie z.B. Motion Blur
4. Dokumentation

Koblenz, den 19.05.2010

Prof. Dr. Stefan Müller

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

☐ Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

☐ Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den . . . . . .

Danksagung

Ich möchte als erstes Erik Knauer danken. Er hat mich auf dieses Diplomarbeitsthema gebracht und hat mir sehr geholfen bei der einigen Lösungsfindung.

Ich möchte Prof. Dr. Stefan Müller und Martin Schumann dafür danken, dass sie mich betreut haben, so diese Arbeit ermöglicht haben und immer zur Verfügung standen um über neue Lösungenswege zu diskutieren.

Mein größter Dank geht an Elisa Ziegler, die mich unermüdlich auf Schwächen in meinen Texten hingewiesen hat. Auch längere Nächte haben sie nicht davor zurückgeschreckt meine Arbeit zu korrigieren. Außerdem stand sie mir immer mit guten Rat zur Seite wenn ich die große Struktur der Textes vor lauter Buchstaben nicht mehr gesehen habe.

Einen besonderen Dank geht auch an Prof. Dr. Grünewald und Dr. Markus Lohoff. Sie haben mir einige Impulse für die Betrachtung aus Sicht der Kunst gegeben. Außerdem gaben Sie mir immer gute Ratschläge und einen unheimlichen Motivationsschub.

Ich möchte hier vor allem auch meiner Familie danken, die mich beim Schreiben der Arbeit unterstützt haben und hin und wieder meine Arbeit gelesen haben um Verständnisfehler zu entdecken. Besonders mein Vater und Jörn haben sich hierbei viel Mühe gegeben.

Ich danke Julia Metzdorf. Sie hat sich zur gleichen Zeit mit dem verwandten Themengebiete Comics beschäftigt und konnte mich so auf einige interessante Themen hinweisen. Auf dem gleichen Themengebiet hat sich mein Freund Patrick Bardow ebenfalls als unschätzbare Informationsquelle erwiesen.

Ich möchte hier nochmal allen Freunden danken, die Opfer meiner Sprachschwäche geworden sind. Die Korrekteure waren:
Elisa Ziegler, Eberhard Großmann, Jörn Großmann, Christina Schmidt, Susanne Maur und Mahdi Darekhshanmanesh.

Inhaltsverzeichnis

1. Einleitung

Bei Betrachtung der verschiedenen Strömungen in der Kunst fällt auf, dass Künstler immer wieder neue Ausdrucksformen suchen und entwickeln. So entstanden seit dem 19. Jahrhundert eine Vielfalt an Darstellungen, die die Realität nicht mehr nachahmten sondern sich von ihr abwandten. Vom Impressionismus, über den Kubismus, den Futurismus bis hin zu Pop-Art und abstrakter Kunst existieren so vollkommen unterschiedliche Richtungen, die alle unzählige Möglichkeiten der nicht realitätsgetreuen Darstellung bieten.

Auch in der Computergraphik soll eine solche Vielfalt der Darstellung erreicht werden. Verantwortlich dafür sind aber nicht nur Designer, sondern auch Programmierer. Ihre verwendeten Gestaltungskonzepte können entscheidend dazu beitragen, möglichst viele Arten der Visualisierung zu ermöglichen. Entsprechend des Kontexts und der zu transportierenden Informationen kann dann stets die passende Darstellungsform gewählt werden.

Mit dem Bereich nicht realitätsgetreuer Darstellung beschäftigt sich das *Non-Photorealistic Rendering (NPR)*, welches im Gegensatz zum photorealistischen Rendern steht. Das photorealistische Rendern strebt möglichst realitätsgetreue Darstellungen an. NPR hingegen lässt die physikalischen Gesetze außer Acht. So verwendet es vereinfachte Darstellungen, reduziert die Farben, schraffiert die darzustellenden Objekte, hebt Kanten und Objektumrisse durch Silhouetten hervor oder deformiert Objekte, die in Bewegung sind. NPR setzt alle nur denkbaren Darstellungsformen um. Dabei gelten die genannten Stilrichtungen der Kunst als Inspiration.

Diese Arbeit will Geschwindigkeit visuell mit Mitteln des NPRs darstellen. Sie stellt Bewegung nicht-photorealistisch dar, z.B. auf Basis der Bewegungsdarstellungen in Comics. In Comics wird Geschwindigkeit von Gegenständen und Personen durch *Speed Lines (Bewegungslinien)* vermittelt. Der sich bewegende Gegenstand bzw. die sich bewegende Person ist während der Bewegung grundsätzlich erkennbar [Cut02]. Hingegen können bewegte Objekte bei realitätsgetreuer Darstellung (z.B. in der Photographie) nicht mehr genau wahrgenommen werden. Sie verschwimmen und Details gehen unter, es entsteht *Motion Blur (Bewegungsunschärfe*). Motion Blur ist ein Effekt der Aufnahme und beschränkt sich auf den Bewegungsradius. Vor allem die photorealistische Computergraphik greift auf diesen Effekt zurück um den Eindruck von Bewegung zu vermitteln. Das Konzept von Speed Lines liegt in starren, gezeichneten Bildern begründet [Sco94] und ist Teilgebiet des NPRs. Speed Lines sollen die Bewegung andeuten und gleichzeitig die wesentlichen Informationen hervorheben. Es ist durchaus möglich, dass Speed Lines Bewegungen übermäßig betonen, indem sie weit über den Bewegungsradius hinaus reichen.

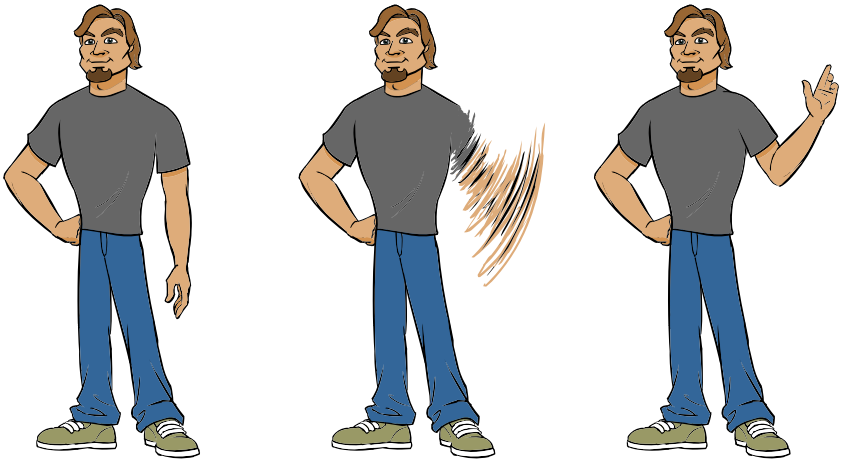

Abbildung 1.1.1.: Annäherung einer Armbewegung durch Speed Lines. [Sim]

Diese Arbeit will neue Möglichkeiten zur Bewegungsdarstellung entwickeln, indem sie sowohl Speed Lines als auch Motion Blur als Grundlage nimmt. Auf ihnen aufbauend entwickelt sie ein neuartiges Verfahren zur Bewegungsdarstellung, welches die Darstellungsformen der Computergraphik erweitert und so zu einer größeren graphischen Vielfalt beiträgt.

1.1. Motivation

Die erwähnten Konzepte zur Darstellung von Bewegung in der Computergraphik lassen sich grundsätzlich unterscheiden in: Speed Lines bei NPR und Motion-Blur im Bereich des Photorealismus. NPR hat das Konzept der Speed Lines von gezeichneten Bildern übernommen und auf animierte Bilder übertragen. Bisher sind für solche Animationen vorgefertigte Datensätze und das Arbeiten auf der Geometrie an sich erforderlich, vgl. [MSS99, Han04]. Dabei bleibt das Objekt bei jeder Bewegung klar zu erkennen. Es gibt kein Verfahren, dass eine Kombination aus Motion-Blur und NPR-Techniken bildet. Das Phänomen der Unschärfe durch das Objekte und Personen mit zunehmender Geschwindigkeit kaum mehr erkennbar sind, könnte die Bewegungsdarstellungen jedoch bereichern und neue Möglichkeiten aufzeigen.

Simmons [Sim] schlägt vor, beim Animieren einer schnellen Handbewegung die Hand bzw. den ganzen Arm durch Speed Lines zu ersetzen. Dieser Effekt ist in Abbildung 1.1.1 illustriert - der Arm verschwindet hier bei einer schnellen Bewegung, nur noch Speed Lines in entsprechenden Farben sind sichtbar. Dies erinnert an Motion-Blur. In [Sim] wird aber kein Verfahren für die Computergraphik beschrieben, sondern eine Vorgehensweise für den künstlerischen Gebrauch bei der Animation mit Flash oder ähnlichen Programmen. Zur Zeit gibt es kein Verfahren in der Computergraphik, das diesen Effekt nachbildet oder einen ähnlichen Ansatz verfolgt.

Diese Arbeit greift den Ansatz von Simmons auf und führt damit den Unschärfeeffekt von Motion-Blur in den Bereich des NPRs ein. Anstatt allerdings nur Unschärfe bei bewegten Ob-

jekten zu erzeugen, stützt sich diese Arbeit zusätzlich auf den vereinfachenden Gedanken des NPRs. Sie verwendet Speed Lines und Bewegungsunschärfe gemeinsam, um so das Repertoire der Bewegungsdarstellung im NPR zu bereichern.

Damit betritt diese Arbeit auf zwei Gebieten Neuland:

- Erstens wird das Konzept von *Motion-Blur* ins NPR übertragen und
- Zweitens werden Speed Lines in *Echtzeit* auf *beliebige Geometrie* angewandt.

1.2. Zielsetzung

Diese Arbeit hat die Entwicklung eines Verfahrens zum Ziel, dass Bewegung und auftretende Bewegungsunschärfe durch Verfahren des Non-Photorealistic Renderings darstellt. Dabei soll das angestrebte Verfahren den Verlauf der Bewegung für ein dargestelltes Bild ermitteln und mit Speed Lines annähern. Das sich bewegende Objekt bzw. der sich bewegende Teil des Objekts wird abhängig von der Stärke der Bewegung teilweise bis komplett ausgeblendet. Das Verfahren soll echtzeitfähig sein.

Aufgrund von vorgestellten Verfahren im Bereich NPR und Motion Blur entwickelt die Arbeit eigene Ansätze, die die genannten Anforderungen umsetzen. Dabei beachtet die Arbeit zwei Aspekte: Sie nimmt so wenige Änderungen wie möglich am verwendeten Szenegraphen vor und führt nach Möglichkeit zur Laufzeit keine Berechnungen auf Seiten der CPU durch. So soll ein Verfahren entstehen, das als Post-Processing Verfahren in Anwendungen integriert werden kann, interaktive Wiederholungsraten ermöglicht und damit auch in Spielen zur Anwendung kommen kann.

Aus diesen Gründen zieht die Arbeit bestehende Verfahren, die auf Shadern aufbauen, heran, die sie verwendet und ggf. weiterentwickelt. Jegliche Geometrie wird nur im Vertex- und Geometry-Shader verändert, wobei der Geometry-Shader zusätzlich Geometrie erzeugen kann. Diese Arbeit entwirft eine Implementation des entwickelten Konzepts, dem Informationen über die Szene nur in Form von G-Buffern[1] zur Verfügung stehen, weswegen das Verfahren die Szene nicht so oft rendern muss. Die Informationen, die ein Bild eines G-Buffers wiedergibt, können z.B. Geschwindigkeitsvektoren, Positionen, Farben oder Normalen sein, die bei der Berechnung helfen. In interaktiven Anwendungen und Spielen liegt kein Wissen über zukünftige Ereignisse vor, weshalb die hier angestrebte Anwendung auf solche Informationen verzichten muss.

[1] G-Buffer sind gerenderte Bilder mit Informationen über den vordersten Oberflächenpunkt, beispielsweise den Normalen.

1.3. Abgrenzung

Diese Arbeit nähert sich Motion Blur nicht, wie in [Gre03, SSC03], photorealistisch an. Das entwickelte Verfahren arbeitet im Gegensatz zu ähnlichen Verfahren wie [MSS99, Han04, Son05, SSBG10] nicht auf Basis der Geometrie sondern nur mit Informationen, die in vorab erstellten, sogenannten G-Buffern zur Verfügung stehen. Das Verfahren verwendet keine Animationsdatensätze und arbeitet darum nur auf schon bekannten Informationen.

Diese Arbeit verwendet wie in Haller et al. [HS04] eine Art Partikel-System und Informationen aus der vorgerenderten Szene um Striche bzw. Speed Lines zu erzeugen. Sie erzeugen aber nur aus gegebenen Bildern stilisierte Bilder, z.B. ein Wasserfarbeneffekt. Im Gegensatz dazu benutzt diese Arbeit jedoch weitere G-Buffer um stattdessen Speed Lines zu generieren und verwendet ein bildbasiertes Partikelsystem anstatt eines objektbasierten.

1.4. Herangehensweise

Um dem Ziel dieser Arbeit gerecht zu werden, ist ein umfangreiches Hintergrundwissen erforderlich. In der ersten Phase dieser Arbeit fand deshalb eine Literaturrecherche statt, die sich Kunst und Bewegung im Allgemeinen und Photographie sowie Comics im Speziellen vornahm. Werke, die sich speziell mit der Kategorisierung von Bewegungsdarstellungen beschäftigen, leisteten hierzu einen Beitrag. Die Literaturrecherche setzte sich außerdem mit existierenden Verfahren zur Illustration von Bewegung im Kontext des NPRs und des Photorealismus auseinander. Die mögliche Realisierung und der dargestellte Bewegungsstil dieser Verfahren bildeten die Basis für eine umfassende Analyse ihrer Stärken und Schwächen.

Eigene Ansätze, die der Zielsetzung dieser Arbeit gerecht werden, entstanden auf Grundlage der gefundenen Verfahren und ihrer Bewertung. Daraufhin fand die Umsetzung und Analyse der entwickelten Verfahren statt, die sich während dieses Prozesses wo nötig veränderten und verfeinerten.

Diese Arbeit dokumentiert die genannten Arbeitsschritte und ihre Ergebnisse. Sie bewertet die Ergebnisse und beurteilt sie anhand der praktischen Umsetzung.

1.5. Ergebnis

Diese Arbeit verfolgt zwei Ansätze. Der erste Ansatz kombiniert eine Tonal Art Map [PHWF01], gestreckte Geometrie [Gre03] und einen Velocity-Buffer [SSC03]. Dabei sollen die Bereiche eines Objekts entsprechend ihrer Deckkraft mit vorgerenderten Schraffurtexturen verschiedener Stärken angenähert werden. Das Vorgehen weist aus mehreren Gründen Schwächen auf.

Der zweite Ansatz berücksichtigt die Erkenntnisse, erzeugt ein Strichpolygon und zeichnet es in die Szene hinein. Hierzu nimmt er die Geschwindigkeit des betrachteten Oberflächenpunkts zu Hilfe. Er verwendet zufällige Punkte, so genannte Seed Points, als Startpunkte für die Striche. Mit den für jeden Blickpunkt ermittelten Informationen wie Geschwindigkeit und Position, legt er Striche dreidimensional in die Szene.

Diese Arbeit präsentiert das aus diesen Entwicklungen entstandene Post-Processing Verfahren, das es ermöglicht Motion Blur mit Strichen anzunähern. Das Verfahren kann auch sehr leicht nur Speed Lines zeichnen und verwendet dafür tiefenunabhängig Strichdicke.

1.6. Übersicht

Zunächst fasst Kapitel 2 auf Seite 7 mehrere Beispiele für Darstellungsvarianten von Bewegung in der Kunst zusammen. Behandelt werden Photographien, Gemälde und Comics. Besonders in Comics sind viele Stile zu erkennen, die die Verfahren des NPRs nachzuahmen versuchen. Abschließend werden verschiedene Bewegungsstile anhand ihrer wahrnehmungsspezifischen Aspekte eingeteilt. Kapitel 3 auf Seite 19 führt dann wesentliche NPR-Verfahren ein. Es stellt Verfahren für Farbreduktion, Gooch-Shading, Hatching, Silhouetten, Deformation und Speed Lines vor. Es stellt alle Verfahren, die sich insbesondere mit Bewegung beschäftigen, einander gegenüber. Für ein besseres Verständnis geht Kapitel 4 auf Seite 39 genauer auf Motion Blur ein. Das Kapitel verdeutlicht die Funktionsweise einer Kamera, um die physikalische Entstehung von Motion Blur zu erklären. Danach bespricht es verschiedene Implementationsvarianten und beleuchtet ihre Vor- und Nachteile.

Nachdem alle entscheidenden Grundlagen behandelt wurden, stellt Kapitel 5 auf Seite 51 den ersten Ansatz für ein Verfahren vor. Schritt für Schritt erklärt das Kapitel das Vorgehen und geht auf sich aus ihnen ergebende Probleme ein. Kapitel 6 stellt einen weiteren Ansatz vor, der die zuvor gewonnenen Erkenntnisse berücksichtigt. Das darauffolgende Kapitel 7 beschreibt die Implementation des endgültigen Verfahrens mit allen eingebrachten Korrekturen und beschreibt weiterführende Techniken. Zum Abschluss vergleicht Kapitel 8 das entwickelte mit existierenden Verfahren und erläutert Vor- und Nachteile.

Im Anhang beschäftigt sich Anhang A mit wichtigen Punkten von OpenGL 3.0 und GLSL 1.5. Außerdem definiert der Anhang eine Reihe von Eingabe- und Ausgabevariablen sowie uniforme Variablen, die die verwendeten Codebeispielen als gegeben voraussetzen.

2. Darstellung von Bewegung

Diese Arbeit beschäftigt sich mit der Darstellung von Bewegung. Dieses Kapitel beleuchtet, wie ein Bild einen Eindruck von Bewegung wiedergibt und dem Betrachter Bewegung suggerieren kann. Hierzu betrachtet das Kapitel die Kunstgeschichte und geht speziell auf die Entwicklung von Photographie und Comics ein.

Zuerst betrachtet Abschnitt 2.1 die Geschichte der Photographie und ihren Einfluss auf andere Strömungen der Kunst. Dabei geht es auf wichtige Künstler, die sich mit der Chronophotographie beschäftigt haben, und deren Werke ein, sowie auf einige photographische Werke, die Bewegung unterschiedlich repräsentieren. Im weiteren Verlauf behandelt Abschnitt 2.2 Comics. Das abschließende Kapitel stellt eine Kategorisierung der verschiedenen Ausdruckmittel von Bewegung vor.

2.1. Kunst und Bewegung

Die Photographie hatte einen großen Einfluss auf die Weiterentwicklung der Kunst im 19. Jahrhundert. Sie inspirierte viele Künstler und thematisierte auch die Bewegung, die in der Kunst vor ihrem Aufkommen eine untergeordnete Rolle einnahm.

2.1.1. Entwicklung der Photographie

Die Erfindung der Photographie 1839 war eine Sensation, die Zeitgenossen als Medium priesen, um Aufnahmen der Welt naturalistisch anzufertigen [SS90, S. 15-21]. In den Anfängen der Photographie waren Aufnahmen von Personen durch die langen Belichtungszeiten schier unmöglich [SS90, S. 56-62]. Das Photomaterial war noch nicht so empfindlich wie heutzutage. In Deutschland beanspruchte eine Aufnahme meist zwischen acht und zehn Minuten, am Äquator zwei bis drei. Abhängig von den Lichtverhältnissen konnte die Belichtungszeit sogar bis zu 30 Minuten betragen [SS90, S. 15]. Aus diesem Grund wurden anfangs nur starre Objekte photographiert. Es verwundert also nicht, dass zwei Aufnahmen des belebten Straßenzugs Boulevard du Temple von Daguerre aufgenommen 1838 weder Menschen noch Wagen zeigen. Auf einem der beiden Photos ist dennoch der Umriss eines Menschen zu sehen, der lange in derselben Position verharrte.

Bereits 1844 behauptete Daguerre die ersten Momentaufnahmen von galoppierenden Pferden und fliegenden Vögeln angefertigt zu haben, was allerdings bezweifelt wird [SS90, S. 56]. Die ersten bekannten Momentaufnahmen wurden 1855 von Bertsch auf der Weltausstellung in Paris präsentiert. Sie zeigten eine abstrakte Szene und wirkten dadurch auf den Betrachter distanziert. Dadurch dass sie keine alltägliche Situation darstellten, konnte der Betrachter keinen Bezug zum Motiv aufbauen.

Die erste Photographie einer Straßenszene von Paris mit sich klar abzeichnenden Menschen entstand um 1860 und verblüffte viele Menschen zu der Zeit. Sie gab eine vertraute Situation wieder, als ob der Photograph die Zeit angehalten hätte. Diese ersten Momentaufnahmen sind jedoch nicht mit heutigen Aufnahmen vergleichbar. Photographen konnten sie nur unter guten Lichtverhältnissen, bei einem Mindestabstand der abzubildenden, bewegten Objekte zur Kamera und mit einem Stativ anfertigen. Heute dauern solche Aufnahmen nur einen Bruchteil einer Sekunde. Die ersten Photographien kannten weiterhin keine farbige Darstellung, sondern waren schwarz-weiß ohne Graustufen.

Mit der Weiterentwicklung des Photoapparats entstanden die ersten Serienaufnahmen und danach Chronophotographien[1]. In den folgenden Jahrzehnten entwickelte sich die Photographie stetig und schnell weiter. Inzwischen existieren so empfindliche Photomaterialien und Beleuchtungstechniken, dass gut belichtete Hochgeschwindigkeitsaufnahmen innerhalb von weniger als 25ns [Oxf10] entstehen können, was erstaunliche Momentaufnahmen von schnellen Vorgängen ermöglicht.

2.1.2. Veränderungen in der Wahrnehmung

Vor der Erfindung der Photographie wurde auf die Darstellung bzw. Andeutung von Bewegungabläufen kein Wert gelegt [Cut02, S. 1166f]. Künstler versuchten nicht ein Ereignis zu einem bestimmten Zeitpunkt darzustellen, sondern stellten komplette Erzählungen eines Ereignisses und damit eine längere Zeitspanne in einem Motiv dar.

Die Möglichkeit der Abbildung immer kürzerer Zeiträume durch die Photographie beeinflusste das zeitliche Verständnis der Menschen. Die Momentaufnahme durchbrach die bis zu ihrer Erfindung wahrgenommene Kontinuität der Zeit. Die Aufnahme immer kürzerer Zeitintervalle ermöglichte dem Menschen tiefere Einblicke in vorher nicht beobachtbare Bewegungsabläufe. Zusammen mit der aufkommenden Benutzung von Uhren als Gebrauchsgegenstand im Alltag prägte die Photographie das intuitive Verständnis einer Momentaufnahme. Heutzutage ist eine solche Interpretation von Photographien selbstverständlich.

[1] Chronophotographie steht für Photographien, die Bewegung erfassen. Serienaufnahmen gehören zu dieser Gattung, kamen aber schon vorher zu Anwendung.

2.1.3. Veränderungen in der Kunst

Die veränderte Wahrnehmung der Zeit brachte neue Betrachtungsweisen mit sich, wodurch das Thema Bewegung und deren Darstellung stärker in den Fokus von Künstlern rückte. Die Photographie inspirierte viele Künstler, denn sie bot viele Möglichkeiten Werke noch natürlicher wirken zu lassen, wenn sie die Grundlage des Werks bildete. Es gab aber auch Künstler, die der Photographie skeptisch gegenüber standen. Sie sahen in der Photographie eine Gefahr für die naturalistische Kunst, weil sie die Realität unmittelbar wiedergeben konnte und ihre große Einkommensquelle - die Porträtmalerei - ersetzte [Imm]. Anfangs betrachteten Zeitgenossen die Photographie als Ersatz zu vorherigen Kunstformen, doch schnell wurde klar, dass sie nicht allen ästhetischen Aspekten anderer Kunstformen gerecht werden konnte. Aus diesem Grund versuchten Photographen eine lange Zeit zu beweisen, dass die Photographie diesen Aspekten entsprechen konnte. Viele dieser Versuche waren glücklos und die angestrebte Nachahmung der Kunst führte dazu, dass Vertreter der klassischen Künste sich lange Zeit weigerten die Photographie als Kunstform anzuerkennen [SS90, S. 15f]. Doch mit ihrer Weiterentwicklung wurde sie schließlich eine eigenständige Strömung in der Kunst.

Für einige Künstler war die Photographie und die durch sie geänderte Wahrnehmung der Zeit eine Inspiration. Diese Künstler entwickelten neue Stile und führten abstrakte Tendenzen in die Kunst ein. Viele Impulse beeinflussten auch Künstler des Impressionismus, die einen flüchtigen Moment festhalten wollten. Andersherum wurde die Photographie auch vom Impressionismus beeinflusst. Die durch Bewegungsaufnahmen gewonnenen Einblicke waren entscheidend für die Entwicklung der Stilrichtung des Futurismus, bei dem Bewegung bzw. Bewegungsabläufe im Vordergrund stehen. Solche Bewegungsaufnahmen waren durch die Entwicklung der Chronophotographie möglich.

2.1.4. Chronophotographie

Die Chronophotographie wurde von Marey erfunden und befasst sich mit dem photographischen Festhalten von Bewegungen bzw. Bewegungsabläufen. Wichtige Vertreter der Chronophotographie waren Eadweard Muybridge und Etienne Jules Marey. Muybridge konzentrierte sich auf Serienaufnahmen, Marey hingegen stellte Bewegungen in einem einzigen Bild dar.

Eadweard Muybridge Muybridge wurde 1872 beauftragt den Beweis zu erbringen, dass ein galoppierendes Pferd während seiner Bewegungen für einen Moment in der Luft schwebt [SS90, S. 63-111]. Diese Fragestellung formulierte zuvor bereits Marey in seinem Buch *Le Machine Animal*. Diese Aufgabe stellte Muybridge vor eine Herausforderung. Zur Lösung musste er die Auslösung einer Aufnahme steuern können und zuverlässige Blendenverschlüsse entwickeln. Außerdem musste er die Empfindlichkeit des Photomaterials erhöhen.

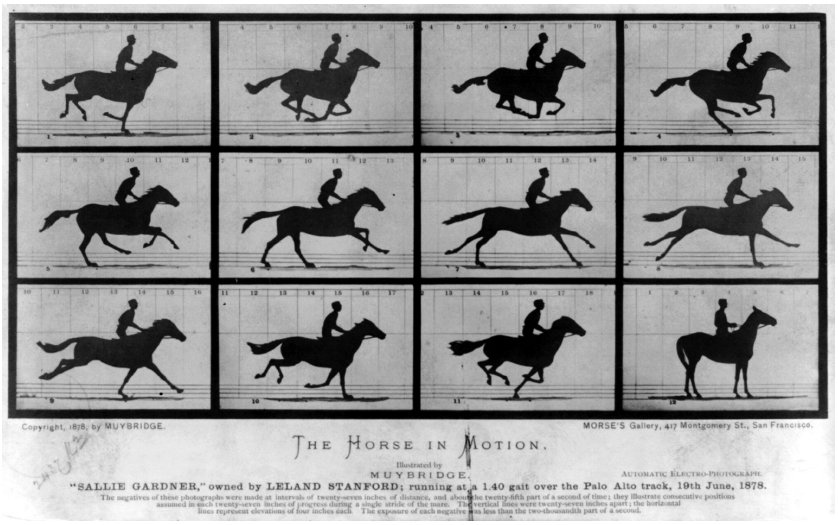

Abbildung 2.1.1.: Trabendes Pferd ohne Wagen. *The Horse in Motion* von Eadweard Muybridge (19. Juni 1878). [Wik06, SS90, S. 71f]

Für den Beweis stellte Muybridge auf einer Trabrennbahn mehrere Kameras hintereinander auf, die durch ein Wagenrad ausgelöst werden könnten. Das aufzunehmende Pferd zogen deshalb einen Rennwagen hinter sich her. Auf diese Weise erzeugte Muybridge bis dahin einzigartige Aufnahmen, die die Bewegung eines galoppierenden Pferdes dokumentieren. In Abbildung 2.1.1 ist eine ähnliche Serienaufnahme abgebildet, die ohne einen Wagen ausgelöst wurde. Bei seinem Experiment erfand er das Zoopraxiscope, das die Wahrnehmung von Photographien als kontinuierliche Bewegung ermöglichte.

Etienne Jules Marey Etienne Jules Marey konzentrierte sich in seiner Tätigkeit als Arzt darauf den Ablauf von Bewegungen so genau wie möglich graphisch festzuhalten. Er wollte Untersuchungsergebnisse nicht allein auf Beobachtungen aufbauen, sondern durch Messungen erlangen und bestätigen [SS90, S. 112-142]. Die Veröffentlichung der ersten Serienaufnahmen machten Marey 1878 auf Muybridge aufmerksam. Marey erkannte, dass Photographie sich als Unterstützung für seine Forschung eignete. Er begann zu photographieren und zu experimentieren. Er stellte eine Beziehung zwischen vergangener Zeit und Veränderung der Position her, indem er einen Bewegungsablauf in einer einzigen Aufnahme darstellte. Dafür entwickelte er eine Kamera, die eine rotierende Scheibe mit Sichtschlitzen als Blendenverschluss verwendete. Damit war es ihm möglich mehrere Aufnahmen in einer Photographie zu vereinen. Da er vor einem schwarzen Hintergrund arbeitete, war es naheliegend auch den Läufer, dessen Bewegung er dokumentieren wollte, schwarz zu kleiden. Die wichtigsten Körperpartien wurden mit weißen bzw. reflektierenden Punkten und Linien versehen. Diese Art der Aufnahmen bot einen nie da gewesenen Detailgrad der einzelnen Bewegungsphasen in Relation zueinander, vgl. Abbildung 2.1.2.

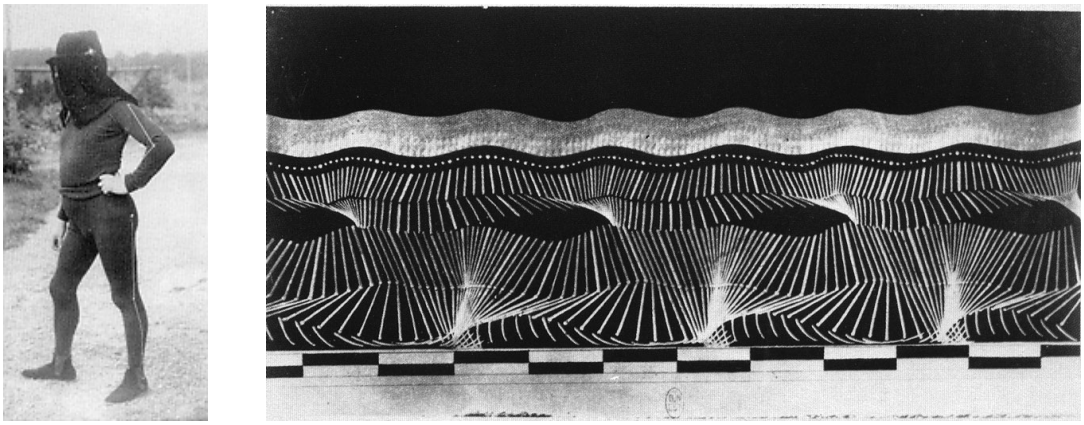

Abbildung 2.1.2.: *links*: Schwarz gekleidete Figur mit Metallbändern. (E.J. Marey 1883)
rechts: Partielle Chronophotographie, *Course de l'homme* (E.J. Marey 1883) [UCB, SS90, S. 125f]

Abbildung 2.1.3.: *links*: *Grand Prix of the Automobile Club of France* von Jacques-Henri Lartique (1911) *[Gra10, Cut02, 1181].*
Mitte: *Einzigartige Formen der Kontinuität im Raum* von Umberto Boccioni (1913), Bronze Skulptur, Museum of Modern Art (New York City) [Wik08].
unten: *Akt, eine Treppe herabsteigend Nr.2* von Marcel Duchamp (1912), Öl auf Leinwand, 146×89 cm, Philadelphia Museum of Art [Gra10, Cut02, S. 1178].

2.1.5. Bewegung in der Kunst

Jacques-Henri Lartique erschuf mit der Aufnahme *Grand Prix of the Automobile Club of France* von 1911 ein Werk, dass Bewegung suggerierte. Das linke Bild in Abbildung 2.1.3 zeigt die Aufnahme. Darauf ist ein Rennwagen zu sehen, der sich nach vorne lehnt. Im Gegensatz dazu lehnt sich die übrige Umgebung nach hinten. Um diesen Effekt zu erzielen verwendete Lartique eine Kamera mit Schlitzverschluss [Cut02, S. 1180]. Für die Aufnahme richtete er die Kamera so aus, dass der Shutter von unten nach oben über das Bild fuhr. Die Kamera selbst bewegte sich mit einer geringeren Geschwindigkeit parallel zum Rennwagen.

Futurismus Der Futurismus bezeichnet Kunst, die sich mit der Abbildung von Bewegungsabläufen beschäftigt. Wichtige Vertreter waren unter anderem Umberto Boccioni und Marcel Duchamp.

Umberto Boccioni war maßgeblich an der Entwicklung des Futurismus beteiligt. 1910 veröffentlichte er zusammen mit weiteren Künstlern das *Manifest der futuristischen Malerei* und das *technische Manifest der futuristischen Malerei*. 1913 erschuf er die in der Mitte von Abbildung 2.1.3 abgebildete Skulptur *Einzigartige Formen der Kontinuität im Raum*. Sie zeigt eine menschenähnliche Figur, die gerade geht. Dabei ist die Figur nicht zu einem bestimmten Zeitpunkt festgehalten worden, sondern die Skulptur gibteine Zeitspanne wieder, womit alle Abläufe der Bewegung gleichzeitig wirken. Die Kleidung ist vom Wind erfasst und verleiht der Skulptur damit rundere Formen. Es ist kein Gesicht zu erkennen und die Arme sind nicht dargestellt.

Ein Werk Marcel Duchamps, welches sich mit Bewegungsabläufen beschäftigt, ist *Akt, eine Treppe herabsteigend Nr.2* zu sehen im rechten Bild von Abbildung 2.1.3. Das Gemälde zeigt ineinander verschmolzene Bewegungsphasen einer Person, die eine Treppe hinunter steigt, s. . Die Person ist stark abstrahiert und besteht vornehmlich aus konischen und zylindrischen Objekten.

2.2. Bewegung in Comics

> „Neben allen Gemeinsamkeiten gibt es wesentliche Unterschiede zwischen Film und Comic. Zum einen läuft der Film in einer festen Reihenfolge der Handlung, der Szene und Schnitte in einer begrenzten Zeit ab. Der Zuschauer kann das Kino verlassen, aber er kann nicht aus dieser Festlegung ausbrechen. Der Comic dagegen kann den Ablauf der Zeit nicht gestalten, da er nur begrenzten Einfluß auf die Lesegeschwindigkeit des Betrachters und die Reihenfolge, in der die dargebotenen Bilder angeschaut werden, hat. Der Zeichner kann durch bestimmte Kniffe den Leser zu schnellem oder langsamen Lesen verleiten, dazu zwingen kann er ihn nicht." [Ihm04, S. 18]

In dieser Aussage hat Ihme die wesentlichen Unterschiede zwischen Film und Comic aufgezählt und verdeutlicht, dass es sich bei Comics um ein Medium handelt, das aus unbewegten Bildern besteht. Außerdem hat nach McCloud dieses Stumme und Starre Medium damit zu kämpfen Geräusche und Emotionen zu transportieren [Sco94, S. 118f].

Für Comics haben Will Eisner und Scott McCloud auch den Begriff *Sequentiell Art* für Comics geprägt. Er soll verdeutlichen, dass Comics eine eigenständige Kunstform sind, dieals solche Anerkennung verdient. Der wesentliche Unterschied der Comics zum Rest der Kunst ist ihr sequentieller Charakter bzw. ihre sequentielle Erzählweise. Der Comic erreicht diesen sequentiellen Charakter, indem er Panels einsetzt, die die Handlung in Sequenzen erzählen. Ein Panel ist dabei ein Rahmen, der eine Handlung oder einen Moment umfasst.

Der Comic vermittelt dem Betrachter durch verschiedene Stilmittel einen Eindruck von Bewegung:

- Bei der Wahl der Panels können
  - die Art der Übergänge,
  - ihre Form und
  - ihre Anordnung Bewegung suggerieren.
- Hinsichtlich Pose und Gestalt dienen
  - Kontrapost,
  - Verformungen und die
  - Darstellung von Wendepunkten diesem Zweck.
- Den Verlauf der Bewegung können
  - Bewegungsphasen und
  - Bewegungslinien darstellen.

Diese verschiedenen Stilmittel werden in den folgenden Abschnitten näher beleuchtet.

2.2.1. Bewegung durch Panels

In Comics werden Ausschnitte der Erzählung von Panels eingerahmt. Jedes Panel stellt üblicherweise einen neuen Ausschnitt der Geschichte dar. Die Geschichte wird also nur in Teilen dargeboten, doch der Wechsel von einem Panel zum nächsten regt die Phantasie des Lesers an, der die fehlenden Übergänge der Erzählung ergänzt. Der Leser erzeugt beim Panelübergang in Gedanken die Bewegung. McCloud beschreibt den Vorgang soweit, dass der Leser zu „einem nicht minder schuldigen Komplizen" wird [Sco94, S. 76] und nennt diesen Vorgang Induktion.

Bei der Wahrnehmung von Bewegung spielen nicht alleine Panelübergänge eine Rolle, auch Form und Komposition des Panels und des Raums zwischen den Panels beeinflussen den entstehenden Eindruck von Bewegung. Wird der Rahmen eines Panels durchbrochen, verleiht dies der Erzählung zusätzliche Dynamik.

Beim Lesen des Comics wandern die Augen über die einzelnen Panels. Die Bewegung der Augen wird dabei von der Leserichtung geleitet. Wird gegen sie verstoßen, entsteht meist ein Bruch in der Erzählung. Gegensätzliche Bewegung zur Leserichtung können aber auch gezielt eingesetzt werden, um Bewegungen härter oder weicher wirken zu lassen [Sch08, S. 71]. Es ist auch charakteristisch, dass die Erzählzeit eines Panels nicht konstant ist und im Wesentlichen von der dargestellten Situation, den Sprechblasen und der Form des Panels beeinflusst wird.

Ein Panel kann mehrere Sekunden der Erzählzeit beanspruchen, wobei sie nicht an jeder Stelle im Panel gleich ist [Sco94, S. 94-96]. Mit einer dargestellten Szene ist nicht immer eine Aufnahme gemeint, wodurch unterschiedliche Zeiten im Panel existieren können und eine Verschmelzung von Raum und Zeit erfolgt. Diese Art der Erzählung ist auch aus der Kunst bekannt, besonders bevor das Konzept der Momentaufnahme aufkam.

2.2.2. Bewegung in Pose und Gestalt

Posen können als gestalterisches Mittel den Eindruck von Bewegung entstehen lassen. Die hierfür häufig verwendeten Variationen sind der Kontrapost, die Abbildung von Wendepunkten im Bewegungsverlauf und Verformungen des Körpers.

Der Kontrapost wird bereits seit der Antike als Stilmittel eingesetzt. Er stammt vor allem aus der Bildhauerei und dient dazu einen Ausdruck von Lebendigkeit in einer Skulptur zu erzeugen. Er erzeugt Dynamik, indem er die Symmetrie der Körperhaltung aufhebt [Cut02, S. 1170]. Eine weitverbreitete Anwendung sind Posen, die klar Standbein und Spielbein unterscheiden. Die resultierenden Posen wirken so, als ob sie Teil einer Bewegung sind.

Ähnliche Effekte erzeugt die Verformung bzw. Schärung von bewegten Objekten. So kann die Bewegung eines Autos zusätzlich betont werden, indem es sich in die Bewegung hineinlehnt. Mit diesem Stilmittel kann auch Trägheit verbildlicht werden. Wird eine Person zum Beispiel stark gezogen und von dieser Beschleunigung überrascht, lässt der Zeichner sie sich nach hinten lehnen, was die Trägheit betont [Cut02, S. 1180f].

Die Auswahl des dargestellten Zeitpunkts hat einen entscheidenden Einfluss auf die Wahrnehmung der Bewegung. Ein Comic stellt Ausschnitte einer Handlung dar, indem er ihre Wendepunkte aufgreift. Die Bewegung wird also durch ihre markanten Positionen wiedergegeben. Der Betrachter vervollständigt diese Wendepunkte durch seine Vorkenntnisse zu einer Bewegung [Sch08, S. 43].

2.2.3. Verlauf der Bewegung

Bisher wurde geschildert, wie der Comic Bewegung suggeriert. Bewegung per se ist in einem unbewegten Bild nicht darstellbar. Damit Bewegung nicht nur durch den Übergang von Panel zu Panel erzeugt wird, werden Bewegungsverläufe hervorgehoben, indem z.B. die vergangene Position angedeutet wird. Durch die Ergänzung ist dem Leser intuitiv bewusst, wo sich das Objekt vorher befand. Bewegungsverläufe werden in der Regel durch Konturen bzw. Bewegungsphasen und Bewegungslinien ausgedrückt oder auch durch die Kombination mehrerer Elemente.

Comics zeigen Phasen einer Bewegung auf unterschiedliche Weise, wobei das Spektrum von großem Detailreichtum bis zur völligen Stilisierung reicht. Meist deuten Comics nur vergangene Phasen an.

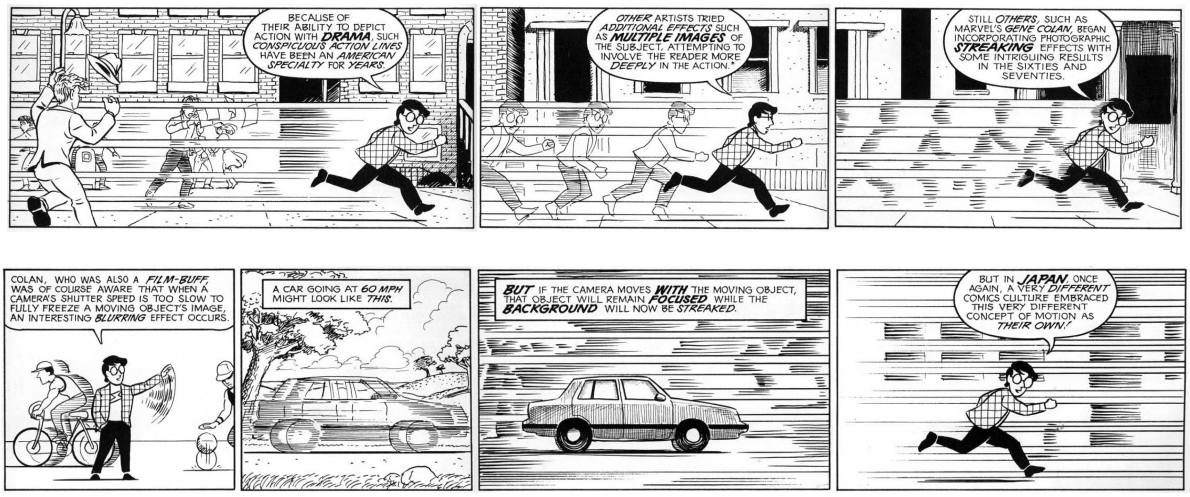

Abbildung 2.2.1.: Mehrere Bespiele für die Darstellung von Speed Lines. *In Leserichtung*: Speed Lines; Speed Lines mit Konturen; Speed Lines mit durch sie stilisierte Konturen in mehreren Beispielen; Speed Lines mit stilisiertem Hintergrund [Sco94, S. 112f].

Bewegungslinien sind Linien, die entlang der Bewegungsbahn verlaufen. Sie beginnen beim Objekt und deuten mittels ihrer Stärke und Länge die Schnelligkeit und Wucht der Bewegung an. Abgesehen von der in dieser Arbeit für Bewegungslinien verwendeten Bezeichnung Speed Lines, existieren in der Literatur auch die Bezeichnungen Action Lines und Motion Lines.

Die Verwendung von Speed Lines variiert stark von Region zu Region [Sco94, S. 110-117]. Europäische Comics setzen Speed Lines sehr reduziert ein, amerikanische Zeichner hingegen setzen sie oft und zahlreich ein. McCloud bezeichnet Speed Lines deswegen in ihrer Ursprungsform, wie er sie in Abbildung 2.2.1 oben links darstellt, als amerikanische Spezialität [Sco94, S. 112f].

Desweiteren kombiniert die Abbildung oben in der Mitte Speed Lines mit Konturen der sich bewegenden Person zu mehreren Zeitpunkten der Bewegung. Der Betrachter kann also erkennen, wo sich die Person im Verlauf der Bewegung befand und kann die verschiedenen Phasen der Bewegung zu einer fließenden Bewegung zusammensetzen. Oben rechts ist zu sehen, was McCloud als photographische Schlieren (Streaking) bezeichnet. Hierbei handelt es sich um eine Methode, die vor allem der amerikanische Comiczeichner Gene Colan verbreitete. Er stilisiert dabei die Phasen der Bewegung durch Speed Lines. So ahmt er die Verwacklungseffekte der Photographie nach, die entstehen, wenn eine ruhende Kamera bewegte Objekte aufnimmt. Beim Streaking lässt der Zeichner, sobald sich etwas in Bewegung befindet, die Konturen mittels Speed Lines verschwimmen. Die untere Reihe von Abbildung 2.2.1 zeigt links zwei weitere Beispiele des Streakings. Ein weiteres Beispiel für die Anwendung von Streaking zeigt die von Simmons erstellte Abbildung 2.2.2. Unter den amerikanischen Comiczeichnern ist Colan mit der Verwendung Streakings allerdings eine Ausnahme.

Die beiden rechten Panels von Abbildung 2.2.1 zeigen eine auf Streaking basierende Entwicklung. Anstatt die Aufnahme von Bewegung durch eine ruhende Kamera nachzuahmen, be-

Abbildung 2.2.2.: *links*: Bild aus einer Beispielanimation von Simmons [Sim]. Speed Lines stellen die Handbewegung einer Person verschwommen dar.
rechts: Kampfszene aus dem japanischen Comic „Berserk". Die Umrisse der Kämpfenden sind nur noch mit Speed Lines dargestellt [Miu03].

schäftigen sich die Panels damit, was passiert, wenn die Kamera sich synchron zum Objekt bewegt. In diesem Fall verschwimmt der Hintergrund, das Objekt selber ist allerdings klar wahrnehmbar. Comics erzielen diesen Eindruck, wenn sie den Hintergrund durch Speed Lines, die sich an den dort vorhandenen Konturen orientieren, ersetzen. Die japanische Comickultur hat sich diesen Effekt zu eigen gemacht, wie das rechte Panel der unteren Reihe zeigt. Schwürer [Sch08, S. 71f] bezeichnet diese Art von Bewegungslinien als Fließlinien. Fließlinien deuten nicht die bereits geschehene Bewegung an, sondern weisen in die Zukunft [Sch08, S. 71f]. Der Zeichner verlängert die Linien dabei über den bisherigen Bewegungsverlauf hinaus. Fließlinien verlaufen entweder parallel in die gleiche Richtung oder auf einen Punkt im Raum zu.

Ein Beispiel, das die Verwendung von Speed Lines sehr anschaulich wiederspiegelt, stammt aus dem japanischen Comic *Berserk* von Kentaro Miura und ist im rechten Bild von Abbildung 2.2.2 aufgeführt. Zum einen verwendet Miura Streaking, um die Dynamik des Kampfes zwischen den beiden Personen aufzuzeigen. Besonders betont er hierbei die Bewegung der Waffen, deren Konturen noch weitaus unschärfer sind, als die der Personen. Zum anderen verwendet er im rechten Bereich des Bildes im Hintergrund Fließlinien, die beim Betrachter den Eindruck erwecken, dass die rechte Person nach rechts zurückgeschleudert wird.

Die Darstellung von Speed Lines und den einzelnen Phasen der Bewegung hilft komplizierte Bewegungsabläufe auszudrücken, die sonst nur durch Bilderfolgen verständlich wären [Ihm04, S. 20]

2.3. Kategorisierung von Bewegungsdarstellungen

Es existieren viele unterschiedliche Methoden um Bewegung darzustellen. Im Folgenden kategorisiert dieser Abschnitt Bewegungsdarstellungen aufbauend auf der Arbeit von Cutting

[Cut02, Nie06, S. 75f]. Cutting unterscheidet die Darstellungsformen von Bewegung in fünf Arten. Einige dieser Bewegungsformen sind aus dem vorangegangenen Abschnitt bekannt, auch in Form von Beispielen. Auf die photographische Unschärfe geht Kapitel 4 näher ein.

Dynamische Balance bzw. Gebrochene Symmetrie umfassen als Kategorie die Bewegungsdarstellungen, die durch Überspitzung oder die Aufhebung der Symmetrie wirken oder Bewegung im Wendepunkt ihres Verlaufs einfangen, vgl. Unterabschnitt 2.2.2.

Mehrere stroboskopische Bilder, also mehrere Einzelbilder eines bewegten Motivs, verschmelzen zu einem Bild und stellen eine weitere Kategorie, vgl. Abbildung 2.1.4 und Unterabschnitt 2.2.3.

Affine Schärung bzw. Vorwärts-Lehnen beschreibt die Schärung eines Objekt in bzw. gegen seine Bewegungsrichtung, vgl. Unterabschnitt 2.1.5 und Unterabschnitt 2.2.2.

Photographische Unschärfe entsteht durch die Aufnahme von schnellen Objekten oder durch Langzeitbelichtungen.

Speed Lines simulieren Geschwindigkeit, indem sie die erfolgte Bewegung durch Linien andeuten, vgl. Unterabschnitt 2.2.3.

Bei Betrachtung der Bewertungskriterien Cuttings werden die Unterschiede der Darstellungskategorien deutlich. Für ihn ist entscheidend,

- ob die *Darstellung Bewegung suggeriert* und somit ein Gefühl von Bewegung hervorruft,
- ob die *Darstellung klar ist*, insofern dass das Objekt immer klar zu erkennen ist,
- ob die *Bewegungsrichtung* erkennbar ist und
- ob die *Bewegung genau* abgebildet ist, der Betrachter also den Verlauf der Bewegung und ihre Stärke an der Darstellung ablesen kann.

Die Tabelle 2.1 führt anhand dieser Kriterien Cuttings eine Bewertung der Kategorien auf. Aufgrund der Auswertung dieser Tabelle zieht Cutting folgende Schlüsse: Die Dynamische Balance kann Bewegung vermitteln, aber in keiner Weise ihren Verlauf beschreiben. Stroboskopische Bilder sind nicht in der Lage die Bewegungsrichtung hervorzuheben. Die affine Schärung lässt nicht zu, die Stärke der Bewegung objektiv zu beurteilen und die photographische Unschärfe kann ausschließlich ein Gefühl von Bewegung vermitteln, stellt das bewegte Objekt aber nicht deutlich genug dar, um weitere Informationen zu transportieren. Bei photographischer Unschärfe kann das Objekt sogar völlig verschwinden. Cutting schließt daraus, dass nur Speed Lines alle gewünschten Kriterien erfüllen und Bewegung geeignet darstellen. Ihre Darstellung bildet das Objekt klar ab, macht die Bewegungsrichtung ersichtlich sowie die Distanz, die das Objekt überwindet. Speed Lines sind also besonders gut dazu geeignet möglichst viele Informationen über eine Bewegung zu transportieren.

Stilarten	Suggestion von Bewegung	Klarheit der Darstellung	Bewegungs-richtung	Genauigkeit der Bewegung
Dynamische Balance bzw. Gebrochene Symmetrie	✓	✓	∼ ✓	×
Mehrere stroboskopische Bilder	✓	✓	×	✓
Affine Schärung	✓	✓	✓	×
Photographische Unschärfe	✓	×	×	×
Speed Lines	✓	✓	✓	✓

Tabelle 2.1.: Bewertung verschiedener Arten von Bewegung anhand von vier Kriterien. Übertragen von [Cut02].
✓ (erfüllt); ∼ (abhängig von der Erfahrung des Betrachters); × (nicht erfüllt).

2.4. Zusammenfassung

Es wurden in verschiedenen Varianten gezeigt, wie die Kunst Bewegung andeutet bzw. den Eindruck von Bewegung erzeugt. Insbesondere ist der Effekt des Streakings interessant, eine Verschmelzung von photographischer Unschärfe und Speed Lines. Dieser Effekt existiert, wie Unterabschnitt 2.2.3 beschreibt, auch im Comic. Speed Lines können zur Darstellung von Motion Blur beitragen und eine besondere Form der Stilisierung erzeugen. Zur Zeit gibt es kein Verfahren in der Computergraphik, das versucht diesen Effekt nachzustellen.

Diese Arbeit konzentriert sich auf die Darstellung Streaking. Aus diesem Grund greifen die folgenden Kapitel Techniken aus dem Themengebiet der Computergraphik auf, speziell Themen des Non-Photorealistic Renderings und Motion Blur Verfahren.

3. Non-Photorealistic Rendering

Abschnitt 3.1 definiert das Non-Photorealistic Rendering (NPR) und stellt die zu unterscheidenden Teilgebiete vor. Abschnitt 3.2 beschreibt daraufhin mehrere grundlegende Verfahren. Abschnitt 3.3 beleuchtet den momentanen Stand der Forschung bezüglich Bewegungsdarstellungen. Abschnitt 3.4 bewertet die vorgestellten Verfahren. Abschließend fasst Abschnitt 3.5 die gewonnenen Erkenntnisse zusammen.

3.1. Übersicht

Non-Photorealistic Rendering (NPR) deckt ein außerordentlich breites Spektrum an Themengebieten ab. Es gehören sämtliche Verfahren und Ansätze zu diesem Themengebiet, die eine nicht photorealistische Darstellung zum Ziel haben. Nach Strothotte [SS02, S. 7-30] ist das Ziel von NPR Verfahren nicht photorealistische Darstellungen formal zu beschreiben und Computerprogramme zum Zeichnen dieser Darstellungen zu entwickeln.

Strothotte identifiziert vier Strömungen, die im Fokus der Forschung stehen. Diese Strömungen können nicht klar voneinander getrennt werden, sondern mehrere können in ein und demselben Verfahren zum Ausdruck kommen. Die vier Strömungen können anhand ihrer Ziele unterschieden werden:

- Nachahmung menschlicher Zeichnungen und Bilder

  Diese Strömung versucht von Hand erstellte Zeichnungen zu imitieren. Ihr Ziel ist eine Zeichnung zu rendern, die nicht von einer von Hand erstellten Zeichnung unterschieden werden kann. Bisweilen bietet sie auch eine Schnittstelle an um am Computer Darstellungen anzufertigen. Programme können z. B. das Malen mit Tusche oder Acryl simulieren oder Bilder zu Zeichnungen und Malereien transformieren, vgl. [BL04].

- Bestmöglicher Informationstransport

  Diese Strömung argumentiert, dass NPR-Darstellungen sehr gut spezifische Informationen transportieren können, besser noch als Photographien oder photorealistische Darstellungen. Um dies zu erreichen reduziert sie die Darstellungen auf nötige Informationen oder stellt die zu transportierende Information in den Mittelpunkt. Der Erfolg der Informationsübermittlung ist hierbei an mehrere Einflussgrößen gebunden, wie z.B. die Zeit, die der Betrachter benötigt um die Information zu verstehen, die Fehlerrate sowie

kulturelle Aspekte. Diese Strömung generiert beispielsweise reduzierte technische oder medizinische Zeichnungen oder färbt Flächen ein, die im Schatten liegen.

- Untersuchung der Beziehung zwischen natürlicher und bildlicher Sprache

  Diese Strömung versucht die Frage zu beantworten, wann die natürliche Sprache die bessere Wahl ist, um Informationen zu transportieren und wann eine bildliche Darstellung diese Aufgabe ebenso gut, wenn nicht sogar besser bewerkstelligen kann.

- Anbieten von Interaktivität

  Mit NPR-Techniken ist es möglich in neuen Medien Wissen in Form von 3D-Modellen wiederzugeben und dadurch ein höheres Maß an Interaktion zu erlangen. Der reduzierte Charakter der Zeichnungen und Modelle trägt dabei zur Veranschaulichung der Information bei. Bei einem interaktiven, eventuell sogar animierten Modell ist es dem Benutzer möglich, für sich selbst die geeignetste Anschauung zu wählen. Sehr hilfreich kann dies z.B. beim Verständnis von medizinischen Modellen sein.

3.2. Verfahren

Die Farbe eines Objekts transportiert nicht nur Informationen über Objekteigenschaften, sondern lässt auf die Form und die Position des Objekts in Relation zum Licht schließen. Durch die Reduktion von Farbe sollen besondere Teilaspekte des Bilds bzw. Objekts hervorgehoben werden. Oft ist es notwendig Informationen darzustellen, die durch einen photorealistischen Ansatz verloren gehen würden, weil sie z.B. unbeleuchtet im Schatten liegen. Dieser Abschnitt stellt mit Cartoon- und Gooch-Shading zwei sehr bekannte NPR-Verfahren, die der Farbreduktion dienen, vor.

Daraufhin beschreibt der Abschnitt Hatching, das einem Bild das Aussehen einer Bleistiftzeichnung verleiht. Es existieren verschiedene Herangehensweisen um Hatching umzusetzen.

Zuletzt beschreibt der Abschnitt Verfahren, die Silhouetten erzeugen. Silhouetten bieten zusätzliche Informationen über Objekte, weil sie die Objektumrisse andeuten und Objektfalten und Materialkanten betonen. Silhouetten können besonders für technische Zeichnungen verwendet werden, wie z.B. für Bauanleitungen von Möbeln.

3.2.1. Cartoon-Shading

Cartoon-Shading reduziert die Farben der darzustellenden Objekte. Das Cartoon-Shading nach Lake [Dec96, LMHB00, GG01, S.170-173] reduziert die Farbe von Objekten auf wenige Helligkeitsstufen. Dieses Verfahren bewirkt flächige Farben und erinnert damit, wie der Name schon sagt, an Cartoons. Viele Programme verwenden das auch als Toon-Shading bezeichnete Verfahren in Kombination mit Silhouetten.

Cartoon-Shading reduziert die Farben, indem es die vorhandenen Farben auf ein Spektrum weniger Farben abbildet.

Durch Verwendung von Shadersprachen erfolgen die Berechnungen auf der GPU. Listing 3.1 und Listing 3.2 zeigen die Berechnung der Beleuchtung im Cartoon-Shading in Form eines Vertex- und eines Fragment-Shaders. Statt Farben liegen Intensitäten dabei als eindimensionale Textur vor [Fer]. Um mehrere Abstufungen zu erzielen kann diese Textur so geändert werden, dass sie nicht nur aus zwei Intensitäten besteht, sondern mehrere Intensitäten enthält. Die Benutzung einer Textur bietet ein hohes Maß an Flexibilität, weil die Shaderprogramme keiner Änderung bedürfen um andere Ergebnisse zu erzielen. Der Vertex-Shader berechnet die Farbe auf Basis der Materialfarbe und der Beleuchtung, wobei er die Glanzeigenschaft des Objekts nicht berücksichtigt.. Um den endgültigen Intensitätswert zu berechnen, interpretiert der Fragment-Shader dann die berechnete Intensität als Texturkoordinate. Dadurch reduziert Cartoon-Shading die vorhandenen Intensitäten auf ein geringeres Maß.

3.2.2. Gooch-Shading

Gooch-Shading, entwickelt von Gooch et al. [GGSC98, GG01, S. 175-188], bewirkt keine Farbreduktion, sondern stellt Bilder bzw. Objekte so illustrativ dar, dass im Schatten liegende Flächen gut zu erkennen sind. Außerdem unterstützt Gooch-Shading die Darstellung von Silhouetten, die meistens schwarz sind.

Gooch-Shading nutzt warme und kalte Farben um ein Gefühl für beleuchtete Flächen zu bieten und die Form zu betonen. Warme Farben sind z.B. rot, orange und gelb, kalte blau, violett und grün. Gooch-Shading nutzt den Umstand, dass warme Farben in den Vordergrund und kalte Farben in den Hintergrund treten. Bei einem Farbwechsel von warm zu kalt entsteht so ein Tiefeneindruck.

Das Vertex-Programm in Listing 3.1 entspricht mit der Ausnahme des Wegfalls der Variable `ambientColor`, einer Implementation für Gooch-Shading. Den entscheidenden Beitrag leistet auch hier der Fragment-Shader, den Listing 3.3 aufführt, wobei auch in diesem Fall der Shader

```
1 out vec4 ambientColor;
2
3 void main(){
4    position = modelView * element.position;
5    gl_Position = projection * position;
6    normal = normalMatrix * element.normal;
7
8    ambientColor = material.ambientColor * (global.ambientColor +
        lightSource.ambientColor);
9 }
```

Listing 3.1: Vertex-Shader Beispielcode für Cartoon-Shading.

```
uniform sampler1D intensityLUT

in vec4 ambientColor;
out vec4 fragColor;

void main(){
   vec3 lightVector = normalize(position - light.position);
   float intensity = dot(lightVector, normalize(normal));
   intensity = (intensity + 1) * 0.5;

   fragColor = material.diffuseColor * textur(intensityLUT, intensity) +
       ambientColor;
}
```

Listing 3.2: Fragment-Shader Beispielcode für Cartoon-Shading.

```
uniform sampler1D coolToWarmLUT, intensityLUT;

out vec4 fragColor;

void main(){
   vec3 lightVector = normalize(position - light.position);
   float intensity = dot(lightVector, normalize(normal));
   intensity = (intensity + 1) * 0.5;

   fragColor = material.diffuseColor * textur(intensityLUT, intensity) +
       textur(coolToWarmLUT, intensity);
}
```

Listing 3.3: Fragment-Shader Beispielcode für Gooch-Shading.

keine Glanzlichter berechnet. Der Fragment-Shader verwendet zwei eindimensionale Texturen. Eine der Texturen enthält Intensitäten. Sie dient der Schwächung aller vorhandener Farben. Die zweite Textur beschreibt einen Farbverlauf von einer kalten Farbe zu einer warmen. Gooch et al. schlagen als Farben blau und gelb vor. Gooch-Shading verwendet die ursprüngliche Intensität als Texturkoordinate und greift so auf beide Texturen zu. Die Texturen beeinflussen die Farbe, die einem Fragment zugewiesen wird, indem sie sie mit den aus den Texturen ausgelesenen Werten verrechnen. Hierbei gilt, dass je weiter ein Fragment vorne liegt, desto wärmer wird seine Farbe, und je weiter es hinten liegt umso kälter. So spiegelt Gooch-Shading die Tiefe der Szene farblich wieder.

3.2.3. Hatching

Intensitätsunterschiede können auch durch Schraffuren dargestellt werden, das als Hatching bezeichnet wird. Das bildraumbasierte Hatching-Verfahren von Lake et al. [LMHB00] funktioniert im Grunde ähnlich wie die bisher vorgestellten Verfahren. Sie berücksichtigen die Lichtintensität und verwenden unterschiedliche Schraffuren für Materialien. Diese Bildraumverfahren können aber nicht die Form eines Objekts betonen, weshalb die gesamte Szene an Plastizität verliert.

Hertzmann et al. [HZ00], Praun et al. [PHWF01, WPFH02], Winkelbach et al. [WS94] und Zander et al. [ZISS04] stellen objektbasierte Verfahren vor. Die vorliegenden Arbeit beschränkt sich auf das Verfahren von Praun et al., weil später die Tonal Art Maps in dieser Arbeit zum Tragen kommen. Das Verfahren verwendet über ein Objekt zufällig verteilte, sich gegenseitig überlappende und an einem Vektorfeld ausgerichtete Schraffurtexturen (Lapped Textures [PFH00]). Verschiedene Intensitäten, repräsentiert durch Schraffurtexturen, werden in einer so genannten Tonal Art Map (TAM) gespeichert, die eine Weiterentwicklung von Winkelbach et al. [WS94] ist. Durch sie werden die Intensität und Form des Objekts wiedergeben. Gleichzeitig wird durch die Tonal Art Map sichergestellt, dass ein Objekt, wenn es aus unterschiedlichen Entfernungen betrachtet wird, die gleiche Strichdicke aufweist. Durch die Lapped Textures wird verhindert, dass Übergänge zwischen den einzelnen Texturen sichtbar werden, womit es möglich ist, beliebige Arten von Geometrie zu verwenden.

3.2.3.1. Tonal Art Map

Um verschiedenste Intensitäten darzustellen, werden Schraffuren mit unterschiedlichen Intensitäten benötigt. Für einen späteren besseren Zugriff werden sie nach ihren Intensitäten geordnet in ein Texturarray abgelegt. Ein Texturarray mit unterschiedlichen Schraffurtexturen ist aber dennoch noch keine TAM.

Mehrere Eigenschaften machen nach Vix [Vix08, S. 17-25] eine TAM aus. Es existiert eine Schraffur für jeden Intensitätsbereich. Jede MipMap-Stufe wird manuell für jeden Intensitätsbereich

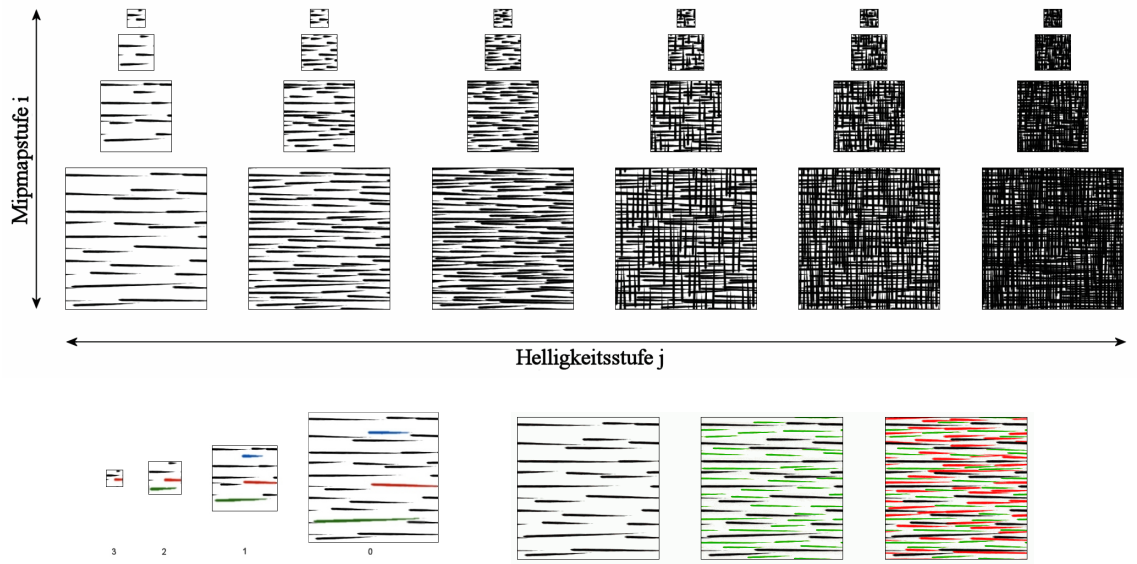

Abbildung 3.2.1.: oben: Tonal Art Map mit verschiedenen Helligkeits- und MipMap-Stufen. *unten*: Kohärenz zwischen den MipMap-Stufen (*links*) und den Helligkeitsstufen (*rechts*) [Vix08, S. 21f].

definiert. Jede Textur der feinsten MipMap-Stufe besitzt eine Größe von 2^n für jede Seite. Schraffuren in den verschiedenen MipMap-Stufen und Intensitätsbereichen besitzen eine hohe Kohärenz zueinander.

Eine Texture $T_{x,\,y}$ gehört zu einem Intensitätsbereich $[a,\ b]$ mit dem Mittelwert i_x, zur MipMap-Stufe l_y und besitzt eine Menge von Strichen $S_{x,\,y}$. Es gibt insgesamt m Intensitätsbereiche, wobei 0 für den höchsten und $x = m - 1$ für den niedrigsten Helligkeitswert steht. Die oberste MipMap-Stufe 0 enthält die größte bzw. feinste Textur und die niedrigste Stufe n die kleinste bzw. gröbste Textur. Jede Textur in der in der Stufe l hat eine Seitenlänge von 2^{n-l} und ist quadratisch.

Der übliche Generierungsprozess von MipMaps bewirkt bei gröberen Stufen kleine Striche und eine zunehmende Vergrauung, weshalb ein entferntes Objekt grau wird. Praun et al. schlägt eine tiefenunabhängige Darstellung vor. Um keine störenden Effekte bei der Variation der Tiefe oder der Intensität zu erlangen, müssen die Striche die gleiche absolute Pixelbreite aufweisen und die gleiche relative Position und Länge haben. Außerdem müssen folgende Kohärenzen gelten:

$$\begin{aligned} \text{Intensitäts-Kohärenz} &\ \hat{=}\ & S_{x,\,y} \subset S_{x',\,y} \quad & \text{mit } x < x' \\ \text{MipMap-Kohärenz} &\ \hat{=}\ & S_{x,\,y} \subset S_{x,\,y'} \quad & \text{mit } y > y' \end{aligned}$$

Dies bedeutet, dass die Strichmenge einer Textur in der nächsten niedrigeren MipMap-Stufe und ebenfalls im nächsten höheren Intensitätsbereich enthalten ist. Um einen Eindruck für die

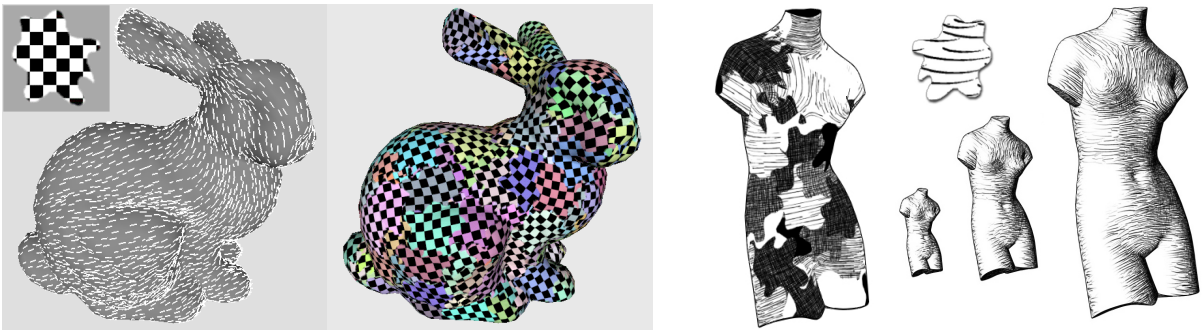

Abbildung 3.2.2.: Krümmungsfeld (*links*) mit aufgetragener karierter Textur auf ein Hasenmodell nach dem Lapped Texture Ansatz [PFH00, S. 3]. Eine Büste Hervorhebung der Lapped Textures und das Ergebnis in verschiedenen MipMap-Stufen [PHWF01, S. 4f].

Koheränzen zu bekommen, sind in Abbildung 3.2.1 Striche, die in nächstgröberen MipMap-Stufen oder dunkleren Intensitätsstufen vorkommen, farbig markiert. Eine Erweiterung der TAM auf dreidimensionalen Texturen wird von Webb et al. vorgestellt [WPFH02].

3.2.3.2. Krümmungsfelder

Die TAM selbst kann nur Schraffuren von Helligkeitsunterschieden andeuten, aber eignet sich nicht dazu die Form wiederzugeben. Damit die Form besser zur Geltung kommt werden Krümmungsfelder benötigt, an denen die Texturen ausgerichtet werden, damit die Form besser betont wird. Ein Krümmungsfeld für ein Modell kann vorabberechnet werden und beim Rendern als Richtungsattribut an den Shader weitergegeben werden.

In der Arbeit von Vix [Vix08, S. 42-53] wird genauer auf die Berechnung von Krümmungsfeldern eingegangen.

3.2.3.3. Lapped Textures

Sich überlappende Texturen spielen beim Verfahren von Praun et al. eine große Rolle. Durch das Verfahren wird eine Textur über ein Patch mehrmals über das Objekt gelegt und anhand des Krümmungsfelds ausgerichtet. Dabei überlappen sich Texturen bzw. Patches und die quadratischen Texturränder sind gut erkennbar. Um dieses Problem zu vermeiden, müssen die Kanten der verwendeten Textur unkenntlich gemacht werden. Die Verwendung einer Alphamaske erbringt den gewünschten Effekt und kann gerade Texturränder vermeiden. Die Texturränder orientieren sich an den offensichtlichen Rändern der Muster in der Textur selber. Für den Fall, dass keine Ränder automatisch anhand von Kanten ausgemacht werden können, muss der Rand manuell definiert werden. Beispiele aus den Arbeiten von Praun et al. sind in Abbildung 3.2.2 dargestellt.

3.2.3.4. Zusammenführung

Das Erstellen von geeigneten Texturen, die Berechnung des Vektorfelds und die sich überlagernden Texturen, werden vorab berechnet. Es bleibt schließlich nur noch die Berechnung der Intensität der Schraffur für einen Oberflächenpunkt und die Auswahl der richtigen Schraffurtextur.

Nach der Berechnung der Intensität $max\left(\vec{L} \circ \vec{n},\, 0\right)$, werden die am nächsten liegenden Texturen aus der TAM genommen und ineinander übergeblendet. Durch das Verfahren entstehen beeindruckende Bilder, die die Form und Beleuchtung von Objekten als Schraffur sehr realistisch wiedergeben, vgl. Abbildung 3.2.2.

3.2.4. Silhouetten

Silhouetten betonen besonders die Umrisse von Objekten. Hinzukommen Objektfalten und Materialkanten. Sie sind besonders beim Rendern im Comicstil und bei der Darstellung von technischen Zeichnungen wichtig. Bei Comics sind sie ein Stilmittel, aber bei technischen Zeichnungen sind sie sehr nützlich, weil bei Konstruktionen die Betonung für die bessere Erkennung von Änderungen im Material oder unscheinbaren Kanten wichtig sind. Vor allem können beim photorealistischen Ansatz durch unterschiedliche Lichtverhältnisse nicht immer Kanten ausgemacht werden oder sind im Schatten nicht zu erkennen.

Es existieren verschiedene Arten von Verfahren zum Generieren von Silhouetten. Darunter befinden sich Verfahren basierend auf dem Objekt- oder Bildraum. In den folgenden Erläuterungen beschränkt sich diese Arbeit auf echtzeitfähige Verfahren.

Es wird im weiteren Verlauf auf Objekt- und Bildraumverfahren eingegangen. Verfahren im Objektraum haben Zugriff auf die Geometrie von verwendeten Modellen und operieren mit diesen Daten. Im Gegensatz dazu verwenden Bildraumverfahren vorgerenderte Bilder und haben keinen Zugriff auf die Geometrie. In der Literatur taucht häufiger der Begriff G-Buffer auf, der Informationen über die vordersten Objekte in der Bildebene abspeichert. G-Buffer können Bilder bzw. Textur sein, die Attribute wie Farbe, Normalen, Geschwindigkeit etc. beschreiben und über Frame Buffer Objekte erstellt werden.

3.2.4.1. Objektraum basierte Verfahren

Das Verfahren nach Buchanan et al. [BS00, GG01, S. 132-134] erzeugt seine eigene Datenstruktur, einen so genannten Edge-Buffer, indem alle Kanten eines Meshs enthalten sind. Der Edge-Buffer wird vor dem Rendern initialisiert und vor jedem Rendern aktualisiert. Es wird für jede Kante festgestellt, ob sie sichtbar ist, woraufhin ein entsprechender Eintrag im Edge-Buffer erfolgt. Gefundene Kanten werden als zusätzliche Geometrie gerendert und bieten auf diese Weise Möglichkeiten zur Stilisierung.

Hermosilla et al. [HV10] erzeugten Silhouetten unter Verwendung moderner Graphikhardware, während die zu zeichnende Geometrie gerendert wird. Dabei ist es wichtig, dass dem Geometry-Shader Nachbarschaftsinformationen (Adjazenzen) über benachbarte Dreiecke in der Geometrie vorliegen. Im Geometry-Shader wird dann anhand der Oberflächennormalen und dem Blickvektor des Betrachters ermittelt, an welcher Kante des eingehenden Dreiecks sich eine Silhouettenkante gebildet hat. Für den Fall, dass eine Kante besteht, wird ein Strichpolygon erzeugt. Ring et al. [RC10] zeigen, dass das Verfahren auch ohne Informationen über Adjazenzen funktioniert, aber behandeln nicht die Stilisierung der Silhouetten.

Beide Verfahren bieten weitere Optionen, wie z.B. die Silhouetten zu stilisieren und den individuellen Bedürfnissen anzupassen. Sie hängen aber stark von der Größe der verwendeten Modelle bzw. der gesamten Szene ab.

Ein weiterer Ansatz von Raskar et al. [RC99, GG01, S. 125-129] rendert die Szene zuerst und zeichnet dann im zweiten Durchlauf alle abgewandten Flächen. Die abgewandten Flächen werden leicht zum Betrachter hin verschoben und schwarz gezeichnet, wodurch der Eindruck von Kanten entsteht. Das Verfahren hat den Nachteil, dass die Strichdicke der erzeugten Silhouetten nicht gesteuert werden kann und keine Stilisierung der Kanten möglich ist.

3.2.4.2. Bildraum basierte Verfahren

Bei der Anwendung von Silhouetten kann auf ein anderes bekanntes Bildraumverfahren von Saito et al. [ST90, GG01, S. 120-123] zurückgegriffen werden. Dabei wird die Szene gezeichnet und in verschiedenen Texturen abgespeichert. Es werden Informationen über die Tiefe und die Oberflächennormale benötigt. Auf den beiden vorliegenden Texturen wird der Sobel-Filter angewandt, durch den Kanten hervorgehoben werden. Die resultierenden Kantenbilder müssen über eine Transformation in ihren Wertbereich angepasst werden, weil dieser im Bereich $[-1, 1]$ liegt. Danach wird ein Schwellwertverfahren angewandt und das Ergebnis besteht aus schwarzen Linien für Kanten und einem weißen Hintergrund. Durch die Multiplikation dieses Bilds mit dem Farbbild der Szene erscheinen Kanten im Endergebnis.

Vix [Vix08, S. 54-74] stellt eine Implementation des Verfahrens von Saito als Teil seiner Arbeit über Real-Time Hatching auf gescannten 3D-Objekten vor. Für jedes Fragment im Shader greift er für den Sobel-Filter auf die umliegenden Werte zu. Er macht sich dabei die Nearest Neighbour Interpolation von Texturwerten zu nutze. Wenn mit der Texturkoordinate genau zwischen zwei Pixel zugegriffen wird, dann wird das Ergebnis der Interpolation der beiden Farbwerte zurückgegeben. Dadurch können die Texturzugriffe von neun auf fünf reduziert werden.

Diese Art Silhouetten zu erstellen erfordert konstanten Zeitaufwand beim Rendern einer beliebigen Szene und ist nur abhängig von der gewählten Größe des Frame Buffer Objekts. Es bietet aber keine Möglichkeit Silhouetten stilisiert zu zeichnen.

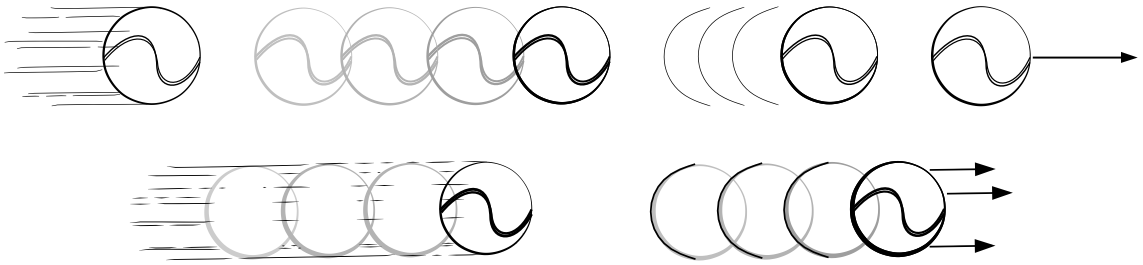

Abbildung 3.3.1.: *oben*: Ein geworfener Ball mit zufälligen, segmentierten Speed Lines, sich wiederholenden, blasser werdenden Konturen, wiederholenden Teilkonturen und Bewegungslinien.
unten: Kombinationen verschiedener Stile können das Gefühl für Bewegung verstärken.[MSS99]

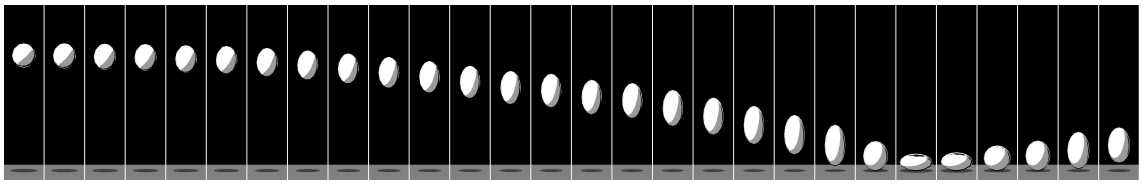

Abbildung 3.3.2.: Darstellung einer zeitlichen Abfolgen eines vertikal hüpfenden Balls.
von links nach recht: der Ball fällt, streckt sich, wird bei der Kollision zusammengestaucht und streckt sich abermals beim Abdrücken [CPIS02].

3.3. Bewegungsdarstellungen - Stand der Forschung

Um Bewegungen in Bilder kenntlich zu machen, gibt es verschiedenste Stile und Verfahren, die z.B. Objekte verformen, ihre Konturen oder Teile davon wiederholen oder mit Bewegungslinien (Speed Lines) ihre Bewegungsrichtung andeuten. Abbildung 3.3.1 illustriert einige angesprochenen Stile und ist gleichzeitig eine computergenerierte Darstellung von Masuch et al. [MSS99].

Wie Verformungen von Objekten erreicht werden können, zeigt Unterabschnitt 3.3.1. Einige Methoden zur Erstellung von Konturen stellt der Unterabschnitt 3.3.2 vor. Anschließend geht Unterabschnitt 3.3.3 näher auf Speed Lines und wie sie mit unterschiedlichen Verfahren erstellt werden ein. Zum Schluss wird ein Hybridverfahren gezeigt, welches Speed Lines mit der Verformung des Objekts kombiniert.

3.3.1. Verformte Objekte

Durch die Verformung von bewegten Objekten zur Bewegungsrichtung wird ein Bewegungseindruck erzeugt. Die ausgedrückte Bewegung gewinnt an Dynamik und wirkt lebendiger. Besonders bei der Anwendung auf einen Ball, der bei einer Kollision zusammendrückt wird, entsteht der Eindruck, dass er mit der Umgebung interagiert. Dieses Phänomen kann in Abbil-

dung 3.3.2 gut nachvollzogen werden. Umgekehrt vermittelt ein gestreckter Ball den Eindruck, dass er sich durch die starke Geschwindigkeit verformt.

Chenney et al. [CPIS02, HHD04] stellen ein Verfahren vor, welches Objekte strecken bzw. zusammenstauchen kann. Dafür werden Informationen wie Geschwindigkeit und Beschleunigung benötigt und unter Einsatz von Shadern in eine Streckung der Geometrie umgewandelt.

Das Verfahren dehnt Objekte im Ruhezustand nicht, wobei die Streckung mit zunehmender Geschwindigkeit stärker wird. Beim Beschleunigen wird das Objekt zusätzlich gestreckt und beim Abbremsen gestaucht. Zur Beschreibung dieses Verhaltens werden sowohl die maximale Geschwindigkeit v_{max} und die maximale Beschleunigung a_{max} benötigt, als auch die Wertebereiche der Skalierungsfaktoren $[1, f_v^{max}]$ für die Geschwindigkeit f_v und $\left[f_a^{min}, f_a^{max}\right]$ für die Beschleunigung f_a, wobei $0 < f_a^{min} \leq 1 \leq f_a^{max}$ gilt. Die Skalierungsfaktoren werden wie folgt berechnet:

$$
\begin{aligned}
f_v &= 1 + \frac{v}{v_{max}} \cdot (f_v^{max} - 1) \\
f_a &= \begin{cases} 1 + \frac{a}{a_{max}} \cdot (f_a^{max} - 1) & \text{für } a \geq 0 \\ 1 + \frac{a}{a_{max}} \cdot \left(1 - f_a^{min}\right) & \text{für } a < 0 \end{cases}
\end{aligned}
$$

Der Faktor f_v wird niemals < 1 und f_a nicht ≤ 0. Die Skalierung erfolgt entlang der Bewegungsrichtung und ist die Kombination aus f_v und f_a. Bei der Streckung ist es wichtig, dass das Objekt weder an Volumen zunimmt noch verliert, weshalb es senktrecht zur Bewegungsrichtung mit dem Faktor f_{norm} gestaucht wird. Außerdem existiert eine Transformationsmatrix D, die diese Deformation beschreibt:

$$
\begin{aligned}
D &= \begin{pmatrix} f_{result} & 0 & 0 \\ 0 & f_{norm} & 0 \\ 0 & 0 & f_{norm} \end{pmatrix}, \\
&\text{mit } f_{result} = f_v \cdot f_a \text{ und } f_{norm} = \sqrt{\frac{1}{f_{result}}}
\end{aligned}
$$

Um die Deformation anzuwenden, muss das betreffende Objekt in den Koordinatenursprung verschoben werden und in die Bewegungsrichtung rotiert werden. Nach der Deformation müssen alle vorherigen Schritte wieder rückgängig gemacht werden.

Auf diese Weise entsteht ein eindrucksvoller Effekt, der besonders Veränderungen in der Geschwindigkeit und Beschleunigung wieder gibt. Die Deformation sieht bei primitiven Objekten glaubhaft aus, kann aber bei komplexeren Modellen evtl. zu unerwünschten Deformationen führen. Diese Deformationen könnten ggf. vermieden werden, wenn weitere separate Streckungen auf Teile des Modells angewandt würden [HHD04, Han04, S. 71f]. Außerdem muss bei einer Physiksimulation die Kollision angepasst werden.

3.3.2. Bewegung durch Konturen

Auch Konturen können Bewegung suggerieren, indem sie die vergangenen Positionen von Objekten markieren. Sie können aus Silhouetten oder der bildlichen Repräsentation des Objekts selber bestehen. Sich wiederholende Konturen geben besonders bei hohen Geschwindigkeiten dem Betrachter die Möglichkeit das Objekt länger wahrzunehmen und zu verfolgen [Han04, S. 74].

Masuch et al. [MSS99] haben ein System entwickelt, welches aus Animationsdatensätzen Bilder rendert und aus ihnen die Konturen und die z-Reihenfolge ermittelt. Da es sich um ein System für Animationen handelt, ist schon der komplette Ablauf der Bewegung bekannt. Aus diesen Daten und der Bewegungsrichtung des Objekts werden Konturen angefertigt, die immer in bestimmten Zeitabständen gesetzt werden. Falls es erforderlich ist, können auch nur Teilkonturen angefertigt werden, die sich an der Bewegungsrichtung orientieren.

Ein echtzeitfähiger Ansatz wird von Hanl [Han04] vorgestellt, bei dem das Objekt in festgelegten zeitlichen Abständen in eine Textur gerendert wird. Diese Textur wird mit der Szene vereint. Damit Konturen mit fortschreitender Zeit verschwinden, wird die Konturtextur bei jeder Aktualisierung um einen Faktor abgeschwächt. Außerdem wird die Möglichkeit angesprochen Konturen weiter zu stilisieren, indem die Textur mit den Konturen mit der eines Musters multipliziert wird, was aber in seiner Arbeit nicht zu Ende geführt werden konnte.

Hanls Verfahren könnte mit bestehenden Silhouettenverfahren kombiniert werden, indem nur die Silhouetten in die Konturtextur einfließen.

3.3.3. Speed Lines

Speed Lines deuten den Bewegungspfad eines Objekts an. Dabei werden von Künstlern unterschiedliche Stilisierungen der Speed Lines verwendet.

Zunächst wird auf die Eigenschaften bzw. Anforderungen von Speed Lines (Bewegungslinien, Action Lines bzw. Motion Lines) eingegangen. Danach werden mehrere Verfahren in Unterunterabschnitt 3.3.3.2 für Speed Lines vorgestellt, die vornehmlich auf Animationsdatensätzen aufbauen und daher nicht an die Rendergeschwindkeit gebunden sind und Bewegungsabläufe bei Bedarf feiner abtasten können. Ein echtzeitfähiger Ansatz wird in Unterunterabschnitt 3.3.3.3 behandelt und zeigt Ansätze zum Generieren von Speed Lines. Zum Schluss wird ein Hybrid-Verfahren vorgestellt, dass nur zweidimensional arbeitet.

3.3.3.1. Eigenschaften und Anforderungen

Masuch et al. [MSS99] haben mehrere Eigenschaften für Speed Lines und Konturlinien definiert. Eine davon ist, dass Speed Lines entgegensetzt zur Bewegungsrichtung entstehen. Wei-

tere sind ebenfalls als Anforderungen **A1**, **A2** und **A4** von Schulz et al. gestellt worden. Desweiteren sind nach Masuch et al. Speed Lines ungeeignet für Bewegungen zum Betrachter hin.

Schulz et al. [Sch99, HHD04, Han04, S. 24] hat folgende Anforderungen für Speed Lines festgelegt, die nach Möglichkeit in einzelnen Bildern gegeben sein müssen:

(A1) Die in Relation zur Bewegungsrichtung oberste und unterste Bewegungslinie sollen auf gleicher Höhe liegen, wie die maximale Ausdehnung des dargestellten Objekts.

(A2) Bewegungslinien sind möglichst an markanten Bereichen eines Objekts anzusetzen.

(A3) Die Anzahl der Bewegungslinien muss kontrollierbar sein.

(A4) Der Abstand der Bewegungslinien untereinander darf nicht zu gleichmäßig sein.

(A5) Häufungen von Bewegungslinien an bestimmten Bereichen des Objekts müssen vermieden werden.

(A6) Die Bewegungslinien dürfen erst nach dem Objekt beginnen und es nicht schneiden.

(A7) Die Länge der Bewegungslinien muss konfigurierbar sein.

Die aufgezählten Eigenschaften bzw. gestellten Anforderungen geben ein Gefühl für die Aspekte, die bei der Implementierung beachtet werden sollten.

3.3.3.2. Verfahren für gängige Animationsprogramme

Masuch et al. [MSS99] verwenden als Ausgangsbasis zum Generieren von Speed Lines Konturlinien (Silhouetten) eines dreidimensionalen Polygonmodells, welches als Animationsdatensatz vorliegt. Der Datensatz enthält zusätzlich Keyframe-Informationen. Das Verfahren wird als ein Post-Processing Schritt durchgeführt und verwendet Tiefeninformationen der Objekte, um Speed Lines visuell und perspektivisch korrekt darzustellen. Anhand der Keyframes wird der Bewegungspfad aller Objekte bestimmt. Bei der Generierung von Speed Lines halten sie sich an die von ihnen definierten Eigenschaften, die in etwa den Anforderungen aus Unterunterabschnitt 3.3.3.1 entsprechen.

Um Speed Lines zu erhalten, werden alle Vertices als Startpunkte ausgewählt, die zu dem projizierten Objektumriss gehören. Der Objektumriss wird in einzelne Segmente unterteilt, denen jeweils ein Speed Line zugeordnet werden kann. Gibt es keinen Vertex vom projizierten Objektumriss, wird ein zusätzlicher Vertex erstellt. Das Verfahren muss nicht auf äußere Silhouetten beschränkt werden, sondern kann auch innere Silhouetten verwenden. Alle Speed Lines werden mit einem Offset zum Startpunkt gezeichnet. Als Ergebnis werden skalierbare Vektorgraphiken generiert, s. Abbildung 3.3.1. Es ist unklar, ob das Verfahren nur zum Erstellen eines Bildes verwendet werden soll oder auch für Animationen geeignet ist.

Zwei weitere Ansätze für Speed Lines werden von Song [Son05] vorgestellt. Der erste Ansatz geht auf das Erzeugen von Bewegungslinien für Kamerafahren ein, bei denen die Kamera rückwärts fährt. Dafür liegen die Startpunkte regelmäßig auf einen Kreis um die Kamera verteilt,

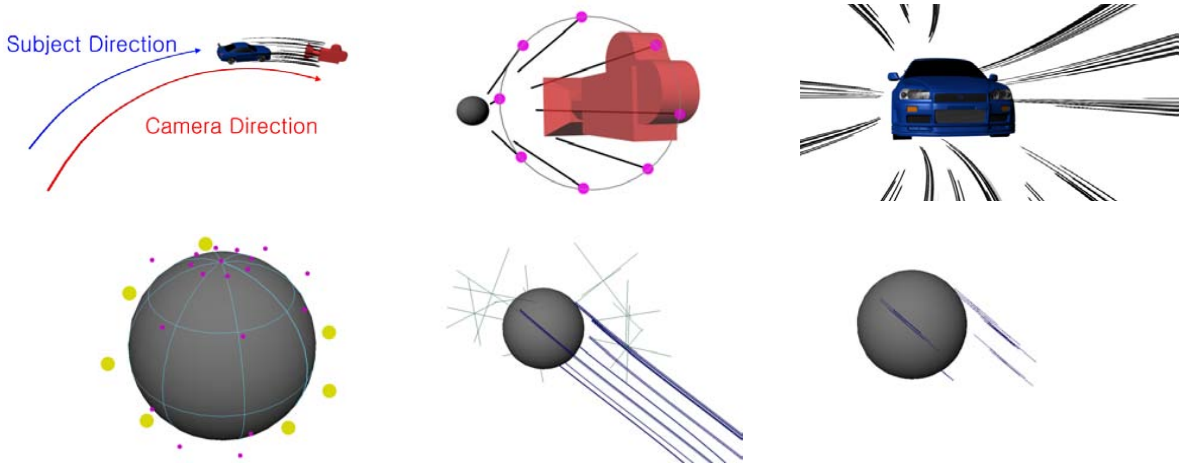

Abbildung 3.3.3.: *oben links nach rechts*: Kamerafahrt mit Blick nach hinten auf ein Auto; *mitte*: schematisch, Kamera und Anfangspunkte der Speed Lines sind zu sehen; *rechts*: Ergebnisbild.
unten von links nach rechts: Auswahl der Punkte zur Speed Line Generierung; kompletter Bewegungspfad aller ausgewählten Punkte; Ergebnisbild. [Son05]

dessen Kreisebene sich parallel zu Bildebene befindet. Die erzeugten Speed Lines hängen von der Geschwindigkeit, dem Bewegungspfad der Kamera und der eingestellten Länge ab, die in Frames angegeben wird. Außerdem werden zusätzliche Startpunkte in unmittelbarer Nähe zu den schon gewählten Punkten benutzt um weitere Speed Lines zu erzeugen. Es entstehen Speed Line Gruppen, die ein weiteres Stilelement für Speed Lines bilden. Alle erzeugten Speed Lines erhalten einen Wahrscheinlichkeitswert, mit dessen Hilfe pro Bild festgestellt wird, ob der entsprechende Speed Line zu sehen ist.

Für bewegte Objekte werden zuerst die Startpunkte auf dem Modell festgelegt. Dabei ist die Anzahl von Punkten definierbar, aber auf die Ermittlung der Punkte geht Song nicht ein. Für die gewählten Punkte wird im ersten Durchlauf der Bewegungspfad aufgezeichnet, um dann im darauffolgenden Durchlauf die Speed Lines und ihre Gruppen zu generieren. Sie erhalten, wie vorher angesprochen, einen Wahrscheinlichkeitswert. Die Ergebnisse seiner Arbeit sind in Abbildung 3.3.3 zu sehen. Zum Schluss wird die Szene abermals gerendert, wobei jetzt alle nötigen Informationen vorliegen um die Speed Lines zu generieren.

Ein Verfahren von Schmid et al. [SSBG10] baut ebenfalls auf Animationsdatensätzen auf. Sie erweitern das Konzept von *Surface-Shadern* zu *Motion Effect Programs*. Anstatt nur das Wissen über einen kleinen Bildausschnitt eines Bildpunkt zu einem Zeitpunkt zur Verfügung zu haben, besitzen die *Motion Effect Programs* Wissen bezüglich der gesamten Szene über die komplette Zeitspanne der Animation. Zusammen mit Objektdaten, Texturen mit Szenen-Informationen, dem Kamera-Sichtvektor und Bewegungsvektoren können diese Programme bei Bedarf einzelne Surface-Shader ausführen, wie es gerade erforderlich ist. Um auf Geometrie von Objekten über eine Zeitspanne zugreifen zu können, werden *Time Aggregate Objects* (TAOs) für bewegte

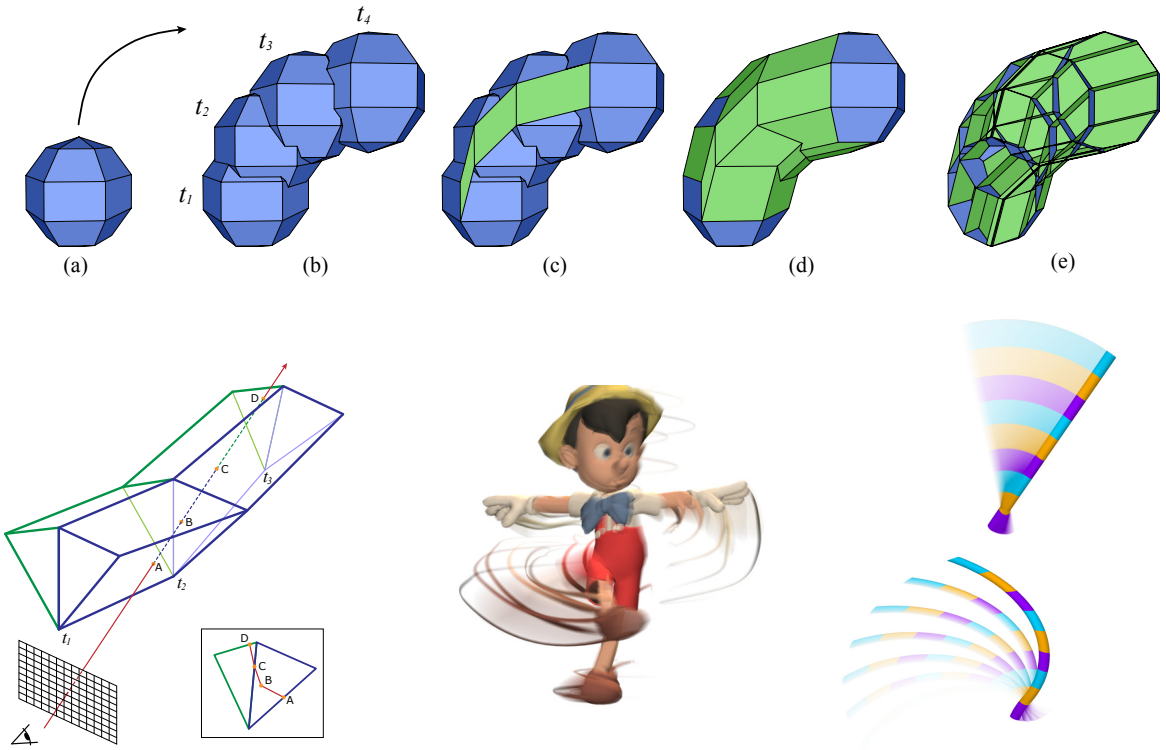

Abbildung 3.3.4.: *oben*: Ein Beispiel, wie eine TAO erstellt wird. *e*) zeigt zusätzlich die innere Struktur.
unten von links nach recht: Generierung von Traces; Ein auf der Stelle drehender Pinocchio; Schwingender Stab, Motion Blur (*oben*) und Konturen mit zeitlicher Verschiebung (*unten*) [SSBG10].

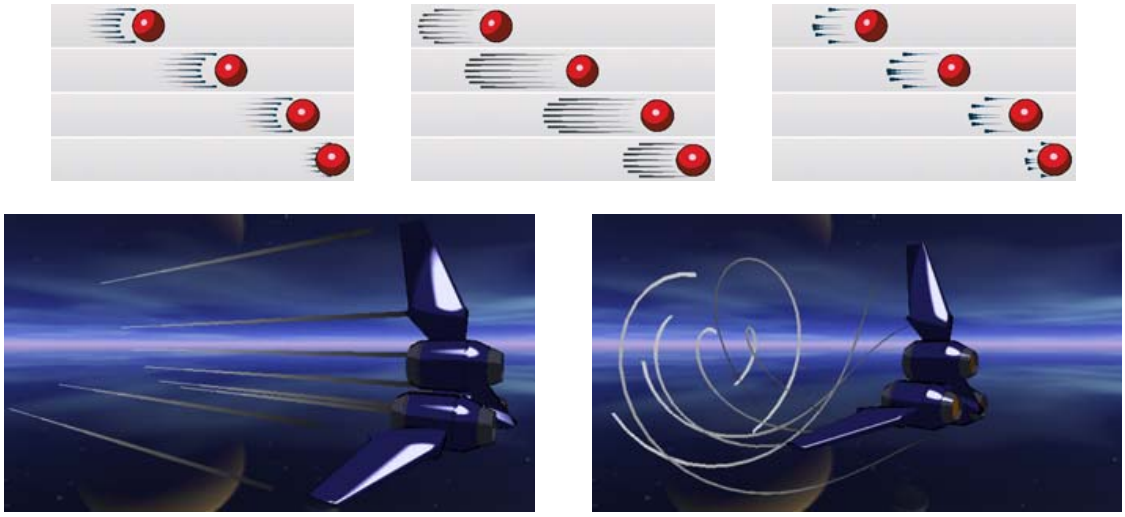

Abbildung 3.3.5.: *oben*: Verschiedene Arten von Bewegungslinien für unterschiedliche Geschwindigkeiten: ausblendend (*links*), am Ende betont (*mitte*) und stark am Ende betont (*rechts*).
unten: Bewegungslinien an einem komplexeren Objekt: geradlinige (*mitte*), gebogen (*rechts*). [Han04].

Objekte erstellt. Sie enthalten die verschmolzene Geometrie für eine Menge von Zeitpunkten, wobei Kanten über die Zeit Polygone aufspannen. Der Begriff *Trace* fasst eine Linie auf dem gesehenen Objekt und dessen gesehene Zeitspanne zusammen. Wie ein TAO erstellt und wie ein Trace auf ihnen ermittelt wird, ist in Abbildung 3.3.4 angedeutet.

Durch die Kombination von Motion Effect Programs, TAOs und Traces können eine Vielzahl von Effekten realisiert werden, weil noch lange nach einem Zeitpunkt Informationen über gesehen Objekte vorhanden sind. Mit diesem Wissen kann entschieden werden, welche Informationen über den Trace aus dem TAO abgefragt werden und ob ein Surface-Shader aufgerufen werden muss.

Auf diese Weise werden nicht nur Speed Lines implementierbar, sondern auch weitere Effekte wie z.B. Motion Blur, nachziehende Konturen und zeitliche Verschiebung der Geometrie. Einige Beispiele sind in Abbildung 3.3.4 gezeigt.

3.3.3.3. Echtzeitfähige Verfahren

Hanl et al. [Han04, HHD04] konzentrieren sich auf ein echtzeitfähiges Verfahren, welches auf Animationsdatensätzen beruht. Ihr System verwendet das Datenformat von Half-Life.

Sie führen Partikelemitter ein, die an markante Vertices im Modell gesetzt werden, wobei die Anforderungen aus Schulz et al. [Sch99] beachtet werden. Jeder Partikelemitter steht dabei für ein Speed Line. Bei jedem Renderdurchlauf setzen die Emitter Partikel frei, die dann zum jeweiligen Speed Line hinzugefügt werden. Ein Partikel ist dabei nichts anderes als die aktuelle

Position des Emitters. Außerdem erhält jedes Partikel noch einen Zeitstempel, der damit dem Partikel eine Lebenszeit gibt. Wird diese Lebenszeit überschritten, wird der Partikel aus dem Speed Line entfernt.

Um markante Vertices für Emitter zu finden, wird in äußere und innere Ansatzpunkte unterschieden. Äußere Ansatzpunkte sind die am weitesten oben und unten liegenden Punkte p_t und p_b eines Modells. Sie müssen senkrecht zur Bewegung die äußersten Punkte im Modell sein und diese Eigenschaften in der vom Betrachter aus projizierten Ebene erfüllen. Je nachdem welche Art von Bewegung vorliegt, gibt es zwei Vorgehensweisen um innere Ansatzpunkte zu erhalten. Bei einer reinen Rotation des Objekts werden auf der Strecke $\overline{p_t p_b}$ Punkte in regelmäßigen Abständen berechnet und um einen geringen Zufallswert verschoben. Die gewählten Punkte bewirken jedoch, dass Speed Lines im Objekt selber starten, was **A6** missachtet. Der zweite Ansatz wird bei normalen Fortbewegungen angewandt und unterteilt das Modell entlang der Bewegungsrichtung in gleichgroße Streifen, um dann jeweils die am weitesten von der Bewegung abgewandten Vertices des Modells auszuwählen.

Die Anzahl der Speed Lines kann durch die Anzahl der Streifen bzw. der ermittelten Punkte auf $\overline{p_t p_b}$ bestimmt werden - abzüglich den zwei äußeren Speed Lines. Die unterschiedlichen Vorgehensweisen zur Ermittlung der Ansatzpunkte beruhen auf unterschiedlichen Arten von Bewegung. Bei Rotationen auf der Stelle sind Ansatzpunkte über Vertices des Modell nicht geeignet, weswegen die Ansatzpunkte zwischen $\overline{p_t p_b}$ gewählt werden. Ansonsten wird das andere Verfahren angewandt. Emitter sind immer auf bestimmte Vertices festgelegt, was auch für Emitter an äußeren Ansatzpunkten gilt. Unter Berücksichtigung der Bedingung **A2** müssten diese Emitter ihre Vertices wechseln, aber dadurch kommt es bei rotierenden Objekten zu springenden Speed Lines.

Die so erzeugten Speed Lines werden als zusätzliche Geometrie gerendert, wobei bei jedem Renderdurchlauf erneut die Attribute jedes Partikel auf Basis des Zeitstempels berechnet werden. Attribute von Partikel bestimmen das Aussehen von Speed Lines, denen sie zugeordnet sind. Mehrere Beispiele für Speed Line Stile sind in Abbildung 3.3.5 zu sehen. In der unteren Hälfte der Abbildung sind zwei weitere Beispiele mit einem etwas komplexeren Objekt zu sehen.

3.3.4. Hybrid-Verfahren

Erwähnenswert ist ein weiteres Verfahren von Kawagishi et al. [KHK03, OKKI05], das ein Hybrid aus Speed Lines und deformierten Objekten darstellt. Außerdem gehört zwei weiteren Speed Line- und Kontur-Verfahren dazu. Alle drei Verfahren zusammen werden Cartoon Blur genannt. Cartoon Blur baut ebenfalls auf vorliegenden Animationsdatensätzen auf und wird im zweidimensionalen Raum angewandt. Um das Objekt zu deformieren, werden der Bewegungspfad und die zur Bewegung abgewandten Seiten ausfindig gemacht, wobei Letztere in mehrere Segmente unterteilt wird. Jeder Vertex der abgewandten Seite wird abwechselnd ge-

Abbildung 3.3.6.: Cartoon Blur von Kawagishi et al. [KHK03]. Es sind Deformationen an einer Axt (*links*) und eines fahrenden Auto (*rechts*) zu sehen.

Anforderungen	Masuch et al.	Song	Schmid et al.	Hanl
Beachtet minimale und maximale Ausdehnung (**A1**)	✓		✓	✓
Verwendet markante Stellen (**A2**)	✓	(✓)	✓	✓
Anzahl einstellbare (**A3**)	✓	✓	✓	✓
Keine konstanten Abstände (**A4**)	✓	✓	✓	✓
Keine Ansammlung auf bestimmten Regionen (**A5**)	✓	(✓)	✓	✓
Hinter Objekt / keine Überschneidungen (**A6**)	✓		✓	(✓)
Länge einstellbare (**A7**)	✓	✓	✓	✓

Tabelle 3.1.: Gegenüberstellung der genannten Verfahren anhand der Anforderungen aus Unterunterabschnitt 3.3.3.1.
✓ =voll erfüllt; (✓) =zum Teil erfüllt bzw. erwecken den Anschein die Anforderungen erfüllen zu können.

ringfügig oder stark entlang des Bewegungspfads verschoben. Einen Eindruck bezüglich der Ergebnisse dieses Verfahrens gibt Abbildung 3.3.6.

3.4. Bewertung der Verfahren für Speed Lines

Wie in Abschnitt 3.3 sind mehrere Verfahren vorgestellt worden, um Verformungen, Konturen und Speed Lines beim Rendern von Szenen zu realisieren. Diese Arbeit hat das Ziel Bewegungsunschärfe mittels NPR-Verfahren umzusetzen. Der Effekt, der erzielt werden soll, wurde von Scott McCloud als Photographische Schlieren bezeichnet und verwendet dafür Speed Lines, vgl. Unterabschnitt 2.2.3. Desweiteren soll das Verfahren in Echtzeit ausgeführt werden können.

Nach den Anforderungen aus Unterunterabschnitt 3.3.3.1 ist die Tabelle 3.1 erstellt worden, in der die vorgestellten Verfahren zu Speed Lines gegenüber gestellt werden. Sie beruht auf den Ergebnissen der vorliegenden Artikel. Es zeigt sich, dass alle Verfahren mit Ausnahme des von Song allen Anforderungen gerecht werden. Beim Verfahren von Song ist nicht bekannt, wie die

Eigenschaften	Masuch et al.	Song	Schmid et al.	Hanl
Echtzeitfähig	(✓)			✓
Stilisierbar			✓	gering
Verwendet Geometrie	✓ (Konturen)	✓	✓ (TAO)	✓
Unstetige Generierung von Speed Lines möglich	(✓)	✓	(✓)	
Abbildung von nicht-linearen Bewegungen	✓	✓	✓	✓
Speed Lines über mehrere Frames	✓	✓	✓	✓

Tabelle 3.2.: Gegenüberstellung weiterer Eigenschaften der vorgestellten Verfahren.

Ansatzpunkte ausgewählt werden (**A2**) und die Speed Lines Gruppen sind genau genommen eine Ansammlung von Speed Lines (**A5**). Wenn Hanl bei einer Rotation die Ansatzpunkte entlang $\overline{p_t p_b}$ auswählt, beginnen die Linien nicht hinter dem Objekt, sondern starten in ihm selber (**A6**).

In Tabelle 3.2 sind die Verfahren anhand einiger weiterer Eigenschaften gegenübergestellt worden. Dabei war es wichtig, weitere Aspekte zu analysieren. Eigenschaften wie Echtzeitfähigkeit und Stilisierbarkeit sind besonders in Spielen Gesichtpunkte, denen eine besonders große Rolle zukommt. Dazu kommen weitere Punkte, die in die Tabelle eingegangen sind, wie: Basiert das Verfahren auf Wissen über die Geometrie? Können Speed Lines über die Zeit gesehen zufällig generiert werden? Ist die Abbildung von nicht-linearen Bewegungen möglich? Und: Sollen Speed Lines die Bewegung über mehrere Frames hinweg beschreiben? Mit Abbildung von nicht-linearen Bewegung ist gemeint, dass Speed Lines sich auch verbiegen können und nicht nur gerade Striche sind.

Die vorgestellten Verfahren zu Speed Lines sind - mit Ausnahme des Verfahrens von Hanl - nicht echtzeitfähig, zeigen aber verschiedenste interessante Ansätze zur Realisierung von Speed Lines. Stilisierung wird in geringem Maß von Hanl behandelt, aber das Vorgehen nicht näher erläutert. Schmid et al. bieten viele Möglichkeiten Stilisierung einzuführen, aber gehen ebenfalls nicht genauer darauf ein. Alle anderen Verfahren zeichnen nur einfache schwarze Linien. Jedes Verfahren basiert sehr stark auf der Verwendung von Animationsdatensätzen und verwendet Geometrie auf der CPU-Seite um Speed Lines zu erzeugen. Song und Hanl verwenden sehr ähnliche Ansätze für die Emitter, unterscheiden sich aber bei der Auswahl geeigneter Startpunkte bzw. Ansatzpunkte. Das Verfahren von Song besitzt als einziges die Möglichkeit Speed Lines zufällig zu wählen. In den anderen Gesichtspunkten unterscheiden sich die Verfahren sehr geringfügig. Jedes Verfahren ist in der Lage komplexe Bewegungsverläufe in einem gerenderten Bild auszudrücken und mit Bewegungslinien den Bewegungsverlauf über mehrere gerenderte Bilder hinweg anzudeuten.

3.5. Zusammenfassung

Nur Hanls Verfahren schafft es, Speed Lines in Echtzeit zu generieren. Außerdem basieren alle Verfahren auf der Verarbeitung von Animationsdatensätzen und Geometrie. In keinem vorgestellten Verfahren konnten Photographischen Schlieren, wie sie Simmons und McCloud beschrieben haben, entdeckt werden.

4. Motion Blur

Wenn sich ein Objekt sehr schnell bewegt, wird es verschwommen wahrgenommen. Es entsteht der Eindruck, es befände sich an mehreren Orten gleichzeitig. Dies wird als *Bewegungsunschärfe* (Motion Blur) bezeichnet.

Dieser Eindruck kann mit einer starren Kamera mit einer entsprechenden Belichtungszeit beobachtet werden. Wenn die Kamera das Objekt verfolgt, kann umgekehrt beobachtet werden, dass schnelle Objekte scharf und deren Umgebung verschwommen dargestellt werden. Bewegungsunschärfe ist ein Wahrnehmungseffekt, der vom momentanen Standpunkt des Betrachter und dessen Bewegung abhängig ist. Weiterhin ist zu beobachten, dass Objekte mit gleicher Geschwindigkeit aus der Nähe betrachtet unschärfer sind, als weiter entfernte Objekte.

Motion Blur ist ein Begriff der Computergraphik. Auf Grund der Rasterisierung einer darzustellenden Szene wird die Bildebene in kleine Rechtecke unterteilt, für die stellvertretend nur ein Oberflächenpunkt des vordersten Objekts betrachtet wird. Dieses Vorgehen führt dazu, dass sich üblicherweise ein Treppeneffekt an Objektkanten ausbildet, der *Aliasing* genannt wird. *Anti-Aliasing* glättet diese scharfen Kanten durch eine höhere Auflösung oder mehrere leicht verschobene Abtastpunkte.

In der Zeitdomäne treten ebenfalls Aliasing-Effekte auf, weil für eine Zeitspanne nur ein Zeitpunkt gerendert wird. Schnelle Bewegungen können kaum wahrgenommen werden. Eine höhere zeitliche Auflösung bzw. mehrere Abtastungen über die Zeit können helfen, diesen temporalen Aliasing-Effekt zu unterdrücken. Motion Blur entspricht temporalem Anti-Aliasing und bewirkt flüssigere Darstellungen bei weniger Frames Per Second (FPS)[1] [AMHH08, S. 490]. Die Computergraphik versucht genau diesen Effekt nachzustellen. Dies führt meistens zu mehr Rechenaufwand, den Echtzeitanwendungen nicht immer leisten können. Es gibt mehrere Verfahren zum Erzeugen von Motion Blur:

- Annäherung durch den Accumulation Buffer
  - Standard Verfahren (n-Pass Verfahren)
  - Verfahren mittels mehrstufigem FBO[2] (2-Pass Verfahren)
  - Echtzeitfähige Verfahren (1 oder 2-Pass Verfahren)

[1] (FPS) ist eine Geschwindigkeitsangabe und beutet: Bilder pro Sekunde.

[2] Frame Buffer Objekte ermöglichen es für einen Rendervorgang mehrere Buffer zu erstellen, in die die Szene hinein gerendert wird.

- Stochastische Annäherung
- Motion Blur mittels Erweiterung der Geometrie
- Motion Blur als Post-Processing Verfahren
  - Geschwindigkeit der Kamera
  - Geschwindigkeit pro Objekt
  - Geschwindigkeit pro Pixel

Um eine bessere Vorstellung über Bewegungsunschärfe zu erlangen, wird in Abschnitt 4.1 auf deren Entstehung eingegangen und die damit verbundenen Einflussgrößen herausgearbeitet. Die darauf folgenden Abschnitte behandeln Verfahren zu Motion Blur, wobei zuerst auf das Standardverfahren mittels Accumulation Buffer in Abschnitt 4.2 eingegangen wird. Es wird kurz auf Motion Blur durch echtzeitfähiges stochastisches Rendern und durch Erweiterung der Geometrie in Abschnitt 4.3 und Abschnitt 4.4 eingegangen. Danach werden Post-Processing Verfahren unter Verwendung eines Geschwindigkeitsvektors und der Line Integral Convolution in Abschnitt 4.5 vorgestellt und ihre Schwierigkeiten beleuchtet. Zum Schluss werden in Abschnitt 4.6 alle Verfahren mit ihren Stärken und Schwächen gegenübergestellt und anhand ihrer Echtzeitfähigkeit und Ergebnisse bewertet.

4.1. Entstehung und Einflussgrößen

Um den Effekt der Bewegungsunschärfe genauer beschrieben zu können, muss geklärt werden, von welchen Einflussgrößen die Entstehung der Bewegungsunschärfe abhängt. Einflussgrößen, die direkt ins Auge fallen, sind die Aufnahmedauer sowie die Größe und Geschwindigkeit eines Objekts. Von Bedeutung ist außerdem die Bewegung der Kamera.

Das mathematische Kameramodell [Shi05, S. 166f] beinhaltet eine Bildebene, auf die die betrachtete Wirklichkeit projiziert wird und die schlussendlich das Sichtfenster ist, durch welches die Aufnahme wahrgenommen wird. Die Größe der Bildebene ist fest, verändert sich normalerweise nicht und wird in der Computergraphik in Pixeln gemessen.

Jeder sichtbare Gegenstand wird auf diese Ebene projiziert. Durch die Projektion fließt die Entfernung zur Kamera in die Entstehung eines Bildes mit ein. Bewegt sich der projizierte Gegenstand während der Aufnahme um eine gewisse Entfernung, ist es entscheidend, wie groß die projizierte Strecke in der Bildebene ist. Ist sie geringer als die Größe eines Pixels, wird sie wahrscheinlich nicht registriert und das Objekt wird scharf wahrgenommen. Ist sie größer als ein Pixel, beginnt das Objekt entlang seines Bewegungspfads zu verschwimmen. Dieses Phänomen äußert sich anfangs nur durch ein unscharfes Objekt, kann jedoch bis zum vollständigen Verschwinden des Objekts führen.

Damit bestimmt allein der zurückgelegte Weg in der Bildebene während der Aufnahme, wie stark das beobachte Objekt verschwimmt. Dieser Umstand ermöglicht es, Bewegungsunschärfe allein aufgrund von Geschwindigkeitsvektoren in der Bildebene zu beschreiben, vgl. Unterabschnitt 4.5.4.

Bei den oben angeführten Annahmen wurde die Bewegung der Kamera nicht als Einflussgröße berücksichtigt. Bei einem Szenario mit sich bewegender Kamera gelten die vorherigen Annahmen dennoch. Innerhalb der Zeitspanne ist nur entscheidend, wie lang die zurückgelegte Entfernung in der Bildebene ist. Die zurückgelegte Entfernung ist schließlich das Resultat der Projektion und der Differenz zur Kamerabewegung.

Die Intensität der Bewegungsunschärfe kann damit über die zurückgelegte Entfernung in der Bildebene bestimmt werden und wird linear durch einen Bewegungsvektor angenähert. Die zurückgelegte Entfernung in der Bildebene ist abhängig von:

- Der vergangenen Zeit,
- der eigenen Bewegung,
- der Kamerabewegung und
- der Entfernung zwischen Objekt und Kamera.

4.2. Annäherung durch den Accumulation Buffer

Der Accumulation Buffer ist ein Bildspeicher (Buffer) in OpenGL. Ein Farbwert im Accumulation Buffer besitzt eine größere Bit-Genauigkeit als die üblichen Farbbuffer. Dieser Umstand ermöglicht es, Bilder mit geringerem Informationsverlust zu akkumulieren und zu gewichten.

Anzumerken ist, dass der Accumulation Buffer ab OpenGL 3.0 als *deprecated* markiert ist. Dies bedeutet, dass bei der Verwendung von OpenGL 3.0 ohne Abwärtskompatibilität der Buffer nicht zur Verfügung steht und nachgebaut werden muss.

4.2.1. Standardverfahren

Wie bei dem üblichen Anti-Aliasing Verfahren im Bildraum kann auch in der temporalen Domäne höher abgetastet werden. Dafür werden n Teilbilder zu verschiedenen Zeitpunkten gerendert und über den Accumulation Buffer aufaddiert und durch n geteilt [HA90, AMHH08, S. 492f]. Es handelt sich dadurch um ein n-Pass Verfahren, wobei standardmäßig alle Teilbilder im gleichen Zeitabstand erzeugt werden sollten. Unter der Annahme, dass jeder Renderdurchlauf die gleiche Zeit beansprucht, kann die Szene n-Mal gezeichnet werden, ohne dass der Zeitpunkt explizit festgelegt werden muss.

Anstatt alle Teilbilder gleich zu gewichten, ist es auch vorstellbar das letzte Teilbild am stärksten zu gewichten. Dieses Vorgehen hat auf der einen Seite eine leichte Abschwächung der

Bewegungsunschärfe zur Folge. Auf der anderen Seite tritt das letzte gezeichnete Bild am deutlichsten hervor, so dass das Endresultat auf die Bewegungsrichtung des Objekts schließen lässt.

Für heutige Anwendungen oder Spiele sind ist diese Verfahren nicht anwendbar, weil es zu viel Zeit beansprucht. Für beispielsweise acht Teilbildern für Motion Blur und mindestens 25 Bildern pro Sekunde - weniger zerstört die Illusion von Bewegung - müssen 200 Bilder in der Sekunde gerendert werden. Daraus folgt, dass ein Renderdurchlauf nicht mehr als 0,005 Sekunden beanspruchen darf. Diese Anzahl an Bildern pro Sekunde reich nicht aus.

4.2.2. Verfahren mittels mehrstufigen FBOs

Anstatt eine Szene n-Mal zu rendern, kann Motion Blur auch durch ein 2-Pass Verfahren erreicht werden. Damit diese Technik angewendet werden kann, muss das Shader Modell 4.0 unterstützt werden. Beim Rendern wird die Szene in ein mehrstufiges Frame Buffer Objekt gezeichnet. Ein mehrstufiges Frame Buffer Objekt (FBO) verwendet statt zweidimensionale Buffer dreidimensionale Buffer. An diese Art von FBOs können nur 3D Texturen oder 2D Textur Arrays angehangen werden, wobei jede Ebene im FBO einer 2D Textur in der 3D Textur entspricht. Alle Ebenen sind gleich groß und können separat beschrieben werden. Im Geometry-Shader kann entschieden werden, in welchen Layer geschrieben wird. Anzumerken ist, dass das Ausgabeprimitiv nur einer Ebene angehören kann, weil es sonst nicht rasterisiert wird.

Diese Eigenschaft macht sich Patidar et al. [PBN07] zu nutze und schreibt während des ersten Renderdurchlaufs eine Szene in n Ebenen. Dafür erhält der Geometry-Shader ein Array von Matrizen, die jede ModelView-Matrix zum Zeitpunkt t_i für $i = 1, \ldots, n$ enthält, und transformiert die Szenen entsprechend. Im zweiten Renderdurchlauf wird das 2D Textur Array in den Fragment-Shader eingebunden und über eine Texturkoordinate werden die Farbwerte jeder Ebene aufaddiert und gemittelt. Leider macht Patidar keine Angaben über die Echtzeitfähigkeit des Verfahrens.

4.2.3. Echtzeitfähige Verfahren

Da es sehr zeitintensiv ist, mehrere Teilbilder zu erzeugen, muss eine andere Lösung gefunden werden. Idealerweise sollte die Szene nur einmal gerendert werden und trotzdem Motion Blur aufweisen. Die folgenden Abschnitte stellen zwei Verfahren vor, die den Accumulation Buffer verwenden und Motion Blur in Echtzeit darstellen können.

4.2.3.1. Zeitliche Abschwächung der Teilbilder

Wie vorher erwähnt müssen die Bilder im Buffer nicht gleichgewichtet werden. Wird beim Akkumulieren eines neuen Bildes der Buffer und das neue Bild zu gleichen Teilen gewichtet, bedeutet das, dass jedes eingebrachte Bild im Buffer immer mehr an Gewichtung verliert. Am

Anfang ist die Gewichtung erst 50%, danach 25%, 12,5% usw. . Diese Abschwächung zeigt, dass innerhalb kurzer Zeit neu hinzugefügte Bilder im Buffer an Bedeutung verlieren. Wenn auf diese Weise jedes neu gerenderte Teilbild mit dem Buffer kombiniert wird, ist es nicht mehr nötig mehrere Teilbilder pro Renderdurchlauf zu erzeugen.

4.2.3.2. Gleichgewichtung aller Teilbilder

Um trotzdem gleichgewichtete Bilder aus n Teilbilder zu erhalten, muss eine andere Technik angewandt werden. McReynolds et al. rendert dafür die Szene in zwei Renderdurchläufen [MB05, AMHH08, S. 492f]. Beim ersten Rendern wird die Szene, wie sie zum Zeitpunkt t' vor n Rendervorgängen aussah, gerendert. Das entstandene Bild wird vom Accumulation Buffer abgezogen. Danach wird die Szene zum gegenwärtigen Zeitpunkt t dargestellt und auf den Accumulation Buffer aufaddiert. Beim Subtrahieren und Addieren wird das zu akkumulierende Bild jeweils mit dem Faktor $\frac{1}{n}$ multipliziert. Dieses Vorgehen bewirkt, dass jedes Teilbild im Buffer gleichgewichtet ist und die Szene nur zweimal gerendert werden muss. Außerdem setzt dieses Verfahren voraus, dass die Szene zu einem definierbaren Zeitpunkt t_i dargestellt werden kann.

Für den Fall, dass eine zeitliche Kontrolle über die Szene nicht vorhanden ist, können stattdessen die Teilbilder in einem Ringbuffer zwischengespeichert und später vom Accumulation Buffer abgezogen werden. Der Vorteil ist, dass die Szene nur einmal gerendert werden muss. Der Nachteil ist, dass jedes zwischengespeicherte Teilbild zusätzlichen Speicher benötigt.

4.3. Stochastische Annäherung

Akenine-Möller et al. verwenden einen stochastischen Ansatz um Motion Blur zu erzeugen [AMMH07, AMHH08, S. 875f]. Es werden *Time Continuous Triangles* (TCT) verwendet und legen um sie Bounding Box zum Abtasten. Mit einem 2×2 pixelgroßen Feld wird die Bounding Box abgetastet und für jeweils eine Menge von Zeiten t_i überprüft, ob das Feld zum Zeitpunkt t_i das Dreieck schneidet. Wird ein TCT geschnitten, wird in Abhängigkeit t_i die Vertices interpoliert und das Feld an den Fragment-Shader weitergeleitet.

Ergebnisse des Verfahrens sind in Abbildung 4.3.1 zu sehen. Die oberen Bilder zeigen eine auf der Stelle drehendes Rad und die unteren Bilder ein sich fortbewegendes Rad. Von links nach rechts ist zuerst die Annäherung mit dem Accumulation Buffer durch viermaliges Rendern abgebildet, danach ist die stochastische Annäherung zu sehen und Referenzbilder, die 64 mal über den Accumulation Buffer abgetastet wurden.

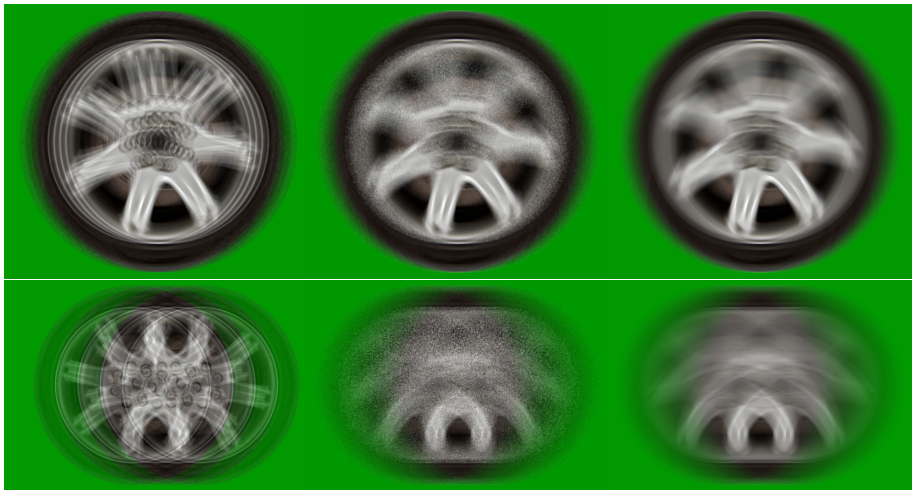

Abbildung 4.3.1.: Beispiele für ein auf der Stelle (*oben*) und ein sich fortbewegendes Rad (*unten*). Der Motion Blur Effekt ist einmal durch den Accumulation Buffer mit 4 Bildern (*linke Spalte*), dem Stochastischen Verfahren und 4 Abtastungen (*mitte*) und mit zufälligen 64 Abtastungen (*rechts*) dargestellt. [AMMH07]

4.4. Motion Blur mittels Erweiterung der Geometrie

Ein Beispiel in der DirectX SDK [AMHH08, Mic07, Example MotionBlur10] erreicht Motion Blur, indem Geometrie entlang der Bewegung vor und hinter dem Objekt generiert wird. Die zusätzliche Geometrie erhält Alphawerte, um sie transparent wirken zu lassen. Zusätzlich werden die verwendeten Texturen des Objekts weichgezeichnet.

Im Geometry-Shader wird entsprechend Geometrie vor und hinter das sich bewegende Objekt generiert und als Triangle Strip weitergereicht. Dazu müssen entsprechende Matrizen in den Shader eingebunden werden. Die Matrizen repräsentieren die verschiedenen Objekttransformationen zum Zeitpunkt t_i. Jedes zusätzlich erzeugte Dreieck erhält einen Alphawert, der eine Ausblendung bewirken soll. Die verwendeten Texturen werden weichgezeichnet, um die Bewegungsunschärfe zu verstärken. Dafür wird die Textur mit einem Geschwindigkeit im Texturkoordinatensystem anisotropisches gefiltert [Wik10, AMHH08, S. 168f]. Da die zusätzliche Geometrie transparent ist, wird das Alpha To Coverage Verfahren angewandt. Es handelt sich um eine vereinfachte Form von Order-Independent Transparency [Mic07, s. Instancing10].

4.5. Motion Blur als Post-Processing Schritt

Shader können zur Berechnung von Motion Blur herangezogen werden. Verfahren für Shader arbeiten dabei mit der momentanen Geschwindigkeit eines Oberflächenpunkts, die vorberechnet und in einem Velocity Buffer abgespeichert oder aus dem vorherigen und jetzigen Tiefenbuffer zurückgerechnet werden kann. Der Abschnitt stellt drei Methoden vorgestellt zur Geschwindigkeitsberechnung.

4.5.1. Geschwindigkeit der Kamera

Das folgende Vorgehen berechnet die Bewegung der Kamera über die Rückprojektion der Tiefe [Ros07, S. 575-581]. Um einen Geschwindigkeitsvektor $\vec{v}$ zu erhalten, wird über die Position p^v im Viewport und der Tiefe t die Position in Weltkoordinaten zurückgerechnet. Durch die Gleichung

$$p^w = M^{-1} \cdot p \quad \text{mit } p = \left(p_x^v \; p_y^v \; t\right)^T \text{ und } M = V \cdot P \cdot C$$

lässt sich der Punkt in Weltkoordinaten berechnen, wobei M das Produkt der Matrizen für den Viewport (V), die Projektion (P) und die Kamera (C) ist. Indem über die Matrix M' die vorherige Kameraposition bekannt ist, kann der Punkt p^w wieder in Viewportkoordinaten $p^{v\prime}$ umgerechnet werden. Die Differenz von p^v und $p^{v\prime}$ ergibt die gesuchte Geschwindigkeit $\vec{v}$.

4.5.2. Geschwindigkeit pro Objekt

Anstatt nur die Geschwindigkeit der Kamera zu ermitteln, können solide Körper durch einen Bewegungsvektor $\vec{v}$ beschrieben werden [Pad09, RMM10]. Dafür wird die gesamte Szene mit einem Objekt ID Buffer gerendert, wobei jedes Objekt eine eigene ID hat, welche zu einer Matrix M_i korrespondiert. Diese Matrizen M_i beschreiben die Bewegung eines Objekts zum vorherigen Zeitpunkt. Wie in Unterabschnitt 4.5.1 wird $p^{v\prime}$ berechnet und $\vec{v}$ aus der Differenz zu p^v gebildet.

4.5.3. Geschwindigkeit pro Pixel

Ein Velocity Buffer speichert die momentane Geschwindigkeit eines Pixels ab. Um die Geschwindigkeit zu ermitteln, übergeben Shimizu et al. die momentane Modelview Matrix M und die vorherige Modelview Matrix M' an den Vertex-Shader [SSC03, Gre03, AMHH08, S. 492f]. Mit Hilfe dieser beiden Matrizen wird jede Position p eines Vertex in Weltkoordinaten p^w und p'^w transformiert. Die Differenz der beiden Punkte ergibt die Geschwindigkeit $\vec{v}$ als Vektor. Diese Berechnungen werden in OpenGL im Vertex-Shader durchgeführt und im Fragment-Shader in einer Textur gespeichert.

Beispielcode ist in Listing 7.1 zu sehen.

4.5.4. Post-Processing mit der ermittelten Geschwindigkeit

Alle drei vorgestellten Methoden zur Ermittlung können im folgenden Post-Processing Schritt angewandt werden, aber aus der Geschwindigkeit alleine ergibt sich noch keine Bewegungsunschärfe. Dafür wird eine *Line Integral Convolution* (LIC) benötigt [CL93, HWSE99, AMHH08].

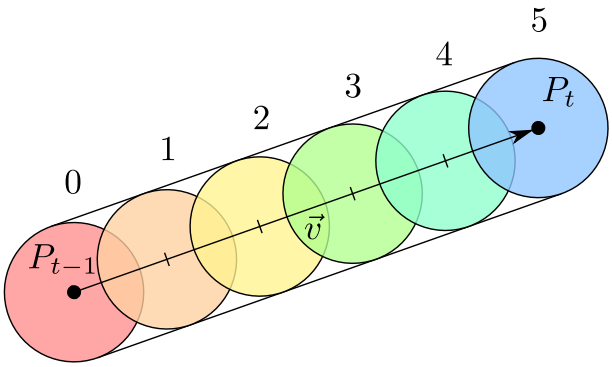

Abbildung 4.5.1.: Abtastung von sechs Farbwerten entlang eines Vektors.

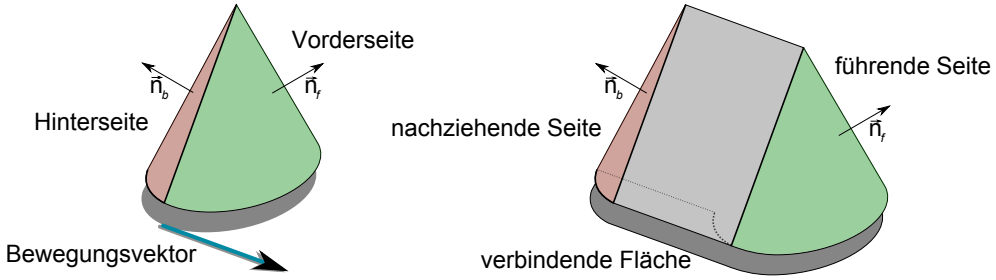

Abbildung 4.5.2.: Streckung der Geometrie eines Kegels entlang seiner Bewegungsrichtung.

Die LIC tastet die einzelnen Farbwerte entlang einer Linie ab und mittelt diese. Die Orientierung und Länge der Linie wird durch den projizierten Geschwindigkeitsvektor festgelegt, wobei der Vektor zunächst in den Texturraum transformiert wird. Abbildung 4.5.1 zeigt die Abtastung von sechs Farbwerten entlang eines Vektors $\vec{v}$ zwischen einem Punkt p_{t-1} und p_t. Damit wird das Objekt nur innerhalb seiner Objektgrenzen unscharf gezeichnet. Die Abtastung kann in beide Richtungen erfolgen.

Diese Verfahren hat als Endresultat scharfe Objektkanten und wirkt damit unrealistisch. Außerdem werden je nach Bewegung Farbwerte von unbewegten Objekten abgetastet, die nicht zum eigentlichen Objekt gehören.

Physikalisch korrekte Bewegungsunschärfe erstreckt sich nicht nur auf die Objektgrenzen, sondern auch über sie hinaus.

4.5.5. Streckung der Geometrie

Um Motion Blur außerhalb der Objektgrenzen zu erhalten, muss das Objekt gestreckt werden. Wloka et al. strecken in einem extra Renderdurchlauf die Geometrie anhand der Geschwindigkeit $\vec{v}$ und der Oberflächennormalen $\vec{n}$ des Vertex [WZ96, Wlo01, Gre03, AMHH08]. Durch sie kann nun im Fragment-Shader außerhalb der Objektgrenzen die jeweilige Geschwindig-

keit ermittelt und in eine Textur geschrieben werden. Das Ergebnis ist ein Velocity Buffer, der Geschwindigkeiten auch außerhalb der Objektgrenzen darstellt.

Die eigentliche Streckung des Objekts geschieht im Vertex-Shader. Zuerst wird das Skalarprodukt aus $\vec{n}$ und $\vec{v}$ ermittelt. Damit kann festgestellt werden, ob die Normale entlang der Bewegungsrichtung verläuft oder von ihr weg zeigt. Weist sie entlang der Bewegungsrichtung, wird p^w an die Pipeline weitergereicht, sonst p'^w. Auf diese Weise werden Polygone auseinandergezogen, die Normalen entlang und entgegengesetzt zur Bewegung besitzen. Ein Beispiel ist in Abbildung 4.5.2 zu sehen.

Durch die Streckung ist es möglich die Geschwindigkeiten außerhalb der eigentlichen Geometrie festzustellen. Diese Modifikation hat zur Folge, dass um das Objekt herum der Hintergrund verschwimmt.

4.5.6. Probleme mit gestreckter Geometrie

Es gibt eine Reihe von Problemen, die sich durch das Strecken der Geometrie entstehen. Die folgenden Abschnitte sprechen diese Probleme und zeigen evtl. einige mögliche Lösungen auf.

4.5.6.1. Verschwimmender Hintergrund

Ein ungewollter Nebeneffekt ist das Verschwimmen des Hintergrunds mit dem bewegten Objekt. Dieser Effekt kann gelöst werden, indem das Objekt separat vor einem schwarzen Hintergrund in eine Textur gerendert wird [Por07, Oh10]. Die Auswertung des LIC wird dadurch nur auf der Textur durchgeführt und zum Schluss mit der Szene verschmolzen.

4.5.6.2. Zerteilte Geometrie

Das Verfahren von Wloka zur Streckung der Geometrie kann auf fast jede Geometrie angewendet werden, funktioniert aber nicht für jede Geometrie zuverlässig. Das Verfahren basiert auf der Annahme, dass ein Vertex zweier Polygone immer die gleichen Normalen verwendet. Im Fall eines Würfels, dessen Vertices die Normale ihrer zugehörigen Fläche besitzen, können ungewollte Artefakte auftreten. Beim Strecken eines solchen Modells würde sofort auffallen, dass es an den Flächen geteilt wird, deren Normalen nicht mehr in Bewegungsrichtung weisen. In Abbildung 4.5.3 kann dieses Problem anhand eines Würfel beobachtet werden.

Da Vertex- und Geometry-Shader nicht alle adjazenten Vertices kennen, besitzen sie nur ein begrenztes Wissen über die umliegende Geometrie. Deshalb kann dieses Problem nicht in einem Shader gelöst werden. Aus diesem Grund ist es notwendig geeignete Geometrie an die Graphikkarte zu schicken. Eine Lösung ist die Einfügung von Glue Patches, die als verbindende Flächen (Joining Surfaces) genutzt werden. Solange sich das Modell nicht bewegt, ist ein

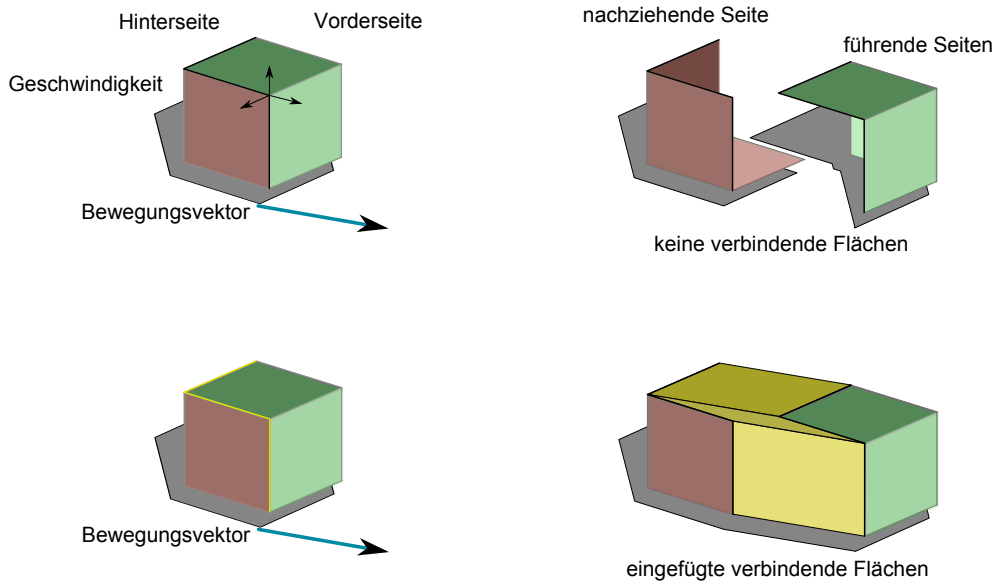

Abbildung 4.5.3.: Ungewollte Zerteilung eines Würfel durch das Verfahren von Wloka.

Glue Patch nichts anderes als eine Linie. Bei Bewegung wiederum entfaltet sich die Fläche, ein Beispiel hierfür ist in Abbildung 4.5.3 gegeben.

Ein Glue Patch wird an jeder Kante eingefügt, an der mindestens ein Vertex mit mehr als einer Normale liegt. Ein Glue Patch besteht somit aus den beiden Vertices und ihren Normalen. Daraus ergibt sich ein viereckiges Patch.

4.5.6.3. Überlagernde Objekte

In einer Szene mit mehreren bewegten Objekten können sich ineinander bewegende Objekte stören, weil Bewegungsvektoren nur für die gestreckte Geometrie des vordersten Objekts berechnet werden. Da diese Geometrie aber nicht immer das sichtbare Objekt repräsentiert, kann es sein, dass verdeckte Objekte unscharf gezeichnet werden bzw. sich anscheinend in die falsche Richtung bewegen. Durch die Hinzunahme der Objekt ID kann dieser Effekt gelindert werden.

Komplett vermeiden lässt sich dieser Effekt nur durch separates Rendern von allen bewegten Objekten in Texturen mit schwarzem Hintergrund. Bei der Vereinigung mit der Szene muss das unscharfe Objekt über die Tiefe der gestreckten Geometrie zusammengeführt werden.

4.5.6.4. Transparenzen

Das Problem mit Transparenzen könnte mit Order-Independent Transparency behoben werden [MB07b, MB07a]. Zum Rendern werden Texturen mit Subpixeln benötigt. Der spätere Zugriff auf die Subpixel muss ebenfalls gewährleistet sein. Dabei wird mit Hilfe des Stencil Buffers nacheinander nur jeweils ein Subpixel für einen Fragment-Shader beschrieben. Nach dem

kompletten Renderdurchlauf können die Subpixel in einem weiteren Renderdurchlauf einzeln betrachtet werden und die Farbwerte mit ihrer gespeicherten Tiefe in die richtige Reihenfolge gebracht werden. Um die Tiefe mit den Farbwerten zusammen zu speichern, muss eine 32-Bit Textur mit zwei Farbkanälen verwendet werden. In den ersten Kanal wird die komplette Farbe kodiert und in den Zweiten wird nur die Tiefe abgelegt.

4.5.7. Weitere Verfahren

Es gibt ein Verfahren von Tatarchuk et al., welches sich speziell mit dem korrekten Farbeindruck bei entstehender Bewegungsunschärfe befasst. Besonders wird auf den Verlust von Glanzeffekten durch übliche Motion Blur Verfahren eingegangen und wie sie dennoch erreicht werden können [TBI03].

4.6. Bewertung der Verfahren

Beim Accumulation Buffer besteht bei den echtzeitfähigen Verfahren das Problem, dass der Betrachter den Eindruck erhält, dass Objekte Schlieren hinter sich herziehen. Dieser Effekt wird nur durch Abschwächung der einzelnen Teilbilder verringert. Der Schliereneffekt wird sogar noch verstärkt, wenn eine schnelle Bewegung vorherrscht und die zeitlichen Abstände zwischen den einzelnen Teilbildern groß sind.

Es gibt eine Menge an echtzeitfähigen Verfahren, die aber jeweils ihre eigenen Probleme mit sich bringen. Stochastisches Rendern liefert gut Ergebnisse, aber bringt Rauschen ins Bild, das vielleicht unerwünscht ist. Bei der Erweiterung der Geometrie müssen weitere spezielle Techniken für die Behandlung von Transparenzen verwendet werden. Beide Verfahren hängen von der Komplexität der Szene ab.

Post-Processing Verfahren, die nur die Kameraposition miteinbeziehen, werden meistens in Rennspielen verwendet. Das Auto wird extra ausmaskiert, damit es kein Ausbluten der Objektfarbe an den Rändern sich ergibt. Die Rückprojektion über die Tiefe für Objekte ist unzureichend für Charaktere, die mehrere Transformationsmatrizen pro Knochen besitzen. Aus dem Grund wirdfür Objekte mit einer Skelettstruktur der Velocity Buffer verwendet. Bei Post-Processing Verfahren gibt es die Einschränkung, dass sie nur auf schon erzeugten Bilder arbeiten können und nur begrenzte Informationen über die Szene verfügen.

4.7. Zusammenfassung

Die vorgestellten Verfahren zeigen, dass momentan die Wahl des richtigen Motion Blur Verfahrens vom dargestellten Kontext, der zu Grunde liegenden Architektur der Anwendung und den persönlichen Präferenzen abhängt.

Für diese Arbeit ist es wichtig Bewegung in Form von Speed Lines darzustellen. Dafür muss die darzustellende Geschwindigkeit ermittelt werden. Für beliebige Geometrie ist die Erzeugung des Velocity Buffers am geeignetsten und wird in den folgenden Ansätzen verwendet. Auf hier genannte Ideen wird später entsprechend zurückgegriffen.

5. Eigener Ansatz über Velocity-Buffer und TAMs

Dieses Kapitel stellt einen Ansatz vor, der Motion Blur durch Speed Lines annähert und so Schlieren erzeugt. Dabei kommen die zuvor präsentierten Verfahren, wie z.B. Screen-Spaced Motion Blur und Real-Time Hatching, zum Tragen. Es wird vorausgesetzt, dass für eine vorher gerenderte Szene Informationen wie Farbe, Normale, Position und Geschwindigkeit bekannt sind und als Texturen vorliegen. Die Umsetzung erfolgt in mehreren Schritten, deren Ergebnisse genauer analysiert werden, um auf deren Basis das weitere Vorgehen zu entwickeln.

Der Abschnitt 5.1 erläutert den Ansatz, der in diesem Kapitel verfolgt wird, gefolgt vom Abschnitt 5.2, der die Umsetzung beschreibt. Außerdem zeigt er, wie die drei ersten Renderdurchläufe durch das Shader Model 4.0 zu einem einzigen Durchlauf zusammengefasst werden können. Abschnitt 5.3 beschreibt die dabei beobachteten Mängel, schlägt Lösungen vor und setzt sie um.

5.1. Beschreibung des Ansatzes

Der in diesem Kapitel beschriebene Ansatz benötigt vier Renderdurchläufe, wobei die Szene zunächst dreimal gerendert wird und erst der letzte Durchlauf die Schlieren erzeugt. Das mehrmalige Rendern dient dazu die gerenderten Objekte in unbewegte und bewegte Objekte einzuteilen sowie die Geometrie der bewegten Objekten zu strecken. Die gestreckte Geometrie wird für die spätere Bestimmung der Schraffurstärke benötigt. Informationen, wie Geschwindigkeit, Farbe und Tiefe des vordersten Oberflächenpunkts werden als Textur abgespeichert, die auch G-Buffer genannt wird. Der vierte und letzte Renderdurchlauf erzeugt aus den G-Buffern und einer TAM die Schlieren.

Die Schlieren befinden sich an den Stellen, die beim Motion Blur Verfahren aus Unterabschnitt 4.5.5 nicht mehr voll deckend sind und den Hintergrund durchscheinen lassen. Da sich alle bewegten Objekte auf einem schwarzen und durchsichtigen Hintergrund befinden, berechnet die LIC über den berechneten Alphawert den Deckungsfaktor. Der Deckungsfaktor gibt die Stärke der Schraffur an, die in der TAM abgelegt ist. Die Texturkoordinate für die Schraffurtextur ergibt sich aus der Texturkoordinate für den momentan betrachteten Pixel, welche anhand des Geschwindigkeitvektors gedreht wird. Dieses Vorgehen richtet die Striche der Schraffur

entlang der Bewegung aus. Das so entstehende Bild mit den schraffierten Objekten wird über die Tiefe mit der Hintergrundszene zusammengeführt.

5.2. Umsetzung

Unbewegte Geometrie stellt häufig den Szenenhintergrund dar, bewegte Geometrie ist dagegen meistens im Vordergrund der Szene zu sehen. Es muss also zwischen unbewegten und bewegten Objekten unterschieden werden. Diese Unterscheidung kann auf der CPU-Seite geschehen oder im Shader selbst, wobei sie auf der CPU-Seite nur auf ganzen Objekten oder kleineren Teilen erfolgen kann. Im Shader läuft die Unterscheidung anhand des Geschwindigkeitvektors $\vec{v}$ für den momentanen Vertex ab. Damit später der Zugriff auf die Geschwindigkeit auch außerhalb des Objekts möglich ist, wird die Geometrie noch zusätzlich gestreckt und in einen G-Buffer dargestellt.

Der Geschwindigkeitsvektor $\vec{v}$ ist die Differenz aus den Punkten p^w und $p^{w\prime}$, und weist in die Bewegungsrichtung. Die Punkte p^w und $p^{w\prime}$ ergeben sich aus der Transformation des Vertex p mit der derzeitigen und vorherigen ModelView Matrix M und M'. Da später nur die Darstellung in der Bildebene von Bedeutung ist, werden p^w und $p^{w\prime}$ erst projiziert und dann $\vec{v}^p$ in der Bildebene ermittelt. Im Fragment-Shader entscheidet schließlich der Vergleich der Länge von $\vec{v}^p$ über einen Schwellwert, ob sich das betreffende Fragment bewegt hat oder nicht.

Die Streckung erfolgt im Vertex-Shader und geschieht durch den Vergleich des Skalarprodukts aus der Normalen $\vec{n}$ und Geschwindigkeit $\vec{v}$ des derzeitigen Vertex. Ist $\vec{n} \circ \vec{v} > 0$, weist $\vec{n}$ in Richtung der Bewegung und es wird die derzeitige Position p^w an die Pipeline weitergereicht, ansonsten wird $p^{w\prime}$ verwendet. Der Unterabschnitt 4.5.5 beschreibt dieses Vorgehen in ähnlicher Form. Bei der Anwendung von gestreckter Geometrie kann es aber auch zu Verdeckungen kommen, vgl. Unterabschnitt 4.5.6 auf Seite 47.

Jeder Renderdurchlauf benutzt ein anderes Frame Buffer Objekt, welches es ermöglicht, die Szene einem G-Buffer zu speichern. In G-Buffern wird die Farbe, die Tiefe sowie weitere Informationen gespeichert. Beim Zeichnen der gestreckten Geometrie wird $\vec{v}$ als Projektion in der Bildebene mitgespeichert.

5.2.1. Erstellung von G-Buffern in mehreren Durchläufen

Um die G-Buffer zu erstellen werden drei Durchläufe benötigt, die jeweils die unbewegten und bewegten Objekte sowie ihre gestreckte Geometrie berechnen und darstellen. Unbewegte Objekte werden auch als Hintergrundszene bezeichnet.

Die beiden Renderdurchläufe für die Hintergrundszene und die bewegten Objekte sind nahezu identisch und unterscheiden sich nur durch den Vergleich der Länge von $\vec{v}$. Beide Varianten werden in separate Frame Buffer Objekten gerendert und jeweils die dazugehörige Farbe und

Tiefe gespeichert. Die tatsächliche Farbe kann bei Bedarf in einen weiteren Post-Processing berechnet werden.

Die gestreckte Geometrie wird zusammen mit der Tiefe, der Farbe sowie der projizierten Geschwindigkeit gerendert. In diesem Durchlauf wird nur bewegte Geometrie gestreckt.

5.2.2. Erstellung von G-Buffern in einem Durchlauf

Anstatt die gesamte Szene dreimal zu Rendern, können alle benötigten Informationen auch in einem Durchlauf berechnet und abgespeichert werden. Diese Variante wird durch mehrschichtige FBOs ermöglicht [PBN07], vgl. auch Abschnitt A.3 auf Seite 115. Mehrschichtige FBOs sind nicht mit den Color Attachments (Farbbuffer) zu verwechseln. Jede Ebene des FBOs hat einen Tiefenbuffer und weitere Farbbuffer. Um ein FBO mit mehreren Ebene zu erlangen, müssen 3D Texturen angehangen werden.

Um in die einzelnen Ebenen schreiben zu können, muss im Geometry-Shader die Variable `gl_Layer` entsprechend gesetzt werden. Deshalb werden alle Berechnungen in den Geometry-Shader verlagert. Zunächst berechnet das Shaderprogramm für jeden Vertex des eingehenden Primitivs die projizierte Geschwindigkeit $\vec{v}^p$ und stellt anhand eines Schwellwerts fest, ob es sich um unbewegte oder bewegte Geometrie handelt. Liegt unbewegte Geometrie vor, wird sie in die erste Ebene geschrieben (`gl_Layer` = 0), sonst in die zweite (`gl_Layer` = 1). Dabei ist zu beachten, dass der Wert der Variable `gl_Layer` für alle Vertices eines Primitivs identisch sein muss.

Damit verschmelzen zwei Durchläufe zu einem. Es fehlt noch die Streckung, welche jedoch nur bei bewegter Geometrie erfolgt. An dieser Stelle wird im Geometry-Shader neue Geometrie erzeugt, entsprechend dem Skalarprodukt $\vec{n} \circ \vec{v}$ verschoben und in die dritter Ebene des FBOs geschrieben (`gl_Layer` = 2).

Beim Rendern von bewegter und unbewegter Geometrie sollen zum Schluss G-Buffer von der Tiefe und der Farbe sowie die Geschwindigkeit der gestreckten Geometrie herauskommen. Ein Beispiel für diese G-Buffer ist in Abbildung 5.2.1 aufgeführt. Diese Fälle werden normalerweise getrennt voneinander in unterschiedlichen Fragment-Programmen behandelt. Während eines Renderdurchlaufs ist es aber nicht möglich ein Programm über den Geometry-Shader zu wechseln. Stattdessen kann eine zusätzliche Ausgabevariable definiert werden, die als Flag dient und dem Fragment-Shader signalisiert, welcher Programmcode ausführen werden soll.

5.2.3. Schlieren

Im letzten Renderdurchlauf erhält der Fragment-Shader die G-Buffer der Szene als Texturen. Aus den gegebenen Informationen wird für einen Geschwindigkeitsvektor die Line Integral

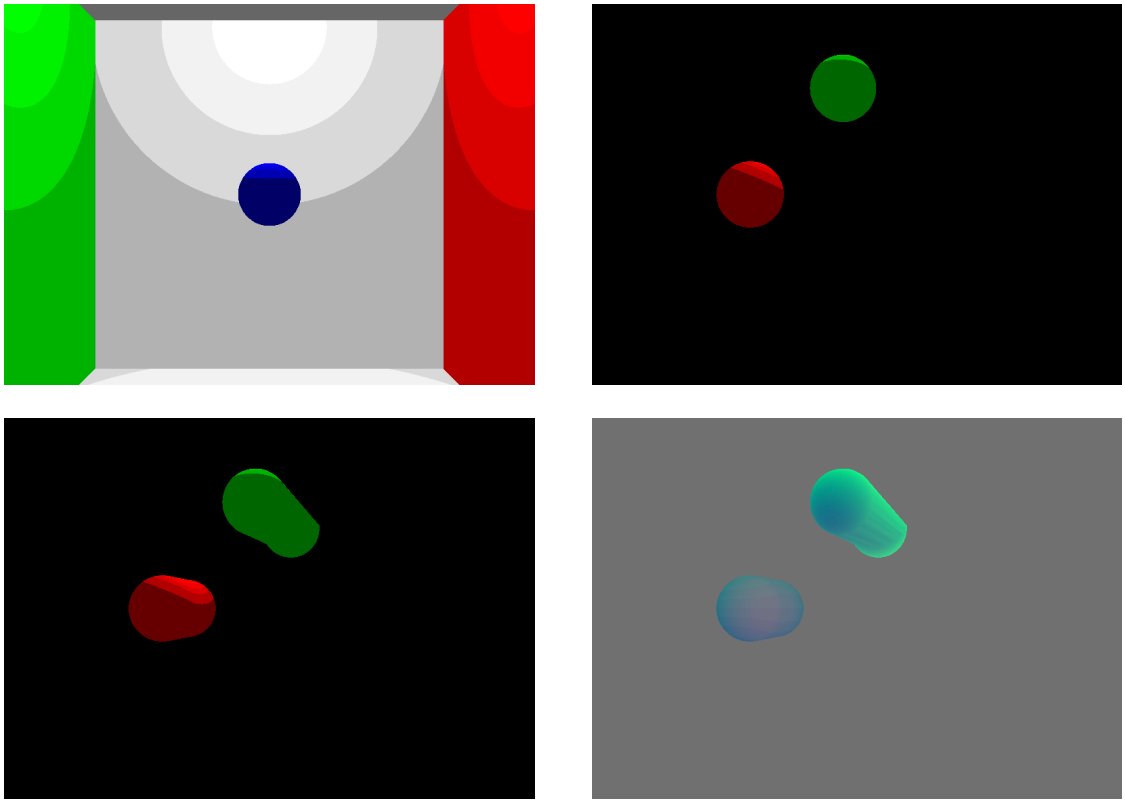

Abbildung 5.2.1.: Cornel Box mit drei Kugeln in drei verschiedenen G-Buffer gerendert: Hintergrund oder auch unbewegte Objekte (*oben links*), bewegte Objekte (*oben rechts*), gestreckte Geometrie von bewegten Objekten (*unten links*) und der dazugehörige Velocity-Buffer (*unten rechts*).

Convolution (LIC) mit acht Abtastungen durchgeführt, wie in Unterabschnitt 4.5.4 beschrieben. Somit lässt sich festzustellen wie deutlich das Objekt zu sehen ist. Dieses Vorgehen setzt voraus, dass alle Pixel im leeren Frame Buffer mit dem Alphawert 0 initialisiert sind.

Der Alphawert der ermittelten Farbe entspricht dem gewünschten Deckungsfaktor i und gibt die Intensität der Schraffur an. Über i wird die richtige Schraffurtextur aus der zur Verfügung stehenden Tonal Art Map ausgewählt und damit das bewegte Objekt texturiert. Die Striche in der TAM müssen dabei entlang einer einheitlichen Richtung in der Textur verlaufen, damit sie entlang einer Bewegungrichtung ausgerichtet werden können. Eine Schraffurtextur an sich wird als eine Alphamaske interpretiert und der aus ihr ermittelte Alphawert A wird mit der Objektfarbe C multipliziert. Bei der Auswahl über i wird nicht eine Intensitätstufe ausgewählt, sondern linear zwischen den am nächsten liegenden Stufen interpoliert.

Außerdem dreht das Fragment-Programm die Texturkoordinaten für die Alphamaske, anhand der Ausrichtung der Geschwindigkeit,. Als Texturkoordinate wird die momentane Position in der Bildebene genommen. Die Rotation ergibt sich aus der Transformation des Koordinatensystems der Striche in das Koordinatensystem des Geschwindigkeitsvektors. Bei der verwendeten TAM sind alle Striche entlang der x-Achse ausgerichtet. Die x-Achse im Koordinaten-

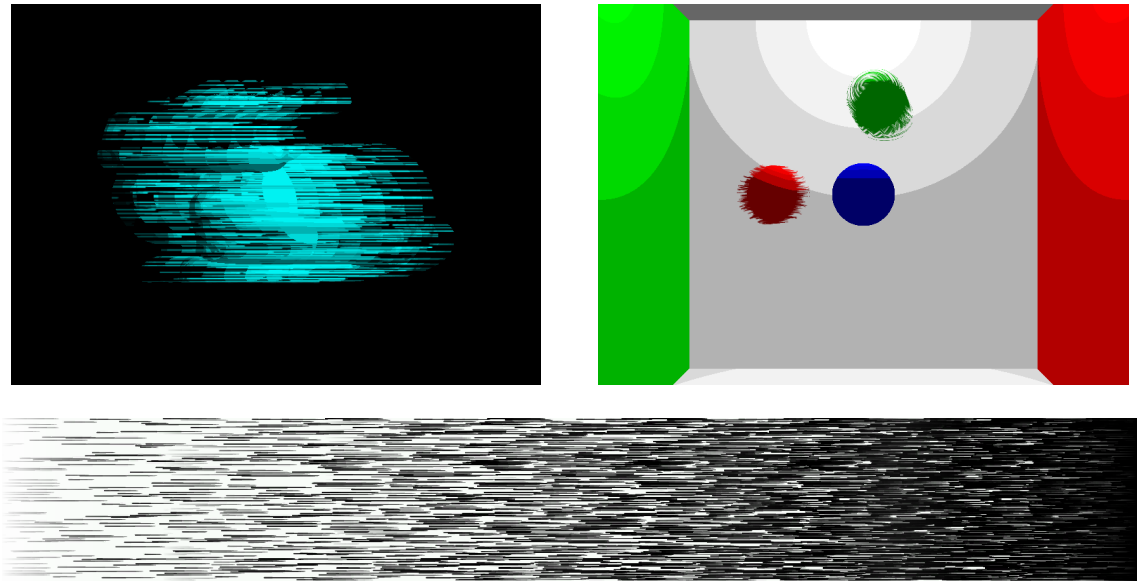

Abbildung 5.2.2.: Erste Ergebnisbilder vom verfolgten Ansatz mit der verwendeten Schraffurtextur.
links: Ein Hase, der sich von rechts nach links bewegt. *rechts*: Zwei Kugeln, die um eine blaue Kugel in der Mitte kreisen. *unten*: Erstellte und verwendete Schraffurtextur.

system des Geschwindigkeitsvektors liegt parallel zum Vektor. Für die gegebene normierte Bewegungsrichtung $\vec{v}_n$ ergibt sich folgende Rotationsmatrix R:

$$R = \begin{pmatrix} v_x & v_y \\ -v_y & v_x \end{pmatrix}, \quad \text{mit } \vec{v}_n = (v_x\ v_y)^T$$

Die Farbe C ergibt sich aus der Farbe der gestreckten Geometrie und wird mit Alphawert A gewichtet. Unter Berücksichtigung der Tiefe führt das Shaderprogramm den resultierenden Farbwert mit dem Hintergrund zusammen. Der Prozess beachtet bei der Zusammenführung die Tiefe des Szenenhintergrunds und der gestreckten Geometrie.

5.2.4. Ergebnisse

Die Abbildung 5.2.2 zeigt zwei Ergebnisbilder der Implementation. Das linke Bild zeigt einen blauen Hasen, der sich von rechts nach links bewegt und nur noch durch Schraffuren angedeutet ist. Das rechte Bild stellt zwei Kugeln dar, die um eine blaue Kugel in der Mitte kreisen. Die rote Kugel rotiert auf der Horizontalen, hingegen umläuft die grüne Kugel die blaue auf einer eher zufälligen Bahn. Als Schraffurtextur wurde das untere Bild verwendet, wobei eine Kombination der TAM darstellt ist.

Der gewünschte Effekt ist deutlich zu erkennen, wobei bei der Umsetzung mehrere Nebeneffekte aufgetreten, die im folgenden Abschnitt genauer untersucht und behoben werden.

5.3. Beobachtete Mängel

Bei der näheren Untersuchung der Ergebnisse aus Abbildung 5.2.2 sind mehrere Mängel festzustellen.

Der erste Mangel betrifft die Übergänge zwischen den verschiedenen Intensitäten der Schraffuren und ist im linken Bild zu sehen. Diese Übergänge sind sehr abrupt und fallen dadurch deutlich auf. Die Lösung wird in Unterabschnitt 5.3.1 behandelt.

Desweiteren wird bei der Umsetzung die Geschwindigkeit perspektivisch nicht korrekt interpoliert. Unterabschnitt 5.3.2 stellt die Ursache und deren Lösung vor.

Außerdem ist nur gestreckte Geometrie zu sehen, wenn sich alle Objekte bewegen. Unterabschnitt 5.3.3 zeigt, dass die Ursache beim berechneten Deckungsfaktor i liegt und stellt eine Lösung vor.

Die sich ergebenen Strichmuster aus dem rechten Bild in Abbildung 5.2.2 sehen nicht gut und treten bei der grünen Kugel am stärksten hervor. Besonders bei Bewegung durch Skalierung oder Rotation erzeugen sie beim Betrachter nicht den Eindruck eines bewegten Objekts und wirkt irritierend. Der Unterabschnitt 5.3.4 befasst sich genauer mit diesem Problem.

5.3.1. Abrupte Übergänge

Bei der näheren Betrachtung des linken Bildes in Abbildung 5.2.2, können insgesamt acht Abstufungen ausgemacht werden. Diese Abstufungen scheinen von der Beschaffenheit der TAM herzurühren. Der Gewichtungsfaktor A, der aus der TAM ermittelt wird, stellt schon das Ergebnis der Interpolation über i zwischen den am nächsten liegenden Intensitätstufen dar. Das Problem muss bei dem verwendeten i liegen.

Der Deckungsfaktor i wird über die LIC ermittelt und wird aus acht Farbwerten berechnet. Da acht Intensitätsstufen vorliegen - eine schwarz, eine weiß und sechs unterschiedliche Schraffuren - wird gar nicht zwischen ihnen interpoliert, weil sich zu wenige Abstufungen ergeben. Aus dem Grund muss eine höhere Abtastung entlang des Vektors erfolgen.

Statt nur acht mal abzutasten und festzustellen, ob das betrachtete Objekt zu sehen ist, kann zweimal oder sogar viermal so oft entlang des Vektors abgetastet werden. Vergleichsbilder mit 16 und 32 Abtastungen sind in Abbildung 5.3.1 einer achtfachen Abtastung gegenübergestellt. Diese Veränderungen bewirken aber auch einen höheren Rechenaufwand, weil nun mehr Texturzugriffe stattfinden.

5.3.2. Perspektivisch korrekte Geschwindigkeitsvektoren

Durch genauere Analyse der Ergebnisse ist festzustellen, dass die Geschwindigkeit bei der Übergabe an den Fragment-Shader nicht erwartungsgemäß berechnet wird. Die Abbildung 5.3.2 zeigt den resultierenden Effekt. Es ist eine Cornel Box zu sehen, dessen bewegte

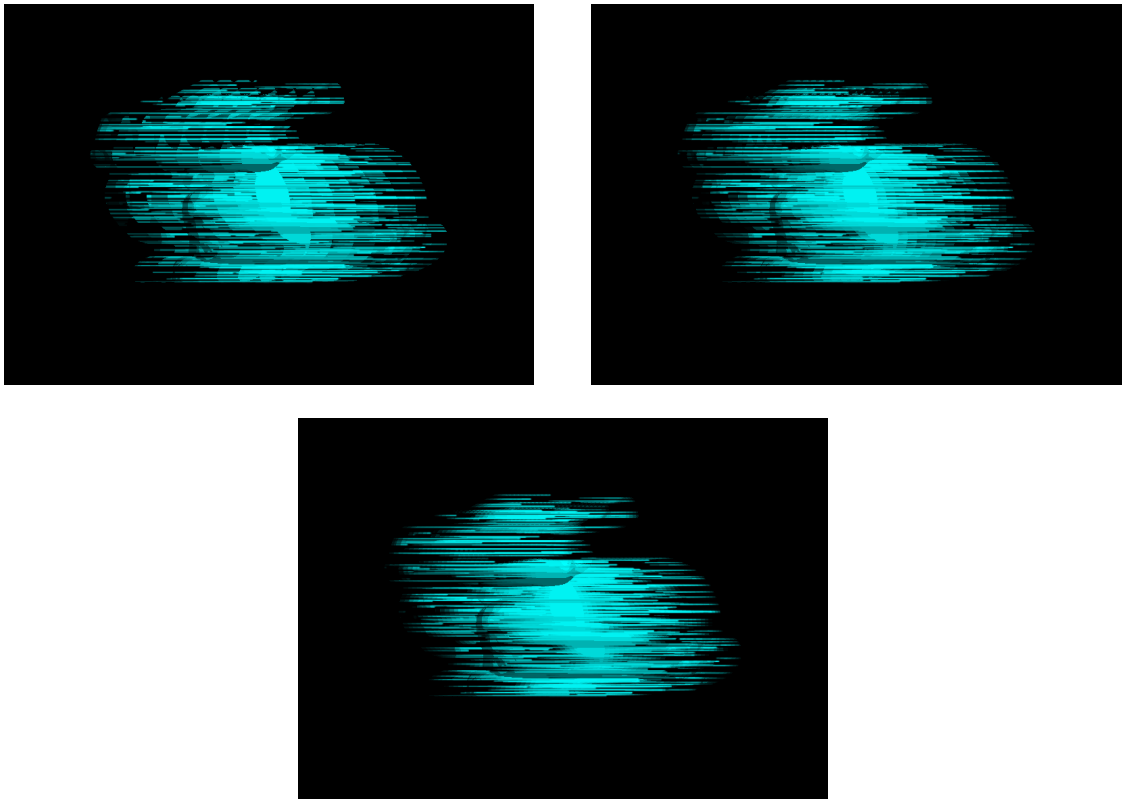

Abbildung 5.3.1.: Ein angedeuteter Hase, der sich von rechts nach links bewegt. Die Abtastungen erfolgten 8 (*links*), 16 (*rechts*) und 32 (*unten*) mal.

Oberflächen in der Bildebene mit einer Schraffurtextur überlagert sind. Diese Textur wurde anhand der vorliegenden Geschwindigkeitsvektoren ausrichtet und die Kamera bewegt sich auf die Cornel Box zu. Im linken Bild sind die sich ergebenden Muster zu erkennen, die in keinster Weise Rückschlüsse auf die Bewegungsrichtung zulassen. Im rechten Bild ist der entsprechende Velocity-Buffer abgebildet.

Anhand der Darstellung im linken und rechten Bild ist zu erkennen, dass über die Flächen der Cornel Box eine Diagonale verläuft, an der im linken Bild die Schraffuren gebrochen werden und die im Velocity-Buffer deutlich hervor tritt. Daraus folgt, dass die Interpolation der Geschwindigkeit zwischen den drei Punkten des gezeichneten Dreiecks erfolgt. Anstatt die Geschwindigkeit dem Fragment-Shader direkt zu übergeben, kann auch die vorherige und jetzige Position weitergereicht werden um dann die Geschwindigkeit zu berechnen. Dadurch ergeben sich korrekte Geschwindigkeitsvektoren, wie das untere Bild beweist.

5.3.3. Änderung der Berechnung des Deckungsfaktors i

Durch die unzureichende Berechnung des Deckungsfaktors i, sind statt angedeuteten Objekten nur gestreckte Objekte zu sehen. Im linken Bild in Abbildung 5.3.3 ist der Mangel gut zu erkennen und sollte eine rote und grüne Kugel zeigen, die um eine weitere blaue Kugel krei-

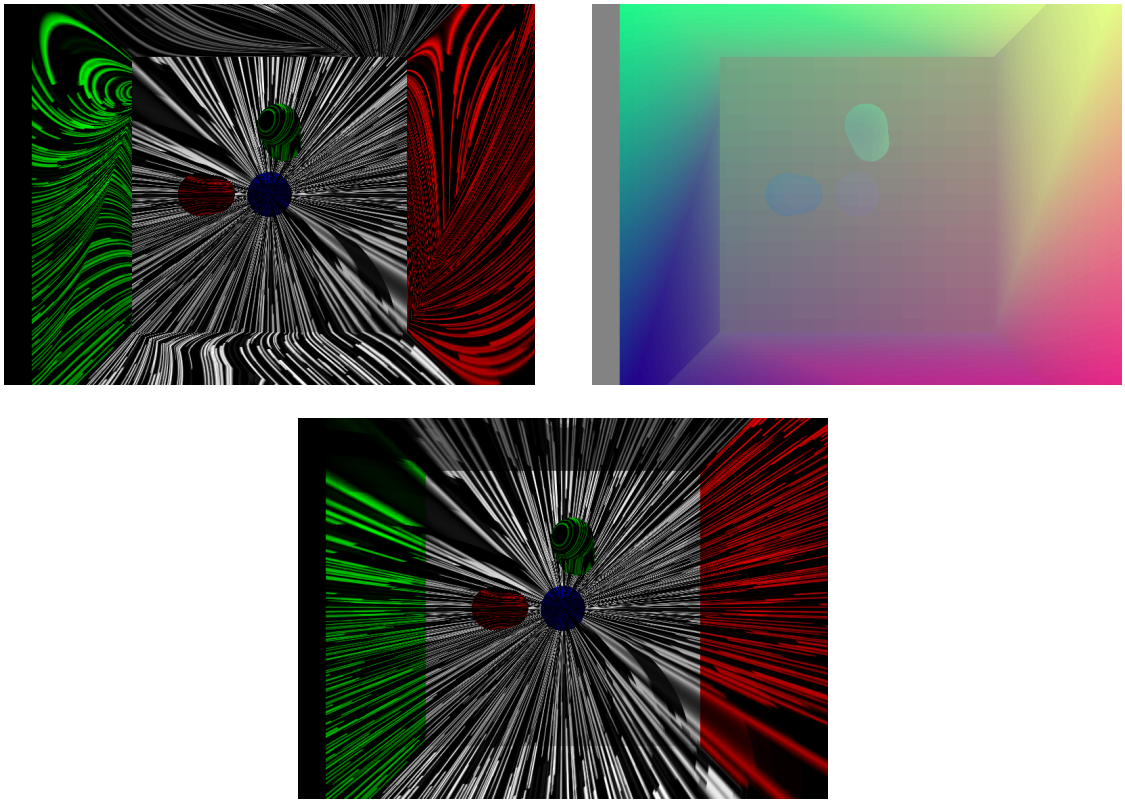

Abbildung 5.3.2.: Eine Cornel Box auf die sich die Kamera zu bewegt. Die Szene (*links*) wurde mit Schraffurtexturen überlagert, die anhand der Geschwindigkeitsvektoren aus dem Velocity-Buffer (*rechts*) ausrichtet wurden. Die korrekte Interpolation der Vektoren ist im *unteren* Bild abgebildet.

sen, während sich die Kamera nach rechts bewegt. Stattdessen ist nur die gestreckte Geometrie ohne den erwarteten Effekt zu erkennen.

Der Deckungsfaktor i wird mittels einer LIC über dem Farbbild der bewegten Geometrie ermittelt. Wenn sich nur ein Objekt bewegt, ist es das einzige Objekt im G-Buffer und umgeben von einem vollkommen transparenten schwarzen Farbton. Der Alphawert des sich nach der Faltung ergebenen Farbwerts repräsentiert den gewünschten Deckungsfaktor i. Aber wenn sich zwei Objekte oder sogar die gesamte Szene bewegt, kann i nicht mehr korrekt ermittelt werden, weil auch die Farbtöne der anderen Objekte verwendet werden.

Die Lösung für das Problem wurde schon in Unterunterabschnitt 4.5.6.3 angedeutet. Dabei ist es erforderlich zusätzlich eine Objekt ID mitzuführen und in einen G-Buffer zu speichern. Die LIC muss so angepasst werden, dass nur der prozentuale Anteil der eigenen Objekt ID auf der Linie ermittelt wird. Damit klar ist zu welcher Objekt ID der prozentuale Anteil ermittelt werden soll, muss nicht nur für bewegte Objekte die ID gespeichert werden, sondern auch für die gestreckte Geometrie. Die ID der gestreckten Geometrie dient dabei als Referenz für den Vergleich. Die ID an sich kann in der z-Koordinate des Geschwindigkeitsvektor gespeichert werden, weil es sich um einen Vektor im Bildraum handelt. Das rechte Bild in Abbildung 5.3.3

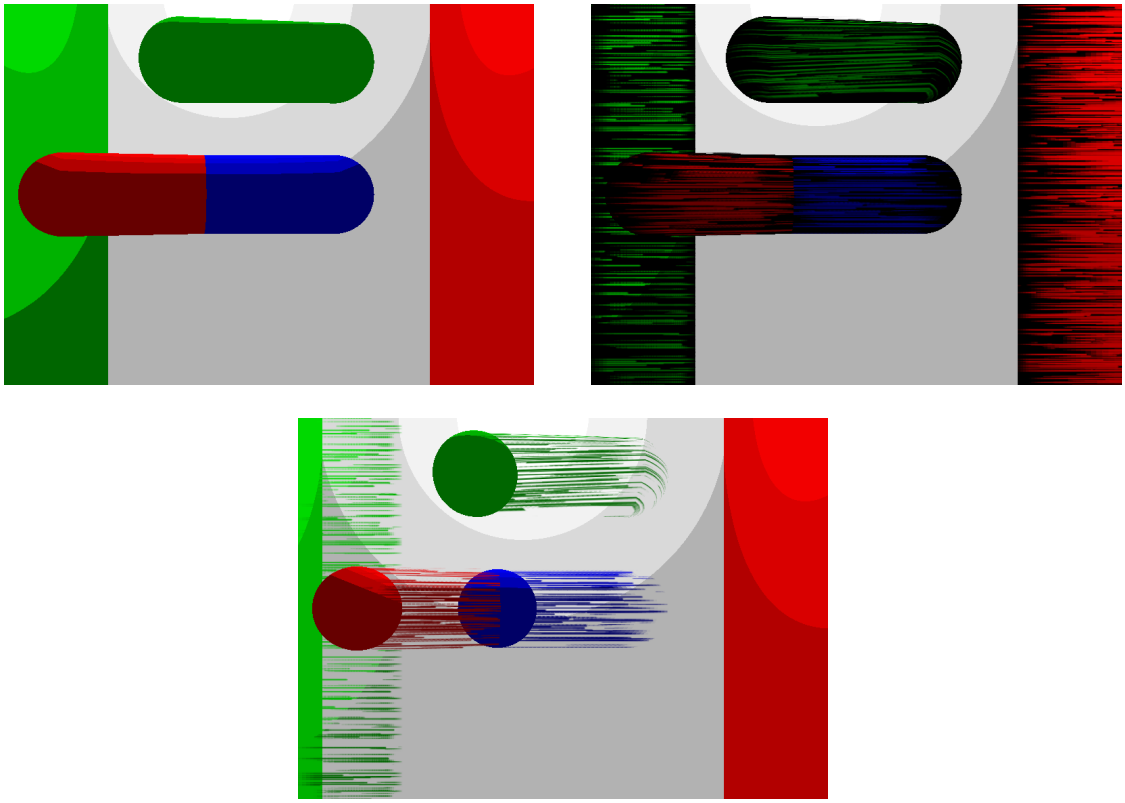

Abbildung 5.3.3.: Zwei Kugeln, die eine blaue Kugel umkreisen in einer Cornel-Box; gleichzeitig bewegt sich die Kamera nach rechts.
Deckungsfaktor i wird über den Alphawert der gestreckten Geometrie (*links*), über die Objekt ID ohne Einbeziehung des Hintergrunds (*rechts*) und mit Hintergrund (*unten*) ermittelt.

zeigt, die vorgenommene Änderung. Es sind zwar Speed Lines zu sehen, aber an den Stellen, an denen sich die gestreckte Geometrie befindet, ist der Hintergrund schwarz.

Wie zu sehen ist, gibt es keinen Hintergrund - also gibt es keine unbewegten Objekte hinter den bewegten Objekten, weil sich entweder alle Objekte bewegt haben oder wie in diesem Fall sich die Kamera bewegt hat. Um trotzdem einen vernünftigen Hintergrund zu erlangen, muss festgestellt werden, ob überhaupt der Hintergrund gezeichnet wurde und falls nicht muss stattdessen die Farbe der bewegten Objekte genommen werden. Zum Feststellen wird der Alphawert des Hintergrunds verwendet. Das Ergebnis im unteren Bild in Abbildung 5.3.3 zeigt die Modifikation. Die durchgeführte Änderung bewirkt leider, dass nicht nur Schlieren gezeichnet werden, sondern auch die bewegten Objekten an sich zu erkennen sind. Dieses Verhalten tritt aber nur auf, wenn sich die gesamte Szene oder die Kamera bewegt.

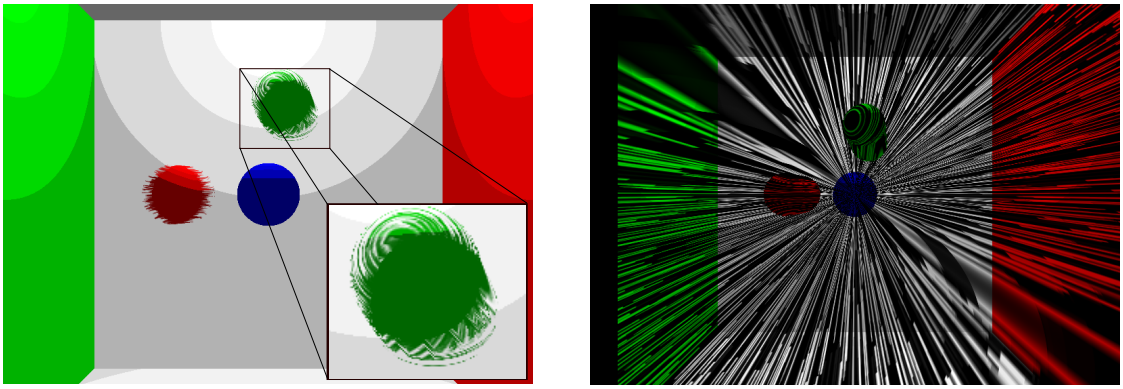

Abbildung 5.3.4.: Zwei Beispiel dafür, dass die gewählte Texturkoordinate nicht korrekt gut gewählt ist. Besonders bei rotierenden Objekten wirken die erzeugten Strichmuster irritierend (*links*). Ungleichmäßige Strichgrößen können ebenfalls entstehen.

5.3.4. Berechnung der Texturkoordinate

Die richtige Wahl einer Texturkoordinate ist wichtig, damit die Striche eine einheitliche Größe bekommen, ordentlich ausgerichtet sind und die richtige Länge erhalten. Abbildung 5.3.4 zeigt zwei Beispiele, bei denen die erzeugten Strichmuster den Eindruck von gerichteter Bewegung nicht vermitteln und die erzielte Strichdicke nicht gleichmäßig ist und stärke vom Betrachungswinkel abhängig ist. Im linken Bild sind rotierende Kugeln in einer Cornel Box zu sehen. Besonders bei der grünen Kugel wirken die Strichmuster irreführend. Der Betrachter ist nicht in der Lage nachzuvollziehen in welche Richtung sich die Kugel bewegt. Beim rechten Bild bewegt sich die Kamera auf die Cornel Box zu. Die Muster können die Bewegungsrichtung eindeutig hervorheben, aber die Strichdicke variiert stark in Abhängigkeit ihrer Ausrichtung.

Ein Lösungansatz ist Triplanares Texturing zu verwenden. Diese Idee kam aber zu einem späten Stadium der Arbeit, weshalb sie nicht umgesetzt werden könnte. Triplanares Texturing ermöglicht es auf unterschiedlichste Geometrie Texturen zu legen [Gei07, GT07]. Dafür werden drei Texturen jeweils entlang der x-, y- und z-Achse projiziert und anhand der Normale zwischen ihnen interpoliert. Die Idee ist die verwendete TAM über das Objekt zu legen und die Projektionsachsen entsprechend der Bewegungsrichtung auszurichten.

5.4. Zusammenfassung

Der verfolgte Ansatz über den Velocity-Buffer und die TAM konnte umgesetzt werden, barg dennoch mehrere Mängel in sich. Der allgemeine Mangel des Verfahrens liegt einmal in der unzureichenden Möglichkeit Objekte mit Texturen zu versehen. Daneben können die Strichgröße und Strichlänge nicht beeinflusst werden. Außerdem entstehen Verdeckungen, die durch die gestreckte Geometrie hervorgerufen werden. Besonders die Verdeckungen erschweren es al-

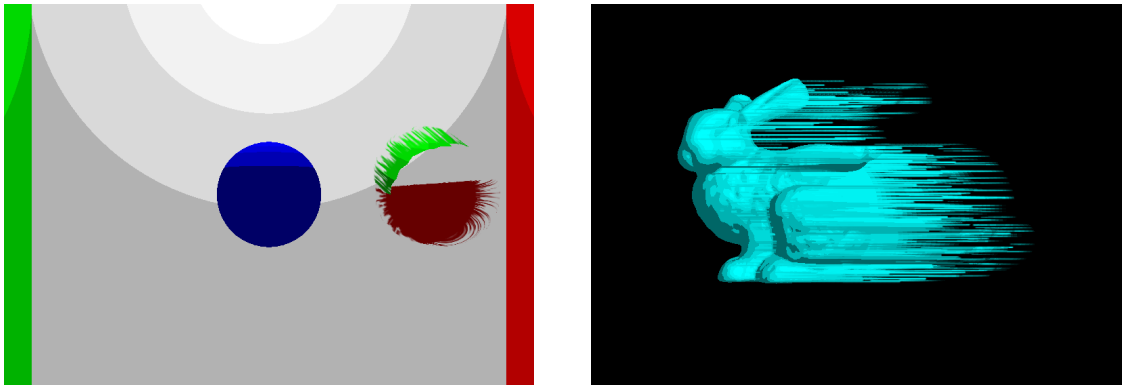

Abbildung 5.4.1.: Weitere Resultate des verfolgten Ansatzes. *links*: Durch die Kollision der beiden Kugeln entsteht Verdeckung, weshalb die Farbwerte nicht mehr korrekt ermittelt werden können. *rechts*: Ein sich von rechts nach links bewegender Hase, der durch das Verfahren mit Speed Lines erlangt hat.

le Objekte korrekt durch Schraffuren anzudeuten, vgl. Abbildung 5.4.1 linkes Bild. Außerdem sind einige weitere Probleme durch die durchgeführten Änderungen entstanden. Die Probleme wurden nicht weiter diskutiert, weil keine Lösung für die richtige Ausrichtung der Schraffuren vorlag.

Mit dem vorgestellten Ansatz ist es aber dennoch möglich, Objekte mit Speed Lines zu versehen. Ein Beispiel dafür zeigt das rechte Bild in Abbildung 5.4.1 mit einen sich bewegenden Hasen. Hierbei werden die sich ergebenden Streifen mit den bewegten Objekten und dann mit dem Hintergrund vereint. Verdeckungen spielen dabei kaum eine Rolle, denn die bewegten Objekte füllt die leeren Stellen mit Farbwerten aus.

Aus den oben genannten Gründen wurde dieser Ansatz verworfen und ein Weiterer verfolgt. In Kapitel 6 stellt diesen Ansatz vor, der statt Schraffuren aus einer TAM zu verwenden selber Strichgeometrie erzeugt und diese in die Szene zeichnet.

6. Eigener Ansatz über Seed Points

Dieses Kapitel beschreibt ein anderer Lösungsansatz, der statt einer Schraffurtextur mehrere Striche. Dabei verwendet bei diesem Ansatz der Geometry-Shader Informationen wie Geschwindigkeitsvektoren, Positionen und Farben sowie zufällig verteilte Punkte um Striche zu erzeugen. Die erzeugten Striche sollen zum Schluss die bewegten Objekte ersetzen und so Schlieren hervorrufen.

Dieser Ansatz erzeugt die gesamte Strichgeometrie auf der GPU-Seite unter Verwendung des Geometry-Shaders. Vertex- und Fragment-Shader verarbeiten dagegen kaum Informationen. Er ähnelt dem Vorgehen von [HS04], aber die Partikel - hier Seed Points genannt - werden nicht über das Objekt gestreut, sondern gleichmäßig über den Viewport. Neue Techniken des Shader Modell 4.0 werden in diesem Abschnitt verwendet, vgl. [PBN07].

Die Erzeugung von Speed Lines erfolgt dabei durch Polygon, basierend auf dem Vorgehen von Muders [Mud10], der Striche durch Polygone und einer Strichtextur darstellt. Im Gegensatz zu seinem Vorgehen, wird der Geometry-Shader zum Erzeugen von Striche eingesetzt.

6.1. Beschreibung des Ansatzes

Für die hier verfolgte Vorgehensweise sind insgesamt drei Renderdurchläufe notwendig. Im ersten Durchlauf wird die gesamte Szene mittels Deffered Shading in verschiedene Color-Attachments eines Frame Buffer Objekts abgespeichert, die aus Texturen bestehen. Informationen über Position, Geschwindigkeit als Vektor und Farbe werden in diesem Schritt berechnet und abgespeichert. Die Geschwindigkeit wird auf die gleiche Weise berechnet, wie schon beim Screen-Spaced Motion Blur Verfahren aus Abschnitt 4.5, hingegen wird sich bewegende Geometrie nicht gestreckt. Unter Verwendung von Frame Buffer Objekten mit zwei Ebenen wird die Szene in bewegte und unbewegte Geometrie geteilt.

Im zweiten Durchlauf werden zufällige Punkte an den Geometry-Shader weitergereicht. Die nötigen Informationen werden an den zufällig verteilten Punkten aus den drei Texturen ausgelesen und mit ihrer Hilfe ein Strich erzeugt. Jeder Strich erhält Texturkoordinaten, mit denen im Fragment-Shader unter Verwendung einer Strichtextur ein Strich gezeichnet wird.

Beim letzten Durchlauf werden die bewegten Objekte, die unbewegte Szene und die Striche ineinander übergeblendet. Dabei sollen langsame Objekte deutlich zu sehen sein und sich mit

zunehmender Geschwindigkeit mehr und mehr ausblenden. Die Striche hingegen nehmen mit zunehmender Geschwindigkeit an Deutlichkeit zu und ersetzen schließlich das ganze Objekt.

Der erste Durchlauf wurde auf ähnliche Weise schon vorher in Abschnitt 5.2 auf Seite 52 eingeführt. Im folgenden Abschnitt 6.2 wird auf den zweiten Durchlauf eingegangen, wobei der Fokus auf der Generierung von Seed Points und der Stricherzeugung liegt. In Abschnitt 6.4 auf Seite 79 wird dann die Szene mit den Strichen und den bewegten Objekten verschmolzen. Die Überblendung der erzeugten Striche mit der gerenderten Szene wird in Abschnitt 6.5 auf Seite 86 diskutiert.

Mögliche auftretende Schwierigkeiten werden nach jedem Abschnitt analysiert, Lösungen hierzu entwickelt und - so fern möglich - umgesetzt.

6.2. Umsetzung der Stricherzeugung

Zu Beginn des zweiten Durchlaufs liegen Informationen wie Geschwindigkeitsvektoren, Positionen und Farben als Texturen vor. Geschwindigkeitvektoren sowie Positionen wurden mit einer Genauigkeit von 32-Bit pro Komponente abgespeichert und liegen in Weltkoordinaten vor. Die Farbe ist wie üblich mit 8-Bit pro Farbkanal in einer Textur hinterlegt.

Die Szene wurde in einem Frame Buffer Objekt mit zwei Ebenen gerendert, in der ersten Ebene ist die starre Szene und in der zweiten sind alle bewegten Objekte zu finden. Es wird vorerst die gesamte Szene in eine Ebene gerendert, weil der hier verfolgte Lösungsansatz erst noch entwickelt wird. Dieses Vorgehen verhilft zu einfacheren Vergleichen. Abschnitt 6.5 auf Seite 86 führt zwei Ebenen ein und erläutert die nötigen Veränderungen im zweiten Durchlauf.

Anzumerken ist, dass die Verwendung von vielen G-Buffern ebenso wie eine höhere Bit-Genauigkeit viel Speicherplatz in Anspruch nimmt. Es wurde sich an dieser Stelle für den Mehrverbrauch an Speicherplatz entschieden, um spätere Rückrechnungen z.B. durch die Projektionsmatrix zu vermeiden.

6.2.1. Seed Points

Seed Points bilden die Grundlage für die Verwendung des Geometry-Shaders, der für jeden Seed Point einen Strich erzeugt. Der Geometry-Shader arbeitet nur auf Basis von Geometrie, wobei immer auf Basis einer vollständigen Primitive operiert wird. Die einfachste Art eines Primitivs ist der Punkt, weswegen Seed Points an den Geometry-Shader weitergereicht werden.

Der Geometry-Shader ermittelt alle Informationen aus der Szene über die Seed Points, weshalb sie als Texturkoordinaten anzusehen sind um die Informationen aus den vorgerenderten Texturen zu erlangen. Da sie als Texturkoordinaten behandelt werden, sind Seed Points Punkte

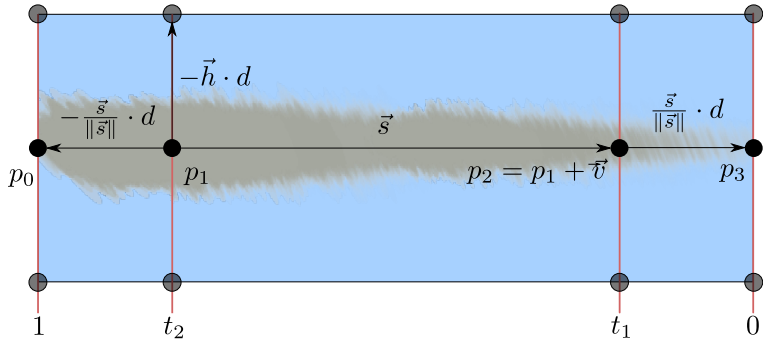

Abbildung 6.2.1.: Diese schematische Zeichnung zeigt, wie ein Strichpolygon konstruiert werden kann. Die grauen Punkte sind die Eckpunkte des zu konstruierenden Polygons. Im Hintergrund ist eine Beispieltextur zu sehen.

aus dem $\mathbb{R}^2$ und all ihre Komponenten liegen im Wertebereich $[0, 1]$. Mit ihnen werden Daten wie z.B. die Geschwindigkeit und die Position eines Objekts ermittelt, die benötigt werden um Linien zu erzeugen. Seed Points werden dafür zufällig über den Texturbereich verteilt.

Seed Points bilden damit eine Wolke aus Texturkoordinaten, die als Geometrie an den Geometry-Shader gegeben wird. Damit sie nicht bei jedem Durchlauf neu generiert werden müssen, werden sie vorab berechnet und in einem Vertex Buffer Objekt abgelegt. Die Punkte werden im Vertex-Shader nicht transformiert, sondern direkt an den Geometry-Shader weitergereicht.

6.2.2. Stricherzeugung

Ein Punkt p und ein Vektor $\vec{s}$, welche im Raum $\mathbb{R}^3$ liegen, beschreiben eine Strecke eindeutig. Der Punkt p ist der Startpunkt und $\vec{s}$ beschreibt die Länge und Orientierung der Strecke und zeigt auf den Endpunkt. Diese Definition reicht aus, um im Geometry-Shader eine Linie zu erzeugen. Sie beschreibt einen Strich dennoch unzureichend, wenn ihm eine Dicke d zugeordnet werden soll. Außerdem ist es mit dieser Definition nicht möglich, unterschiedliche Arten von stilistischen Strichen zu zeichnen. Stattdessen kann im Geometry-Shader mit der Dicke d ein Polygon erzeugt werden, welches eine Textur aufspannt. Damit bei einem Strich der Anfang und das Ende über eine Textur richtig beschrieben werden kann, sind Strichenden zur Definition hinzugefügt worden.

Die Abbildung 6.2.1 zeigt die Konstruktion eines Strichpolygons mit den wichtigsten Punkten und Vektoren schematisch. Die halb durchsichtigen grauen Punkte sind die acht Punkte des Polygons, die es zu konstruieren gilt.

Die Punkte p und $p + \vec{s}$ markieren den Anfang und das Ende des Strichrumpfs und werden als p_1 und p_2 bezeichnet. Die Strichenden werden erzeugt, indem zu den beiden Punkten jeweils der Vektor $\frac{\vec{s}}{||\vec{s}||} \cdot d$ subtrahiert bzw. addiert wird. Es entstehen die Punkte $p_0 = p_1 - \frac{\vec{s}}{||\vec{s}||} \cdot d$ und $p_3 = p_2 + \frac{\vec{s}}{||\vec{s}||} \cdot d$.

Das Polygon muss zum Betrachter ausgerichtet werden und noch eine Breite erhalten, damit der Strich in seiner vollen Größe wahrgenommen werden kann. Dafür wird ein Hilfsvektor $\vec{h}$ durch das Kreuzprodukt aus dem Vektor $\vec{v}$ und dem Sichtvektor $\vec{e} = (0, 0, -1)^T$ berechnet. Der gewichtete Hilfsvektor $\vec{h} \cdot d$ ermöglicht es das Polygon entlang der Strecke zwischen $\overline{p_0 p_3}$ aufzuspannen. Mit dem gewichteten Hilfsvektor $\vec{h}$ und den vier Punkten kann ein Polygon mit acht Eckpunkten konstruiert werden.

Die Strichtextur muss einen Strich vollständig abbilden, d.h. Strichanfang, -rumpf und -ende sind in der Textur enthalten. Auf diese Weise kann jeder Strich zuverlässig begonnen und beendet werden. Es werden zudem zusätzlich Texturkoordinaten benötigt. Die Orientierung des Strichs in der Textur ist dabei wichtig. In dieser Arbeit werden Strichtexturen verwendet, deren Striche von rechts nach links verlaufen. Es werden deshalb Texturkoordinaten benötigt, die die Strichenden entlang der x-Achse berücksichtigen. Für den Strichanfang liegen die Texturkoordinate im Wertebereich $[1,\ t_1] \times [0,\ 1]$. Für den Rumpf gilt der Wertebereich $[t_1,\ t_2] \times [0,\ 1]$ und für das Strichende $[t_2,\ 0] \times [0,\ 1]$. Dabei gilt, dass t_1 das Ende des Strichanfangs bzw. den Anfang des Strichrumpfs und t_2 den Anfang des Strichendes markiert, wobei beide Werte abhängig von der jeweils verwendeten Textur variieren können. In Abbildung 6.2.1 auf der vorherigen Seite ist eine Strichtextur zu sehen, dessen y-Koordinaten mit roten Linien hervorgehoben sind.

6.2.3. Stricherzeugung im Geometry-Shader

Für jeden Seed Point, der durch die Pipeline geschickt wird, wird jeweils ein Strich erzeugt. Dabei wird davon ausgegangen, dass ein eingehender Seed Point ein Punkt q aus dem $\mathbb{R}^2$ ist und jede Komponente von q im Wertebereich $[0, 1]$ liegt. Ein Seed Point wird als Texturkoordinate interpretiert um Informationen aus Texturen zu lesen. Geschwindigkeit, Position, Farbe und weitere Daten können dadurch ermittelt werden.

Die Position p^w liegt in Weltkoordinaten vor, ebenso die Geschwindigkeit $\vec{v}^w$ und entsprechen den Variablen p und $-\vec{s}$ bei der Strichkonstruktion. Die Variablen d, t_1 sowie t_2 werden als Uniform Variablen an den Shader übermittelt. Aus diesen Informationen, wie in Unterabschnitt 6.2.2 besprochen, werden ein Strichpolygon erzeugt und zur Rasterisierung an die Pipeline weitergegeben. Bevor jeder Punkt weitergegeben wird, muss er mit der Projektionsmatrix P transformiert werden.

6.2.4. Weiteres Vorgehen im Fragement-Shader

Im Fragment-Shader bekommt jeder Pixel die Farbe C^p übergeben, die zuvor vom Geometry-Shader durch einen Seed Point ermittelt wurde. Wenn eine Strich-Textur zugeordnet ist, wird die Fragmentfarbe C durch die Multiplikation der Texturfarbe C^t mit C^p bestimmt. Farbwerte mit dem Alphawert $C_\alpha = 0$ können verworfen werden. Diese Vorgehensweise bietet die Möglichkeit die Transparenz, Form und Farbe des Strichs durch die Strichtextur zu beeinflussen.

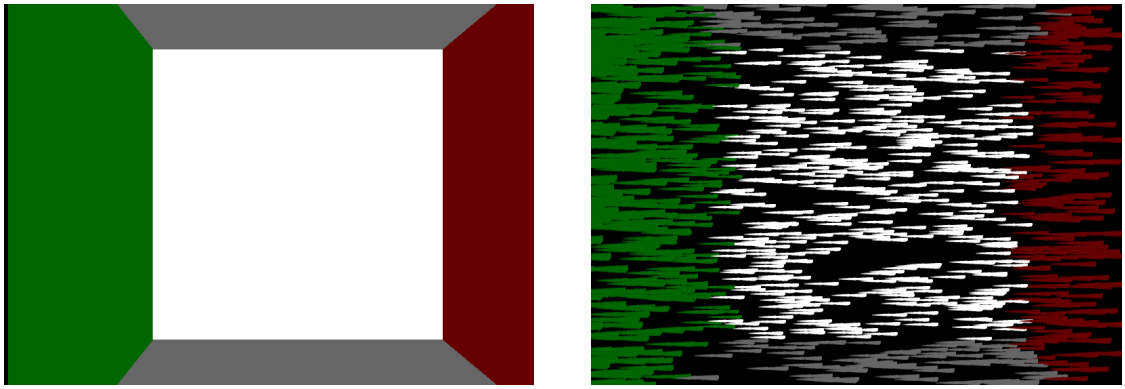

Abbildung 6.2.2.: Eine sich von rechts nach links bewegende Cornel Box.
links: normal. *rechts*: mit Strichen angedeutet. Außerdem sind ungleiche Verteilung von Seed Points und Randlücken bei seitlichen Bewegungen zu sehen.

Weitere Anmerkung Bei der Verwendung einer Strichtextur ist das richtige Blending erforderlich. Es hat sich bei der Implementation herausgestellt, dass die Komponenten des Farbwerts C aus der Textur schon mit dem Alphawert C_a gewichtet sind. Diese Gegebenheit muss bei der richtigen Einstellung der Blend-Funktion beachtet werden.

Die verwendeten Einstellungen in dieser Arbeit waren `glBlendEquation(GL_FUNC_ADD)` und `glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)`. Weitere Informationen zu Blending im Allgemeinen sind im Abschnitt A.5 oder in der OpenGL 4.0 Spezifikation [SA10] enthalten.

6.2.5. Ergebnis

Das Ergebnis der Implementation ist in Abbildung 6.2.2 anhand einer sich nach rechts bewegenden Cornel Box zu sehen. Das linke Bild mit der dargestellten Cornel Box soll nur als Anhaltspunkt verstanden werden, um die im rechten Bild mit Strichen angedeutete Cornel Box besser zu erkennen. Die verwendete Strichtextur ist schematisch als Konstruktionsdarstellung eines Strichs in Abbildung 6.2.1 auf Seite 65 zu sehen. Bei diesem Darstellungsverfahren treten Nebeneffekte auf, die im folgenden Abschnitt genauer untersucht und behoben werden.

6.3. Nebeneffekte und deren Behandlung

Beim Betrachten der Abbildung 6.2.2 sind folgende Mängel in der Darstellung erkennbar.

Der erste Mangel ist am rechten Rand der Abbildung zu beobachten. Ausgelöst durch seitliche Bewegungen entstehen sichtbare Lücken. Es fehlen Striche, die eigentlich aus dem Viewportrand in das Bild hineinragen sollten. Mehr zur Ursache und dessen Behebung in Unterabschnitt 6.3.1.

Außerdem zeigt sich, dass durch die zufällige Generierung von Seed Points Lücken im Bild entstehen. Der Bildbereich ist nicht gleichmäßig genug abgedeckt, s. Unterabschnitt 6.3.2.

Ein weiterer festgestellter Mangel tritt bei Bewegungen entlang der z-Achse auf, wobei keine Striche erzeugt werden. Die Untersuchung zu diesem Problem wird in Unterabschnitt 6.3.3 durchgeführt.

Zusätzlich ist zu beobachten, dass bei naheliegenden Objekten die Striche größer gezeichnet werden als bei weiter entfernten Objekten. Das Verhalten hängt mit der ermittelten Ursprungstiefe der Weltkoordinaten zusammen. Die Abbildung 6.3.4 auf Seite 74 verdeutlicht diesen Fall; er wird in Unterabschnitt 6.3.4 weiter besprochen und in Unterabschnitt 6.3.5 eine Lösung präsentiert.

Momentan starten alle Striche am selben Anfangspunkt, der durch den jeweiligen Seed Point bestimmt wird. Die starr positionierten Seed Points führen zu regelmäßigen Zeichnungen von Strichen, die besonders bei Kameraschwenks mit einer konstanten Geschwindigkeit und Richtung zu beobachten sind. Damit werden sie dem künstlerischen Aspekt nicht gerecht. Unterabschnitt 6.3.6 bespricht einige Ansätze zur Variierung der Strichpositionen.

6.3.1. Behandlung von Randlücken

Beim Schwenken der Kamera und seitlichen Bewegungen werden am Rande des Viewports nicht genügend Striche generiert und es entstehen Lücken, s. Abbildung 6.2.2 im rechten Bild rechter Rand. Um dieses Problem zu lösen, sind zusätzliche Seed Points generiert worden. Sie befinden sich damit auch außerhalb ihrer Textur. Der neue Wertebereich ist $[-0.1,\ 1.1]$. Außerdem wurden die verwendeten Texturen so modifiziert, dass Zugriffe außerhalb des Wertebereichs von $[0,\ 1]$ Farbwerte bzw. Informationen vom Rand zurückgeben. In OpenGL wird dafür das Textur-Attribut `GL_TEXTURE_WRAP_{S,T,R}` auf den Wert `GL_CLAMP_TO_EDGE` gesetzt.

In Abbildung 6.3.1 links kann beobachtet werden, dass nun die Lücken am rechten Rand geschlossen werden, aber zu viele Striche an deren Stelle entstehen. Eine Überbevölkerung von Strichen entsteht. Die Ursache hierfür ist die Positionsermittlung durch die Seed Points. Beim Ermitteln der Position kann nur die sichbare, abgespeicherte Position am Texturrand zurückgegeben werden. Um dennoch korrekte Positionen außerhalb der Textur zu erhalten, muss zuerst festgestellt werden, ob sich der Seed Point außerhalb befindet. Ein Seed Point q liegt außerhalb einer Textur, wenn q_x und q_y sich nicht innerhalb des Wertebereichs $[0,\ 1]$ befinden. Ist die Bedingung für ein q_i erfüllt, so wird das Verhältnis $\frac{q'_i}{q'_i \cdot \beta_i}$ für diese $i$te Komponente berechnet. Dabei steht q' für q dargestellt im Koordinatensystem des Viewports. Der Faktor β_i ist immer ≥ 0 und skaliert q' so, dass der resultierende Wert die maximale Ausdehnung zum Viewportrand beschreibt. Mit diesem Verhältnis wird die entsprechende Komponente p_i^w gewichtet. Da der Viewport im Wertebereich $[-1,\ 1]$ liegt, er gibt sich aus der Bedingung, dass $q' \cdot \beta_i$ immer -1 bzw. 1 und q'_i immer > 1 ist. Dadurch sieht die Berechnung für jede Komponente wie folgt

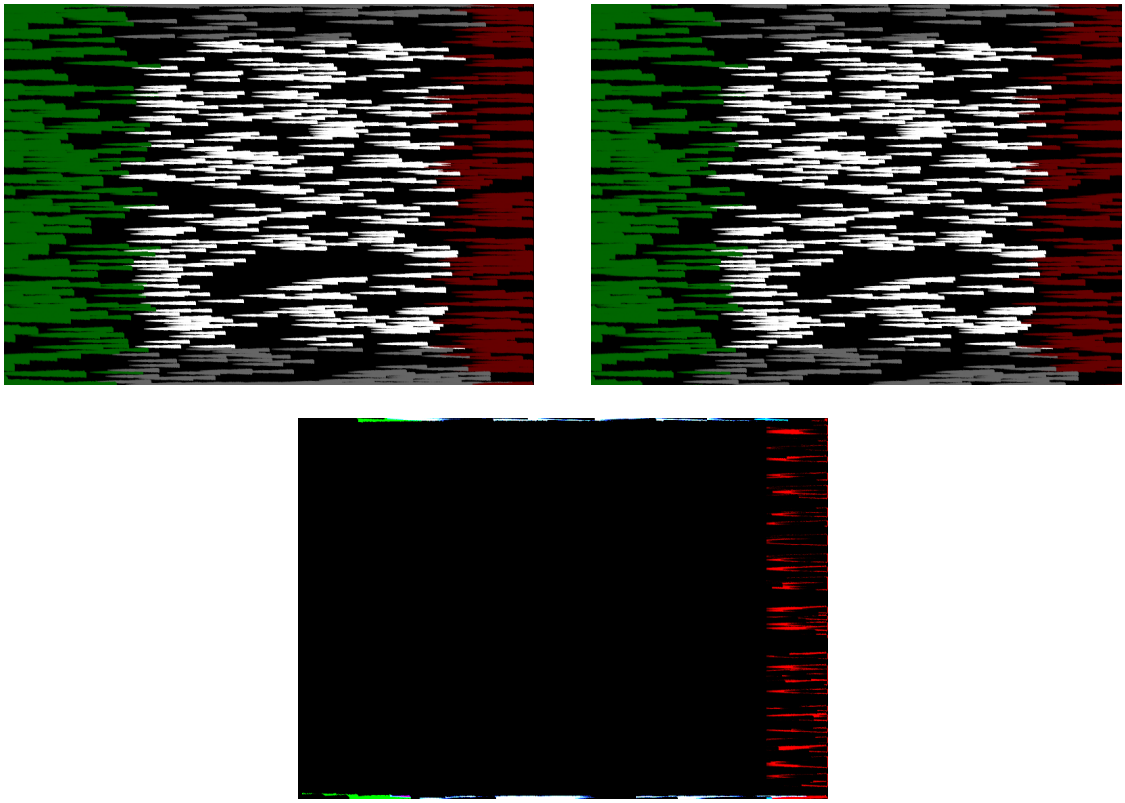

Abbildung 6.3.1.: Korrektur des Randlückenproblems.
links: Ausweitung des Wertebereichs der Seed Points von $[0,\ 1]$ auf $[-0.1,\ 1.1]$. *rechts*: Zusätzliche Verschiebung der Weltpunkte über den Viewportrand hinaus $\Rightarrow$ weniger Striche am Rand. *unten*: Differenzbild.

aus:

$$p_i^{w\,\prime} \quad = \quad p_i^w \cdot \frac{q_i'}{q_i' \cdot \beta_i} = p_i^w \cdot \frac{1}{\beta_i} = p_i^w \cdot \left|q_i'\right| = p_i^w \cdot |(q_i \cdot 2) - 1| \quad \text{mit}\, \beta_i = \left|\frac{1}{q_i'}\right|$$

$p_i^{w\,\prime}$ ist die neu berechnete und p_i^w die eigentliche Koordinate eines Punkts p^w.

Mit dieser Korrektur sind weder störende Lücken noch eine Überbevölkerung von Strichen zu beobachten. Das Resultat ist in Abbildung 6.3.1 auf der rechten Seite zu sehen. Im Vergleich mit dem linken Bild sind nun weniger Striche am rechten Rand sowie sehr schwach am unteren und oberen Bildrand zu sehen. Das untere Bild verdeutlicht die Unterschiede.

6.3.2. Generierung geeigneter Seed Points

Die Seed Point Generierung ist die Basis für gleichmäßig und zufällig verteilte Striche. Das einfachste Vorgehen ist eine große Anzahl an Punkten zufällig zu generieren. Die Generierung von Punkten ist ein Vorverarbeitungsschritt und sollte nicht während des Renderns geschehen.

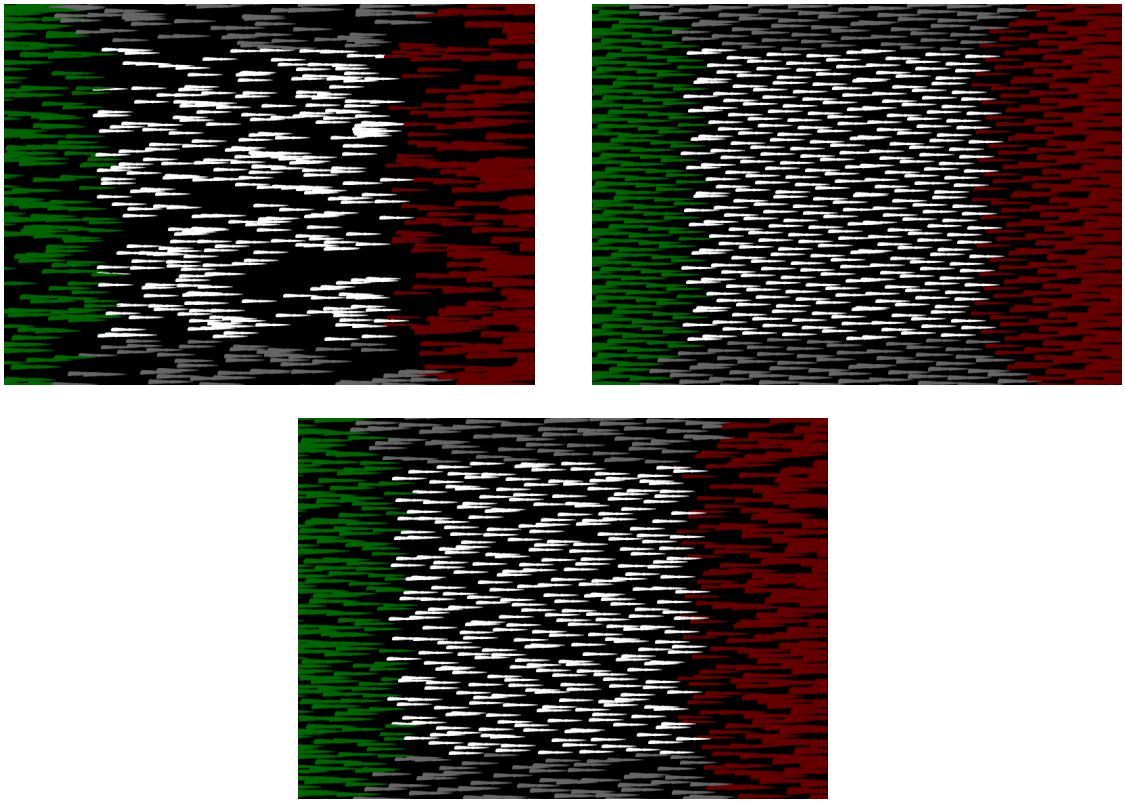

Abbildung 6.3.2.: Unterschiedliche Verteilungen von generierten Seed Points.
Von links nach rechts: Zufällig gewählte Punkte, Punkte durch die Hammersley-Sequenz und durch die Halton-Sequenz.

In Abbildung 6.2.2 ist zu beobachten, dass bei der Generierung von zufälligen Punkten zu viele offene Stellen existieren, an denen keine Striche generiert werden. Um den offenen Stellen entgegen zu wirken, sind zwei Generierungsverfahren für gleichverteilte Punkte getestet worden. Das erste untersuchte Verfahren benutzt die Hammersley-Sequenz und das zweite Verfahren die Halton Sequenz. In Unterunterabschnitt 6.3.2.3 wird ein Vergleich zwischen allen drei Generierungsverfahren gezogen, die die zufällig verteilten Punkte miteinbeziehen.

6.3.2.1. Hammersley Sequenz

Um eine Punktemenge nach Hammersley [Wei, Wika] im $\mathbb{R}^2$ aus dem Wertebereich $[0,\,1]\times[0,\,1]$ zu erlangen, werden alle Zahlen x_i mit $i = 0,\,1,\,\ldots,\,2^m - 1$ als Brüche $\frac{x_i}{2^m}$ in Fließkommadarstellung zur Basis 2 erzeugt. Für $m = 2$ ergeben sich die Zahlen

$$x_0 : 0.00_2 = 0, \quad x_1 : 0.01_2 = \tfrac{1}{4}, \quad x_2 : 0.10_2 = \tfrac{1}{2} \quad \text{und} \quad x_3 : 0.11_2 = \tfrac{3}{4}.$$

Um die korrespondierenden Zahlenpaare für jedes x_i zu erlangen, wird die Bit-Reihenfolge umgedreht, dargestellt mit $\overline{x_i}$.

$$\overline{x_0} : 0.00_2 = 0, \quad \overline{x_1} : 0.10_2 = \tfrac{1}{2}, \quad \overline{x_2} : 0.01_2 = \tfrac{1}{4} \quad \text{und} \quad \overline{x_3} : 0.11_2 = \tfrac{3}{4}$$

Jede Zahl x_i wird mit ihrer umgekehrten Bit-Reihenfolge zu einem Zahlenpaar zusammengefasst, der als Punkt interpretiert wird. Es ergeben sich die Punkte

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} 0,25 \\ 0,5 \end{pmatrix}, \quad \begin{pmatrix} 0,5 \\ 0,25 \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} 0,75 \\ 0,75 \end{pmatrix}.$$

6.3.2.2. Die Halton Sequenz

Die Halton Sequenz [Iow, Wikb] wird aus Zahlen zu einer gegebenen Basis b generiert. Als Basen werden Primzahlen verwendet. Es können beliebig viele Zahlen x_i generiert werden, für $i = 1, \ldots, n$. Jede Zahl x_i wird zur Basis b in umgekehrter Bit-Reihenfolge dargestellt und als Nachkommazahl einer Fließkommazahl interpretiert. Für $b = 2$ ergibt sich die Sequenz

$$x_1 : 0,100_2 = \tfrac{1}{2}, \quad x_2 : 0,010_2 = \tfrac{1}{4}, \quad x_3 : 0,110_2 = \tfrac{3}{4}, \quad x_4 : 0,001_2 = \tfrac{1}{8}, \ldots$$

Die weitergeführte Sequenz sieht in Brüchen folgendermaßen aus:

$$\tfrac{1}{2}, \tfrac{1}{4}, \tfrac{3}{4}, \tfrac{1}{8}, \tfrac{5}{8}, \tfrac{3}{8}, \tfrac{7}{8}, \tfrac{1}{16}, \tfrac{9}{16}, \tfrac{5}{16}, \tfrac{13}{16}, \ldots .$$

Für Punkte aus dem Raum $\mathbb{R}^2$ werden zwei Halton-Sequenzen zu unterschiedlichen Basen generiert. In dieser Arbeit wurde die Basis 2 für die x-Koordinate und die Basis 3 für die y-Koordinate verwendet.

6.3.2.3. Vergleich aller Punktgenerierungsverfahren

In Abbildung 6.3.2 ist eine Gegenüberstellung aller verwendeten Generierungsverfahren zu sehen. Das linke Bild ist mit zufällig erzeugten Punkten erstellt worden, das mittlere zeigt die Hammersley-Sequenz und das untere die Halton-Sequenz. Es ist zu beachten, dass ein Teil der Punktmenge sich außerhalb des Viewports befindet, s. Unterabschnitt 6.3.1. Die Anzahl beläuft sich auf 1024 erzeugte Punkte für die Hammersley-Sequenz und 1000 Punkte für die restlichen Verfahren.

Bei den zufällig generierten Punkten zeichnen sich deutlich Lücken ab. Diese Lücken haben zur Folge, dass an diesen Stellen keine Striche generiert werden.

Die Verteilung bei der Hammersley-Sequenz ist sehr regelmäßig und lässt wenig Lücken entstehen. Es entsteht ein regelmäßiges Muster, dass sehr starr und künstlich wirkt.

Die Halton-Sequenz zeigt ebenfalls eine gute Verteilung der Striche. Sie wirkt nicht so dicht aber auch nicht so regelmäßig wie die Verteilung der Hammersley-Sequenz.

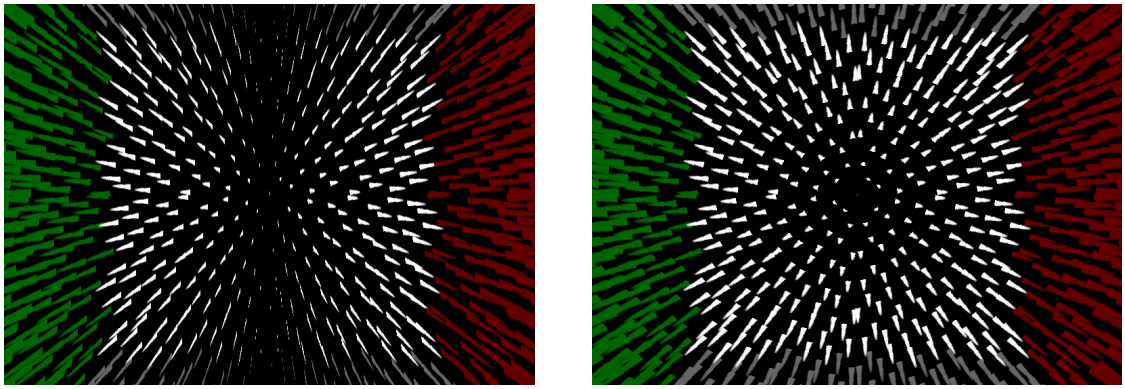

Abbildung 6.3.3.: Szene mit Cornel Box, wobei sich der Betrachter vorwärts bewegt.
links: Inkorrekt berechneter Hilfsvektor mit Hilfe eines Sichtvektors. *rechts*: Aus projiziertem Bewegungsvektor berechneter Hilfsvektor.
Anmerkung: Die Seed Points sind mit der Hammersley Sequenz erzeugt worden.

Die Erzeugung von Strichen soll zufällig wirken, damit das Bild nicht künstlich erscheint, sondern den Eindruck erweckt es sei gezeichnet oder gemalt worden. Ebenfalls müssen erzeugte Seed Points so regelmäßig verteilt sein, dass keine Lücken entstehen und so keine Bewegungsinformationen abhanden kommen. Diese Kriterien erfüllt die Halton-Sequenz.

6.3.3. Fehlende Normalen bei Bewegungen entlang der z-Achse

Bei Vorwärts- bzw. Rückwärtsbewegungen entlang der z-Achse ist zu vermuten, dass ebenfalls Striche erscheinen. Dies trifft allerdings momentan nicht zu. Striche werden bei Bewegungen entlang der z-Achse mit $\vec{h} = 0$ erzeugt und sind deshalb nicht zu sehen. Der Hilfvektor ist das Kreuzprodukt aus Sicht- und Bewegungsvektor. Aus dem Sichtvektor $\vec{e} = (0,\, 0,\, -1)^T$ und dem Bewegungsvektor $-\vec{v} = \vec{s} = (0,\, 0,\, -x)^T$ ergibt sich $\vec{s} \times \vec{e} = \vec{h} = 0$. Der gewählte Sichtvektor muss daher angepasst werden.

Der erste Lösungsansatz ist den Sichtvektor durch $\vec{e}_a = (1, 0, -1)^T$ zu ersetzen. Das Ergebnis ist im linken Bild der Abbildung 6.3.3 aufgeführt. Es zeigt sich, dass jetzt die Berechnung Striche erzeugt und somit der Hilfsvektor ungleich 0 ist, aber es bildet sich ein senkrechter Bereich in der Mitte des Bildes, in dem keine Striche zu sehen sind. Das Phänomen ergibt sich aus der Wahl des Vektors. Bei dem folgenden Rechenbeispiel verläuft der Bewegungsvektor entlang der z-Achse nach hinten. Der Sichtvektor liegt mit einem 45° Winkel am Bewegungsvektor an. Beide Vektoren liegen auf der Ebene, die von der x- und z-Achse aufgespannt wird. Daraus folgt, dass jeder aus dem Kreuzprodukt berechnete Vektor parallel zur y-Achse verläuft. Dadurch werden die Strichpolygone bei dieser Bewegung entlang der y-Achse aufgespannt. Ein auf der yz-Ebene liegendes Strichpolygon mit $x = 0$ wird zwar erzeugt, kann aber nicht vom Betrachter wahrgenommen werden, da es zu flach ist. Außerdem werden für alle Bewegungs-

vektoren keine Striche erzeugt, die eine Skalierung von $\vec{e}_a$ darstellen. Das Problem mit $\vec{h} = 0$ hat sich somit nur auf eine andere Bewegungsrichtung verschoben.

Anstatt einen weiteren Sichtvektor zu verwenden, wird der projizierte Bewegungsvektor $\vec{v}^p$ verwendet. Dafür werden die Anfangs- und Endpunkte des Bewegungsvektors mit der Projektionsmatrix multipliziert und dann durch die anschließende Homogenisierung projiziert. Die Differenz der beiden Punkte ergibt $\vec{v}^p$. Der Hilfsvektor $\vec{h}$ ergibt sich wie folgt:

$$\vec{h} = \begin{pmatrix} -\vec{v}_y^p \\ \vec{v}_x^p \\ 0 \end{pmatrix} \cdot \frac{1}{\left|\left|(-\vec{v}_y^p,\ \vec{v}_x^p,\ 0)^T\right|\right|} = \frac{\vec{v}^p \times \vec{e}}{||\vec{v}^p \times \vec{e}||}, \quad \text{wobei } \vec{e} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \text{ ist.}$$

In Abbildung 6.3.3 sind auf der rechten Seite Striche mit korrekt ausgerichteten Hilfsvektoren zu sehen.

Eine andere funktionierende Variante, die hier außer Acht gelassen wurde, interpretiert den Startpunkt p^w als Vektor und verwendet ihn als Sichtvektor $\vec{e}$.

6.3.4. Variierende Strichdicke in Abhängigkeit von der Tiefe

Beim Erzeugen von Strichpolygonen werden weiter entfernte Objekte mit kleineren Strichen versehen als näher liegende Objekte, s. Abbildung 6.3.4. Die Ursache liegt in der Abhängigkeit der Strichdicke von der projizierten Größe des Hilfsvektors. Da in weiter Ferne Projektionen erwartungsgemäß kleiner erscheinen, ist auch das Strichpolygon kleiner. Die unterschiedlichen Strichdicken der Projektion haben zur Folge, dass bei Objekten in weiter Ferne die Abstände zwischen den Strichen größer ausfallen. Dieser Effekt erweckt den Anschein, als ob Bewegungen in weiter Ferne weniger stark zur Geltung kommen.

Um die Abstände zwischen den Strichen zu füllen gibt es zwei Möglichkeiten:

1. Seed Points werden nicht im $\mathbb{R}^2$ sondern im $\mathbb{R}^3$ generiert. Dabei werden Punkte mit hinreichender Tiefe zur Generierung von Strichpolygonen hinzugenommen, ansonsten werden sie verworfen.
2. Bei der Berechnung der Strichdicke wird die Tiefe miteinbezogen, so dass der Hilfsvektor invariant zur Tiefe skaliert wird.

Vorschlag 1 Der Generierungsprozess muss daher angepasst werden um Punkte im Raum $\mathbb{R}^3$ zu erzeugen. Die hinzugewonnene Tiefe vom Seed Point q wird als Mindesttiefe interpretiert. D. h., wenn der ermittelte Punkt p anhand seiner projizierten Tiefe p_z^p noch vor q_z liegen sollte, dann wird die Generierung eines Strichpolygons abgebrochen.

Damit der Vergleich überhaupt durchführbar ist, müssen die *Near*- und *Far*-Werte der Projektionsmatrix bekannt sein bzw. eine Projektion durchgeführt werden. Außerdem muss beachtet

Abbildung 6.3.4.: Variierende Strichdicke bei unterschiedlich weit entfernten Dragon-Figuren, die sich von links nach rechts bewegt.
von links oben nach unten: Dragon nahe, große Striche; Dragon weit entfernt, kleine Striche; Dragon mit reduzierten Farben. Quellbild für die Farben vom Bild *links oben*.

werden, dass durch die Projektion die z-Werte nicht-linear skaliert werden und dabei Nebeneffekte auftreten können.

Um die entstandenen Lücken zwischen den Strichen zu füllen, müssen mehr Seed Points generiert werden. Mehr Seed Points bedeuten mehr Striche und belasten mit zunehmender Geometrie die Graphikkarte, wodurch die gesamte Geschwindigkeit der Anwendung sich reduziert. Dadurch würde nicht nur eine Abhängigkeit der Rendergeschwindigkeit zur Menge und Intensität der Bewegung bestehen, sondern auch zur Beschaffenheit des Tiefenbuffers.

Die variierende Strichdicke wird nicht beseitigt, aber auf diese Weise ist es möglich Striche in Abhängigkeit zur Tiefe skaliert zu zeichnen und weiter entfernte Objekte mit dichter liegenden Strichen zu zeichnen.

Vorschlag 2 Bei der Berechnung des Skalierungsfaktor für den Hilfsvektor wird die Tiefe miteinbezogen. Momentan wird der Hilfsvektor $\vec{h}$ mit einer festgelegten Dicke d skaliert. Es muss ein Faktor γ gefunden werden, der in Abhängigkeit von der Tiefe der Verkleinerung des Hilfsvektors entgegen wirkt und den bisherigen Skalierungsfaktor d ersetzt.

Bei dieser Variante wird der Strichkopf immer gleich groß gezeichnet, aber das zum Betrachter hin oder abgewandte Ende wird entsprechend größer bzw. kleiner projiziert.

Gegenüberstellung und Auswahl Der erste Vorschlag garantiert mehr Striche bei entfernten Objekten und füllt damit die Lücken, die durch größere Abstände entstehen. Die Striche werden aber nicht invariant zur Tiefe dargestellt und der Rendervorgang wird durch zusätzlich erzeugte Geometrie belastet.

Eine gleichbleibende Strichdicke, die unabhängig von der Tiefe ist, wird durch den zweiten Vorschlag erreicht. Sie führt zu einem unwesentlichen Mehraufwand im Geometry Shader.

Aufgrund dessen wird der zweite Vorschlag umgesetzt, damit der Rendervorgang nicht eine zusätzliche Abhängigkeit bekommt. Die Umsetzung wird im folgenden Abschnitt beschrieben.

6.3.5. Ermittlung des Skalierungsfaktors γ

Zur Berechnung eines tiefeninvarianten Skalierungfaktors γ muss geklärt werden wie er ermittelt werden kann. Dabei ist nicht offensichtlich, ob $\vec{h}$ *direkt* über die Projektionsmatrix P projiziert werden kann oder *indirekt* durch die Differenz der projizierten Punkte $p^p = h(P \cdot p)$ und $p^p_h = h(P \cdot (p + \vec{h}))$ ermittelt werden muss. Die Funktion h liefert die Homogenisierung eines Vektor, indem sie den Vektor durch seine homogene Koordinate teilt. Erst durch die Anwendung von h erfolgt die eigentliche Projektion. Bei dieser Betrachtung ist klar, dass das indirekte Vorgehen korrekt sein muss, da die Koordinaten auf der xy-Ebene im kanonischen Volumen ermittelt werden, die zur Rasterisierung verwendet werden.

Bei der weiteren Betrachtung ist es wichtig festzustellen, ob die *direkte* Projektion von $\vec{h}$ möglich ist. Dafür wird zuerst die Differenz $P \cdot (p + \vec{h}) - P \cdot p$ ermittelt und überprüft, ob sie den Werte von $P \cdot \vec{h}$ ergibt. Danach werden die Ergebnisse unter Anwendung der Funktion h betrachtet. Der sich ergebende Term für die indirekte Projektion ist $h(P \cdot (p + \vec{h})) - h(P \cdot p)$ und für die direkte Projektion $h\left(P \cdot \vec{h}\right)$.

Die Projektionsmatrix P, die in OpenGL verwendet wird, sieht wie folgt aus:

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} a & 0 & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & e & f \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Die Matrix P kann für die weiteren Rechnungen vereinfacht werden, indem sie substituiert wird. Aus der substituierten Matrix und den beiden zu transformierenden Vektoren ergibt sich:

$$
\begin{aligned}
P \cdot p &= \begin{pmatrix} p_x a + p_z b \\ p_y c + p_z d \\ p_z e + f \\ -p_z \end{pmatrix} \\
P \cdot \left(p + \vec{h}\right) &= \begin{pmatrix} \left(p_x + \vec{h}_x\right) a + \left(p_z + \vec{h}_z\right) b \\ \left(p_y + \vec{h}_y\right) c + \left(p_z + \vec{h}_z\right) d \\ \left(p_z + \vec{h}_z\right) e + f \\ -\left(p_z + \vec{h}_z\right) \end{pmatrix}
\end{aligned}
$$

Aus den sich ergebenden Vektoren wird nun die Differenz der beiden Vektoren errechnet.

$$
P \cdot \left(p + \vec{h}\right) - P \cdot p = \begin{pmatrix} \vec{h}_x a + \vec{h}_z b \\ \vec{h}_y c + \vec{h}_z d \\ \vec{h}_z e \\ -\vec{h}_z \end{pmatrix} = P \cdot \vec{h}
$$

Damit ist gezeigt, dass $\vec{h}$ direkt mit der Projektionsmatrix multipliziert werden kann. Es muss noch untersucht werden, ob die Terme $h(P \cdot \vec{h})$ und $h(P \cdot (p + \vec{h})) - h(P \cdot p)$ identische Ergebnisse liefern. Für die beiden Punkte und den Vektor sehen die Projektionen wie folgt aus:

$$
h\left(P \cdot p\right) = \begin{pmatrix} \frac{p_x}{-p_z} \cdot a - b \\ \frac{p_y}{-p_z} \cdot c - d \\ \frac{1}{-p_z} \cdot f - e \\ 1 \end{pmatrix}, \qquad h\left(P \cdot \left(p + \vec{h}\right)\right) = \begin{pmatrix} \frac{p_x + \vec{h}_x}{-p_z - \vec{h}_z} \cdot a - b \\ \frac{p_y + \vec{h}_y}{-p_z - \vec{h}_z} \cdot c - d \\ \frac{1}{-p_z - \vec{h}_z} \cdot f - e \\ 1 \end{pmatrix}
$$

$$
h\left(P \cdot \vec{h}\right) = \begin{pmatrix} \frac{\vec{h}_x}{\vec{h}_z} \cdot a - b \\ \frac{\vec{h}_y}{\vec{h}_z} \cdot c - d \\ -e \\ 1 \end{pmatrix}
$$

Mit den erhaltenen Vektoren muss nur noch gezeigt werden, dass $p_h^p - p^p \neq \vec{h}^p$ gilt.

$$
\begin{aligned}
h\left(P \cdot \left(p + \vec{h}\right)\right) - h\left(P \cdot p\right) &= \begin{pmatrix} \frac{p_x + \vec{h}_x}{-p_z - \vec{h}_z} \cdot a - b \\ \frac{p_y + \vec{h}_y}{-p_z - \vec{h}_z} \cdot c - d \\ \frac{1}{-p_z - \vec{h}_z} \cdot f - e \\ 1 \end{pmatrix} - \begin{pmatrix} \frac{p_x}{-p_z} \cdot a - b \\ \frac{p_y}{-p_z} \cdot c - d \\ \frac{1}{-p_z} \cdot f - e \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} \frac{p_x + \vec{h}_x}{-p_z - \vec{h}_z} \cdot a - \frac{p_x}{-p_z} \cdot a \\ \frac{p_y + \vec{h}_y}{-p_z - \vec{h}_z} \cdot c - \frac{p_y}{-p_z} \cdot c \\ \frac{1}{-p_z - \vec{h}_z} \cdot f - \frac{1}{-p_z} \cdot f \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} a \\ c \\ f \\ 0 \end{pmatrix} \cdot \begin{pmatrix} \frac{p_x + \vec{h}_x}{-p_z - \vec{h}_z} - \frac{p_x}{-p_z} \\ \frac{p_y + \vec{h}_y}{-p_z - \vec{h}_z} - \frac{p_y}{-p_z} \\ \frac{1}{-p_z - \vec{h}_z} - \frac{1}{-p_z} \\ 0 \end{pmatrix} \\
&\neq \begin{pmatrix} \frac{\vec{h}_x}{\vec{h}_z} \cdot a - b \\ \frac{\vec{h}_y}{\vec{h}_z} \cdot c - d \\ -e \\ 1 \end{pmatrix} = h\left(P \cdot \vec{h}\right) \quad \Rightarrow \textit{Wiederspruch}!
\end{aligned}
$$

Die Umformung zeigt, dass $p_h^p - p^p \neq \vec{h}^p$ ist. Dieser Umstand hat zur Folge, dass die projizierte Länge eines Vektors nur durch die Projektion seiner Endpunkte und die anschließende Berechnung der Differenz erfolgen kann.

Der Skalierungsfaktor γ wird folgendermaßen berechnet:

$$
\gamma = \frac{d}{\left|\left|\vec{h}' \cdot (1\,1\,0)^T\right|\right|}, \quad \text{mit } \vec{h}' = h\left(P \cdot \left(p + \vec{h}\right)\right) - h\left(P \cdot p\right)
$$

Zu beachten ist, dass nur die Länge in der Projektionsebene entscheidend ist und nicht die zusätzliche Tiefe, die in $\vec{h}'_z$ steckt. Aus diesem Grund wird $\vec{h}'$ mit dem Vektor $(1\,1\,0)^T$ multipliziert.

Anmerkung Aus dem Unterabschnitt 6.3.3 auf Seite 72 ist bekannt, dass der sich ergebende Hilfsvektor nur eine x- und y-Komponente besitzt und die z-Komponente gleich Null ist. Dadurch können weitere Vereinfachungen an den vorher berechneten Formeln durchgeführt

Abbildung 6.3.5.: Von der Tiefe unabhängige Strichdicke bei unterschiedlich weit entfernten Dragon-Figuren, die sich von links nach rechts bewegen.
links: Dragon nahe; *rechts*: Dragon weit entfernt.

werden.

$$P \cdot \vec{h} \quad = \quad \begin{pmatrix} \vec{h}_x\, a \\ \vec{h}_y\, c \\ 0 \\ 0 \end{pmatrix} \quad \text{und} \quad h\left(P \cdot \left(p + \vec{h}\right)\right) - h\left(P \cdot p\right) = \begin{pmatrix} -\frac{\vec{h}_x}{p_z}\, a \\ -\frac{\vec{h}_y}{p_z}\, c \\ 0 \\ 0 \end{pmatrix}$$

Weiter ist zu beachten, dass $\vec{h}$ nach der Multiplikation mit P nicht homogenisiert werden kann, da sonst eine Singularität auftritt. Außerdem zeigt die Vereinfachung deutlich, dass bei $P \cdot \vec{h}$ keine Tiefeninformationen enthalten sind und aus diesem Grund diese Berechnung für die Bestimmung von γ unzureichend ist.

Anwendung des Skalierungsfaktors γ Beim Vergleich der Abbildung 6.3.4 auf Seite 74 und den Ergebnissen in Abbildung 6.3.5 zeigt sich, dass die Korrektur mit γ funktioniert. Der Hilfsvektor wird nun mit dem Faktor γ anstatt mit der Dicke d skaliert.

6.3.6. Variierung der Seed Points Positionen

Damit der Startpunkt eines generierten Strichs variiert, müssen Seed Points verschoben werden. Es gibt verschiedenste Möglichkeiten einen Seed Points zu verschieben.

Verschiebung anhand eines Zufallsvektors Bei jedem Renderaufruf wird einen neuer Vektor $\vec{r}$ zufällig generiert. Dieser Vektor wird an den Shader weiterreicht und mit ihm jeder Seed Point verschoben. Es ist offensichtlich, dass alle Punkte immer in die gleiche Richtung verschoben werden.

Unterschiedliche Verschiebungsvektoren pro Seed Point Für den Fall, dass alle Seed Points in unterschiedliche Richtungen verschoben werden sollen gibt es eine weitere Variante. Dabei wird ebenfalls ein Vektor generiert und an den Shader weitergegeben. Außerdem wird eine Textur mit Zufallsvektoren bereitgestellt. Der verschobene Seed Point ermittelt zuerst seine Verschiebung mittels der Textur mit Zufallsvektoren und greift dann erst, nachdem er wieder verschoben wurde, auf die Geschwindigkeits- und Positionsdaten zu.

Beobachtungen Bei beiden Implementationen haben sich zwei entscheidende Einflussfaktoren herauskristallisiert. Der erste Faktor betrifft das Zeitintervall, indem sich $\vec{r}$ verändert. Der zweite Faktor betrifft die Stärke um die $\vec{r}$ den Seed Point verschiebt.

Ändert sich $\vec{r}$ bei jedem Renderaufruf, so wirken Bewegungen sehr unruhig und können evtl. sogar fehlinterpretiert werden. Kleine Verschiebungen von $\vec{r}$ werden kaum wahrgenommen, wobei große Verschiebungen den Betrachter auch verwirren können bzw. die Seed Points so stark verschoben werden, so dass wieder Randlücken wie in Unterabschnitt 6.3.1 auftreten. Damit zufällige Verschiebungen unterdrückt werden, könnte der Verschiebungsvektor als Punkt interpretiert werden, der innerhalb eines festgelegten Gültigkeitsbereichs ruhige, zufällige und auseinanderfolgende Bewegungen beschreibt. Dieser Vorschlag wurde nicht weiter verfolgt.

6.3.7. Zusammenfassung

Alle beobachtbaren Nebeneffekte sind analysiert und korrigiert worden. Durch die Korrekturen sind die Randlücken geschlossen, alle Striche quasi zufällig und gleichmäßig verteilt, alle Hilfsvektoren korrekt ausgerichtet und die Strichdicke invariant zur Tiefe festgelegt worden, so dass ein korrekter Eindruck von Bewegung erzielt wird. Der nächste Schritt verbindet die gerenderte Szene mit den gezeichneten Strichen.

6.4. Verschmelzung von Strichen und Szene

Da sichergestellt wurde, dass Striche ordnungsgemäß gezeichnet werden, kann nun die Szene mit der Strichzeichnung vereint werden um einen ersten Eindruck zu erlangen, wie die Striche zusammen mit den Objekten wirken und ob jeder Strich die richtig Lage hat.

Da die Szene vorab in eine Textur gerendert wurde, kann die Farbtextur als Grundlage zum Zeichnen verwendet werden. Dafür wird die Textur, die gleichzeitig ein Framebuffer ist, inklusive der Tiefe in den aktuellen Framebuffer durch die Funktion `glBlitFrameBuffer()` kopiert. OpenGL bietet dafür noch andere Funktionen bzw. Vorgehensweisen. Mehr Informationen dazu gibt es im Abschnitt A.4.

Beim Zeichnen der Striche ist der Tiefentest aktiviert, während das Schreiben der Tiefe nicht erlaubt ist. Ein Tiefentest dient dazu Strich korrekt in den Bildraum zu zeichnen. Das Schrei-

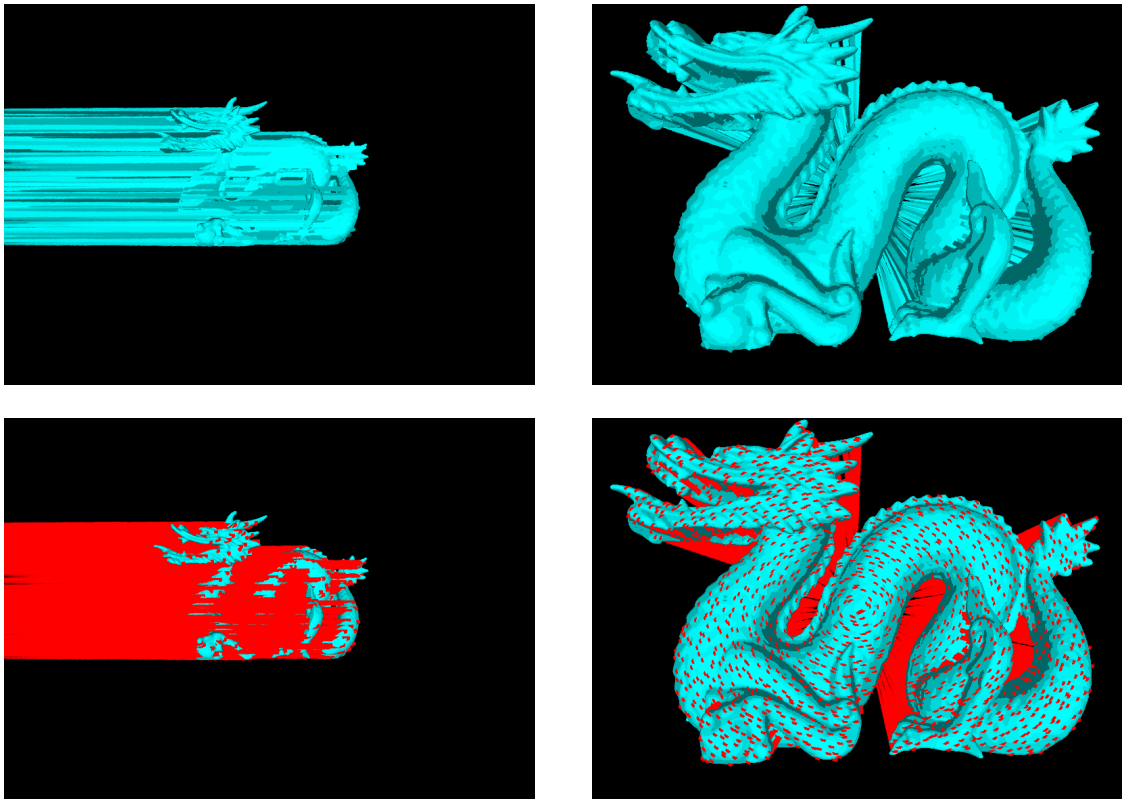

Abbildung 6.4.1.: Bewegter Dragon mit entsprechend den Quellfarben eingefärbten Strichen (*oben)* und rot eingefärbten Strichen (*unten)*.
links: von links nach rechts bewegend. *rechts*: auf den Betrachter hinbewegend.

ben der Tiefe ist deaktiviert, damit es später auch möglich ist, transparente Striche zu zeichnen. Dieses Vorgehen birgt die Gefahr, dass sich die Bewegungslinien eines weiterentfernten Objekts die eines näheren Objekts überlagern. Der hier beschrieben Effekt wird später in Unterabschnitt 6.5.1 näher erläutert.

Um das Schreiben der Tiefe zu deaktivieren, muss die Funktion `glDepthMask()` verwendet werden. Außerdem muss das Blenden von Quell- und Zielfarbe richtig einstellt werden, damit beim Blenden von transparenten Flächen keine Abdunklungen geschehen. Nähere Informationen dazu wurden schon in Abschnitt 6.2.4 auf Seite 67 gegeben.

6.4.1. Nebeneffekte

In den Bilder aus Abbildung 6.4.1 ist die Verschmelzung der Szene mit den generierten Strichen zu sehen. In allen Bildern ist der Dragon dargestellt. Die erzeugten Striche sind nicht transparent und der Tiefentest ist aktiviert, aber die Tiefe ist schreibgeschützt. Bei den Bilder auf der linken Seite bewegt sich der Dragon nach rechts, wobei er bei den rechten Bilder sich auf den

Betrachter zubewegt. Für die bessere Veranschaulichung sind alle Striche rot eingefärbt und die Bildausschnitte erneut dargestellt, siehe untere Bilderreihe.

Zu Erkennen ist ein unbeabsichtigter Effekt, der erzeugte Striche in Bewegungsrichtung aus dem Objekt herausragen lässt. Diese Überlagerungen können in der oberen Bilderreihe nicht gut erkannt werden, weshalb die Striche nochmals rot eingefärbt sind. Um die Korrektur dieser Überlagerungen kümmert sich der Unterabschnitt 6.4.2.

Da erzeugte Striche keine Mindestlänge besitzen, werden sie schon bei kleinsten Bewegungen gezeichnet. In Unterabschnitt 6.4.3 ändert sich zwar die Erzeugung des Strich, aber die Mindestlänge bleibt bestehen. Striche werden nach der Änderung der Stricherzeugung nur erst ab einer Mindestgeschwindigkeit gezeichnet. Bei der Überschreitung der Mindestgeschwindigkeit, werden die Striche atok dargestellt. Unterabschnitt 6.4.4 beschäftigt sich mit dem Übergang von nicht existierenden zu existierenden Strichen, um weiche Einblendungen zu gewährleisten.

6.4.2. Unbeabsichtigte Überlagerung von Strichen mit der Szene

Diese unbeabsichtigte Überlagerungen äußern sich indem der Strichanfang aus dem Objekt herausragt. Besonders offensichtlich ist dieser unwollte Effekt bei den Objektgrenzen zu beobachten und ist selbst bei den oberen beiden Bilder zu erkennen. Folgende Vorschläge könnten umgesetzt werden.

Gültigkeit eines Strichs bestimmen Es könnten Gültigkeitsflächen definiert werden, auf denen Striche generiert werden dürfen. Dazu könnte das Skalarprodukt aus der Oberflächennormalen und dem Bewegungsvektor der Objekte verwendet werden. Wenn das Skalarprodukt negativ werden würde, würde ein Strich generiert sonst nicht.

Strichverschiebung Der nächster Vorschlag ist, Strich nur entlang der negativen Bewegungsrichtung zu verschieben. Der Strich würde auf diese Weise weiter hinten generiert und würde nicht mehr aus dem Objekt herausragen.

Änderung der Strichdefinition Beim letzte Vorschlag wird die vorherige Strichdefinition vom Unterabschnitt 6.2.2 auf Seite 65 verändern. Momentan wird der Strecke eines Strichs ein Strichanfang und Strichende hinzugefügt. Daraus ergibt sich ein Strich, der länger ist als die gegebene Geschwindigkeit. Anstatt den Strichanfang und das Strichende zur Strecke hinzuzufügen, werden sie als Bestandteil des Strichs definiert.

Bei besonders stark gekrümmten Objekten wird es vorkommen, dass sich dennoch Artefakte bilden. Sie entstehen auf Oberflächen, deren Normale zur Bewegungsrichtung zeigt.

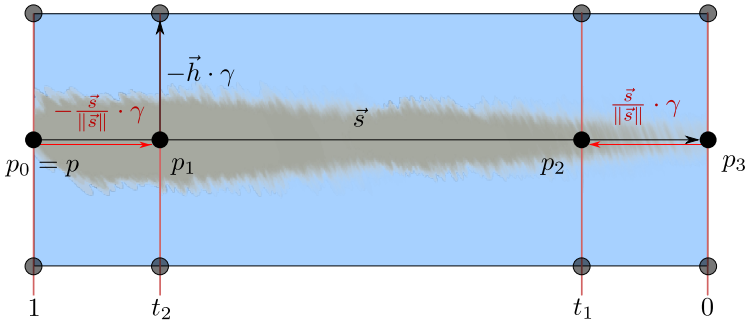

Abbildung 6.4.2.: Diese schematische Zeichnung zeigt, wie ein Strichpolygon konstruiert werden kann. Die grauen Punkte sind die Eckpunkte des zu konstruierenden Polygons. Eine Beispieltextur wird als Hintergrund verwendet. Die Zeichnung beinhaltet Veränderungen zur Abbildung 6.2.1 auf Seite 65.

Abwägung der Vorschläge Der Vorschlag mit dem Gültigkeitsbereich ist viel versprechend, würde aber bedeuten, dass für ein großes Objekt mit einem kleinen sichbaren Gültigkeitsbereich wenige Striche generiert werden.

Der Zweite verhindert die aus dem Objekt ragenden Striche und garantiert mehr Striche für die Bewegung als der erste Vorschlag.

Ähnlich zum zweiten Lösungsvorschlag wird beim Dritten durch das Korrigieren der Strichdefinition der Strich etwas nach hinten verschoben. Es kann dennoch vorkommen, dass sich Artefakte auf den Objektoberflächen bilden.

Für das weitere Vorgehen wurde sich für eine Verbindung aus den beiden letzten Vorschläge entschieden. Sie garantieren in Kombination, dass der Strich nicht länger gezeichnet wird als er sollte, und verhindern zusätzlich die Artefaktentstehung auf Flächen, die entlang der Bewegungsrichtung nach vorne zeigen.

6.4.3. Umsetzung der neuen Strichdefinition und der Verschiebung

Es werden zuerst die neue Strichdefinition umgesetzt und die Ergebnisse betrachtet. Danach werden die Verschiebung der Striche hinzugefügt und erneut die Resultate verglichen.

Die neue Konstruktion eines Strichs ist in Abbildung 6.4.2 illustriert. Der Punkt p_0 ist dabei der Anfang der Strecke und $p_3 = p_0 + \vec{s}$ das Ende. Der Rumpf beginnt wie vorher ab dem Punkt $p_1 = p_0 + \frac{\vec{s}}{|\vec{s}|} \cdot \gamma$ und endet mit dem Punkt $p_2 = p_3 - \frac{\vec{s}}{|\vec{s}|} \cdot \gamma$, die auf der Strecke zwischen $\overline{p_0 p_3}$ liegen. Der Skalierungsfaktor γ wurde diesmal anstatt d verwendet, damit garantiert werden kann, dass der Anfang und das Ende des Strichs immer korrekt im Verhältnis zur Strichdicke dargestellt werden. Das Polygon wird wie gewohnt vom skalierten Hilfsvektor aufgespannt.

Anzumerken ist, dass ein Strich, der länger ist als die beiden Linienenden, inkorrekte Darstellungen zur Folge hat. Für eine korrekte Darstellung muss die Bedingung $|\vec{s}| > 2\gamma$ erfüllt sein.

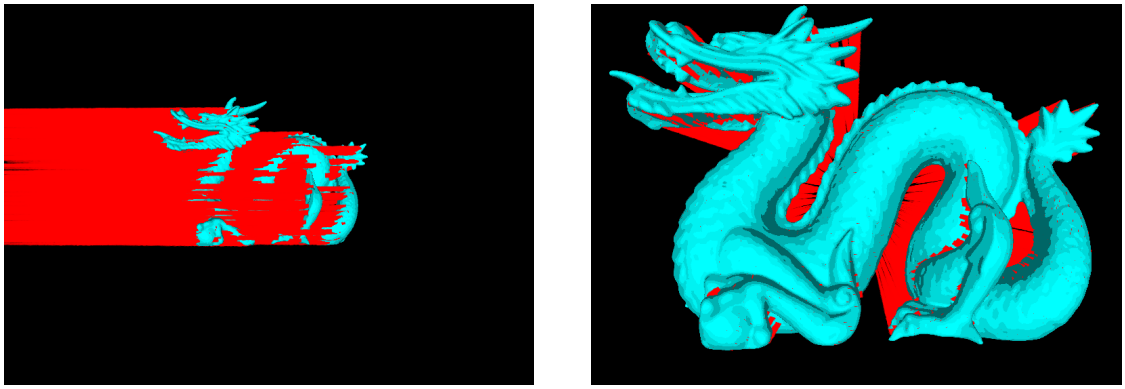

Abbildung 6.4.3.: Bewegter Dragon mit rot eingefärbten Strichen. Die Erzeugung der Striche hat sich geändert, weshalb weniger Artefakte zu erkennen sind.
links: von links nach rechts bewegend. *rechts*: auf den Betrachter hinbewegend.

Die Umsetzung führt zu geringeren Artefakten, wie in Abbildung 6.4.3 im linken und rechten Bild festzustellen ist. Es bilden sich nur noch sehr wenige Artefakte aus, deshalb ist eine stärker Verschiebung der Striche notwendig.

Die neue Anfangsposition p_0 ist die verschobene Position p und ergibt sich aus $p_0 = p - \frac{\vec{s}}{||\vec{s}||} \cdot \delta$, wobei der Skalierungfaktor δ genauer bestimmt werden muss. Wenn der tiefeninvariante Skalierungsfaktor γ für δ verwendet wird, wird der Strich bei weit entfernten sowie nahen Objekt im gleichen projizierten Abstand zum Ursprungspunkt entstehen. Wenn stattdessen die Dicke d verwendet wird, wird der Strich in weiter Entfernung auf der projizierten Ebene deutlich näher an seinem Ursprung liegen und stärker verschoben sein als bei nahen Objekten. Um sich ein bessere Bild von den Auswirkungen zu machen wurden beide Verfahren angewandt.

Um einen besseren Vergleich zu erlangen, sind mehrere Bilder angefertigt worden, die in Abbildung 6.4.4 aufgeführt sind. Es wurde diesmal eine türkise Sphäre als Vergleichsobjekt verwendet, die sich von links nach rechts bewegt. Die linken Bilder sind dabei mit dem Skalierungsfaktor γ erstellt worden und die rechten mit d. Die obere Reihe zeigt die Sphäre aus sehr nahen Perspektive, die mittlere Reihe aus einem mittleren Abstand und die untere Reihe aus einem weiten Abstand. Bei der Verwendung von γ treten besonders bei nahen Objekten deutlich Artefakte auf, was im linken oberen Bild zu beobachtet ist. Bei weiter entfernten Objekten sind stattdessen keine störenden Strichereste zu erkennen. Bei der Dicke d sind die Beobachtungen genau umgekehrt. Es stellt sich heraus, dass bei den Faktoren die Ergebnisse nicht vollkommen zufriedenstellend sind.

Da die auftretenden Überlappungen von Strichanfängen mit Objekten kaum sichbar sind, solange die Striche die gleiche Farbe erhalten wie ihr Ursprungspunkt, kann dieser Mangel hingenommen werden. Bei der Entscheidung zwischen den beiden zur Verfügung stehenden Faktoren hängt die Wahl entscheidend von der Verwendung, den Strichtexturen und der dargestellten Szene ab. Anstatt sich für einen Skalierungsfaktor zu entscheiden, kann das Maximum

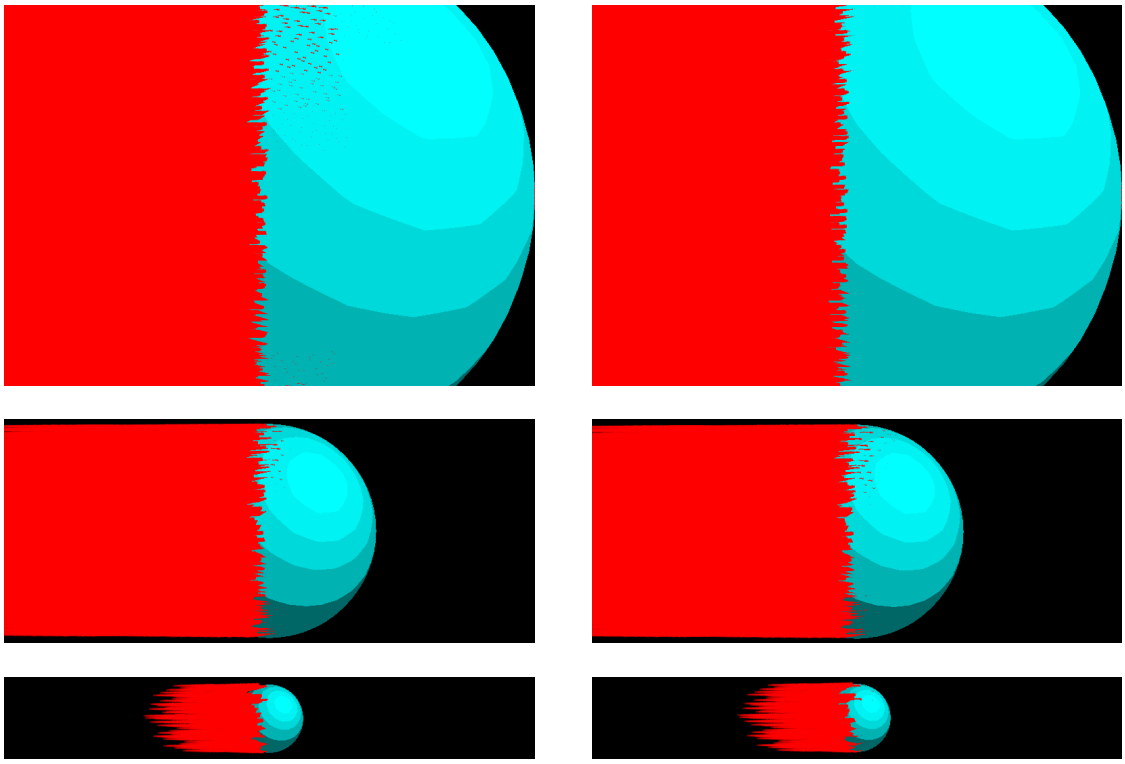

Abbildung 6.4.4.: Bewegte Sphäre mit rot eingefärbten Strichen. Um Faktor γ (*links*) bzw. d (*rechts*) verschobene Striche.
oben: sehr nahe betrachtet; *mitte*: aus mittlerer Entfernung; *unten*: weit entfernt.

$max\,(\gamma,\ d)$ benutzt werden. In dieser Arbeit wurde sich für den Skalierungsfaktor d entschieden, weil die Überlappungen bei kleiner bzw. entfernten Objekten kaum zu beobachten sind und bei weiter entfernten Objekten die Striche näher am Objekt bleiben.

6.4.4. Weiche Einblendung von Strichen

Durch die festgelegt Mindestlänge 2γ eines Strichs in Abbildung 6.4.3 auf Seite 82 erscheinen die Striche erst, wenn sie sich mit einer Geschwindigkeit $> 2\gamma$ fortbewegen. Dieser Umstand führt zu abrupten Einblendungen. Das abrupte Einblenden der Striche wirkt störend und tritt besonders stark bei Geschwindigkeiten auf, die nahe am Schwellwert 2γ liegen.

Für einen fließenden Übergang zwischen der Abwesenheit und der plötzlichen Existenz eines Strichs, wird die an den Fragment Shader übergebene Farbe geändert. Genauer gesagt muss eigentlich nur der Alphawert verändert werden, aber wie in Abschnitt 6.2.4 auf Seite 67 angemerkt ist, sind die Farben in der Textur mit dem Alphawert verrechnet worden. Aus dem Grund wurde die Blend-Funktion angepasst, weshalb auch jede Komponente der verwendeten Farbe mit dem errechneten Blend-Faktor multipliziert werden muss.

Der Blend-Faktor wird durch die Funktion $f\ :\ \mathbb{R}^3\ \rightarrow\ [0,\ 1]$ berechnet. Sie berechnet den richtigen Blend-Faktor, wenn sich der eingehende Geschwindigkeitsvektor $\vec{v}^w\ \in\ \mathbb{R}^3$ und $||\vec{v}^w|| \in [a,\ b]$ ist, wobei $a, b \in \mathbb{R}$ und $0 \leq a \leq b$ gilt. Ist die Geschwindigkeit $|\vec{v}| < a$, wird $f = 0$ und ist $|\vec{v}| > b$ erfüllt, wird $f = 1$. Daraus ergibt sich die Funktion:

$$f:\mathbb{R}^3 \ \rightarrow\ [0,\ 1] = \begin{cases} 0 & |\vec{v}| < a \\ \frac{|\vec{v}|-a}{b-a} & |\vec{v}| \in [a,\ b] \\ 1 & |\vec{v}| > b \end{cases}$$

Die Variablen a und b können frei gewählt werden und könnten als Uniform-Variablen an den Geometry-Shader übergeben werden. Sie sollten dennoch die zusätzliche Bedingung $2\gamma \leq a \leq b$ erfüllen. Wird diese Bedingung nicht erfüllt, werden die Striche zu klein und nicht richtig dargestellt. Wenn $a < 2\gamma$ gilt, ist die Strichlänge zu gering und $\vec{s}$ muss wie folgt angepasst werden, damit der Strich korrekt dargestellt wird:

$$\vec{s}' \ = \ \frac{\vec{v}}{|\vec{v}|} \cdot 2\alpha.$$

Auf diese Weise wird für jeden Strich eine Mindestlänge garantiert, die aber bei einem zu klein gewählten a zu viel zu großen Strichen bei geringer Bewegung führt und diese nicht der Länge der Bewegung entsprechen. Die resultierende Darstellung könnte den Betrachter irritieren, aber wird durch die Transparenz des Striches gelindert.

Abbildung 6.5.1.: *links*: Zwei ineinander bewegende Kugeln, von denen nur ihre Speed Lines zu sehen sind. Der Tiefentest ist aktiviert und es treten Artefakte auf.
rechts: Der entsprechende Tiefenbuffer zum linken Bild.

6.5. Überblendung von bewegten Objekten und Speed Lines

Bis jetzt zeichnet die Umsetzung verschiedene Striche und kann sie auch mit der Szene verschmelzen. Jetzt fehlt nur noch, dass das bewegte Objekt sich langsam ausblendet, nur noch die Striche an deren Stelle zu sehen sind und so Schlieren erzeugt. Damit die Hintergrundszene, die bewegten Objekte und die Speed Lines richtig miteinander kombiniert werden können, muss jeweils ein Farbbuffer mit Tiefeninformationen vorliegen.

6.5.1. Tiefeninformation für Striche

Die Tiefe für die Hintergrundszene und die bewegten Objekte zu erhalten ist trivial und wird von OpenGL schon zur Verfügung gestellt. Die Tiefe der Striche zu erhalten ist ein komplexeres Problem. Wenn der Tiefentest eingeschaltet wird, kann es zu gegenseitigen Verdeckungen der Striche führen. Besonders weil ein Strich nichts anderes als ein Polygon mit einer Textur ist, erscheint jeder Strich im Tiefenbuffer als Rechteck, vgl. rechtes Bild in Abbildung 6.5.1. Mit eingeschalteten Tiefentest bedeutet das leider, dass nicht alle Striche richtig zeichnet werden und besonders an transparenten Stellen weiter hinten liegende Striche evtl. nicht mehr gezeichnet werden, s. linkes Bild in Abbildung 6.5.1.

Dieser Mangel kann gelindert werden, indem jedes Fragment verworfen wird, dass vollkommen transparent ist. Die Ergebnisse sind in Abbildung 6.5.2 zu erkennen. Dennoch sind immer noch Artefakte zu erkennen.

Diese Problem könnte gelöst werden, wenn die Striche nach ihrer Tiefe sortiert werden könnten, aber da sie im Geometry-Shader erzeugt werden, kann keine Sortierung durchgeführt werden. Ein anderer Lösungsvorschlag wäre mit *Order-Independent Transparency* zu arbeiten, indem mit einem Multisampler und dem Stencilbuffer pro Pixel mehrere Farbwerte gespeichert wer-

Abbildung 6.5.2.: *links*: Zwei sich ineinander bewegende Kugeln, von denen nur ihre Speed Lines zu sehen sind. Der Tiefentest ist aktiviert und es werden vollkommen transparente Fragmente verworfen. Es treten leichte Artefakte an den Strichrändern auf.
rechts: Der entsprechende Tiefenbuffer zum linken Bild.

Abbildung 6.5.3.: *links*: Zwei ineinander bewegende Kugeln, von denen nur ihre Speed Lines zu sehen sind. Es sind keine Artefakte zu sehen.
rechts: Der entsprechende Tiefenbuffer zum linken Bild.

den und in einem weiteren Durchlauf schließlich sortiert und miteinander geblendet werden, siehe hierzu [MB07b, MB07a]. Dieser Lösungvorschlag wurde nicht umgesetzt.

In Abschnitt 6.4 wurde die Entscheidung getroffen, dass der zuletzt gezeichnete Strich immer alle vorherigen Striche überzeichnet. Dafür füllt der erste Renderdurchlauf den Tiefenbuffer und der zweite zeichnet die Striche mit ausgeschaltetem Tiefentest.

Durch die Möglichkeit einer Feedback Loop - die aber laut der OpenGL Spezifikation zu undefinierten Ergebnisse führen kann [SA10] - kann dieser Prozess in nur einem Zeichenvorgang erledigt werden. Um eine Feedback Loop zu nutzen, benötigt der Fragment-Shader seinen eigenen Tiefenbuffer als Textur. Für jedes Fragment wird immer die aktuelle Farbe ausgegeben und die vorderste Tiefe in die Variable `gl_Depth` geschrieben. Dabei ist zu beachten, dass für jeden Ausführungspfad im Fragment-Programm diese Variable gesetzt sein muss.

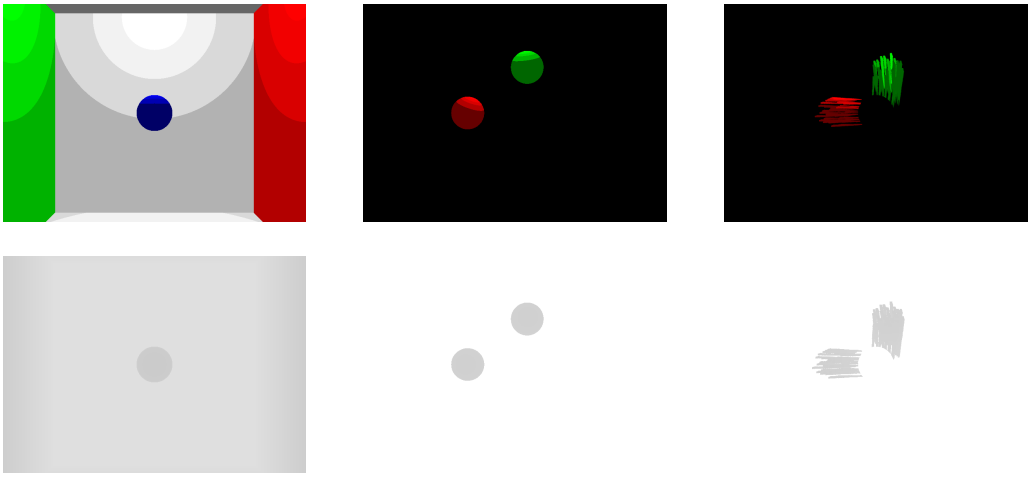

Abbildung 6.5.4.: Eine Szene aufgeteilt in drei Farbbuffer (*links*) und ihre entsprechende Tiefe (*rechts*). Es ist zu sehen der Hintergrund der Szene (*oben*), die sich bewegenden Objekte (*mitte*) und die generierten Speed Lines (*unten*), die einen Tiefenvergleich mit der Hintergrundszene unterzogen wurden.

Durch Verwendung der Feedback Loop ist es möglich in nur einem Durchlauf den Farbbuffer und die vorderste Tiefe zu erhalten. Ergebnisse dieses Prozesses sind in Abbildung 6.5.3 zu sehen. Es ist anzumerken, dass die sich hier überlagernden Striche keine korrekte Darstellung abbilden, aber würde ein Künstler ein Strich vollziehen, würde dieser ebenfalls alle vorherigen Striche überzeichnen. Außerdem ist dieser Effekt nur für sehr kurze Zeit zu sehen und wird damit nicht auffällt.

6.5.2. Verschmelzung

Nun liegen allen nötigen Farbbuffer mit ihrer Tiefe vor, um sie miteinander zu verschmelzen, s. Abbildung 6.5.4. Im nächsten Schritt werden die Farbbuffer miteinander kombiniert, indem sie anhand ihrer Tiefe sortiert werden und dann von hinten nach vorne ineinander übergeblendet werden. Dabei ist zu beachten, dass die Farben evtl. schon mit dem Alphawert vormultipliziert sind, je nachdem welche Blendfunktion verwendet wurde.

Die Kombination aller sortierten Farbbuffer ergibt die Szene im linken Bild in Abbildung 6.5.5. Unter Anwendung einer Gewichtung, können bewegte Objekte ab einer gewissen Geschwindigkeit ausgeblendet werden. Hierfür wird die projizierte Geschwindigkeit und die schon beschriebene Gewichtungsfunktion aus Unterabschnitt 6.4.4 benötigt. Ein Beispiel ist im rechten Bild in Abbildung 6.5.5 zu erkennen.

6.5.3. Beobachtungen

Die Abbildung 6.5.6 zeigt eine Cornel Box, die immer stärker nach links verschoben wird und der Hintergrund sich immer stärker ausblendet. Hierbei ist unschöner Effekt zu erkennen. Weil

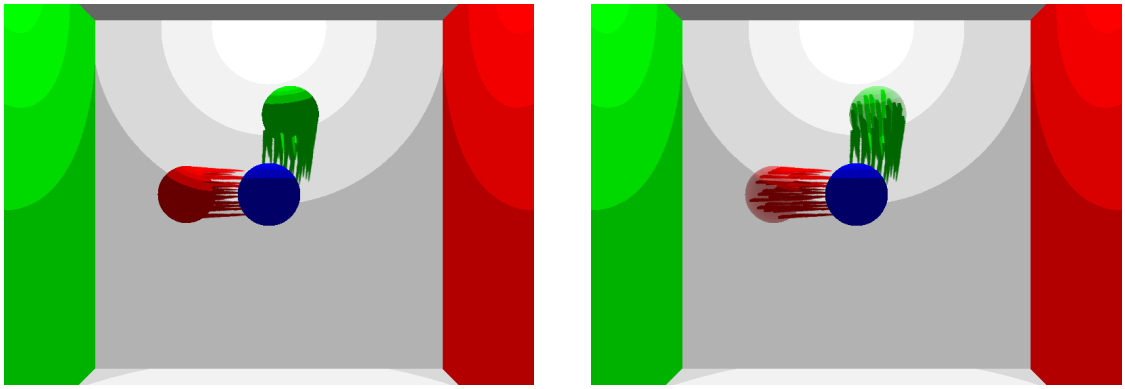

Abbildung 6.5.5.: Die Kombination alle Farbbuffer ergibt eine Szene mit rotierenden Kugeln (*links*). Die Kugeln können unter Verwendung einer Gewichtungsfunktion leicht ausgeblendet werden (*rechts*).

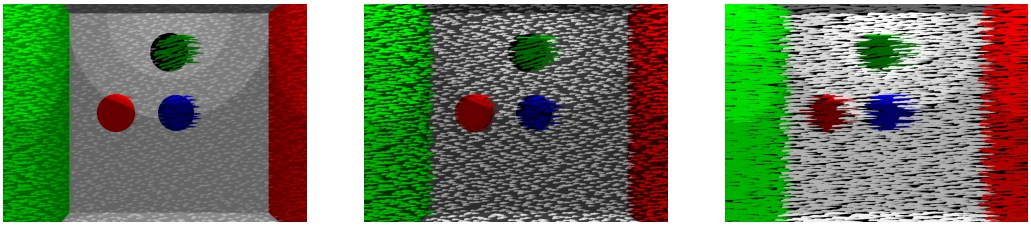

Abbildung 6.5.6.: Eine bewegte Szene, die von *links nach rechts* gesehen immer stärker nach links verschoben wird. Es ist eine immer stärkere Ausblendung der gesamten Szene zu beobachten, bis nur noch Striche zu sehen sind.

die Cornel Box auf einen schwarzen Hintergrund gezeichnet wurde, wirkt das Bild zuerst hell und wird dann zeitweise insgesamt dunkler und sobald die Striche eine gewisse Mindestlänge überschritten haben, wieder heller. Aus dieser Beobachtung ist zu schließen, dass der hierdurch erzielte Effekt eventuell nur auf bestimmt Objekte angewandt werden sollte.

Außerdem fällt bei genauerer Betrachtung des linken Bilds in Abbildung 6.5.6 auf, dass die grüne Kugel kaum erkennbar und eher schwarz wirkt. Beim erwarteten Effekt sollte die Kugel wenigstens über den Hintergrund, hier die Cornel Box, geblenden werden. In der Szene bewegen sich alle Objekte, weshalb kein Hintergrund existiert, mit dem geblendet werden könnte. Aus diesem Grund wird die Kugel zunehmend dunkler.

Der beschriebene Effekt führt immer dann zu Artefakten, wenn sich bewegende Objekte überlagern und ausgeblendet werden sollen. So können schon bei leichten Bewegungen Artefakte entstehen, die Objekte z.B. schwarz wirken lassen. Die Schlussfolgerung ist, dass das Verfahren besonders beim Ausblenden keine guten Ergebnisse liefert. Der Effekt könnte abgeschwächt werden, wenn er nur auf eine Gruppe von ausgewählten Objekten angewandt wird. Diese Schlussfolgerung könnte auch bedeuten, dass die so zu rendernden Objekte nicht zwingend in bewegte und unbewegte Objekte eingeteilt werden müssten, was den ersten Durchlauf stark vereinfacht.

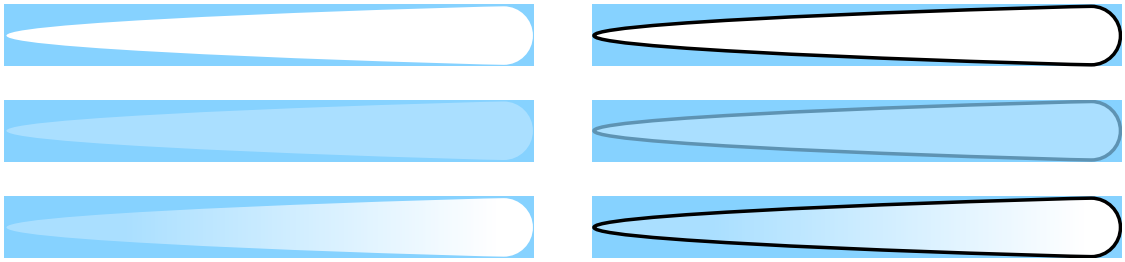

Abbildung 6.6.1.: Verschiedene Stricharten. *links*: Striche ohne Umrandung; *rechts*: Striche mit Umrandung.
oben: weiße Striche; *mitte*: transparente Striche; *unten*: Farbverlauf von links transparent zu recht weiß.

Ein Lösungsvorschlag, der im Verlauf dieser Arbeit erst sehr spät entstand, sieht folgendermaßen aus. Statt das Ausblenden des bewegten Objekt erst im dritten Durchlauf durchzuführen könnte das Ausblenden schon im ersten Durchlauf geschehen. Die Unterscheidung zwischen bewegten und unbewegten Objekten würde wegfallen. Die Szene müsste von hinten nach vorne gerendert werden oder ein so genannter A-Buffer bzw. K-Buffer zum Einsatz kommen, der *Order-Independent Transparency* ermöglicht, vgl. [MB07b, MB07a]. Damit die Speed Lines aber ihre richtige Farbe erhalten, muss zusätzlich ein Farbbuffer erzeugt werden, der Szene ohne das Ausblenden von starkbewegten Objekten zeigt.

6.6. Verschiedene Stricharten

Dieser Ansatz benutzt als Grundlage Polygone, damit eine Textur verwendet werden kann. Alle bisher gegebenen Beispiele benutzen die Strichtextur, die in der schematischen Strichzeichnung in Abbildung 6.4.2 auf Seite 82 angedeutet ist.

Jede Strichtextur gibt Form und Farbe vor. Durch die Angabe von t_1 und t_2 wird der Strichanfang und das Strichende in der Textur festgelegt, wodurch ebenfalls die Form beeinflusst wird. Bei der Rasterisierung eines Strich, erhält der Fragment-Shader die Farbe des Ursprungspunkts, der durch den Seed Point ermittelt wurde. Durch die Multiplikation der Ursprungsfarbe und der aus der Strichtextur ermittelten Farbe ergibt sich die endgültige Farbe. Diese Umstände bieten ein hohes Maß an Flexibilität, weil es sehr einfach ist, die Strichtexturen auszutauschen und die Variablen t_1 und t_2 im Geometry-Shader festzulegen.

Um beispielhaft unterschiedliche Arten von Strichen und ihre Anwendung darzustellen, sind die in Abbildung 6.6.1 aufgeführten Striche angewendet worden. Es sind sechs verschiedene Stricharten aufgeführt, dabei gibt es komplett weiße, halbtransparente und solche mit einen Farbverlauf von transparent nach weiß. Auf der rechten Seite besitzen alle Striche zusätzlich eine Umrandung. Die daraus resultierenden Bilder sind in Abbildung 6.6.2 gegenübergestellt entsprechend den aufgeführten Stricharten angeordnet.

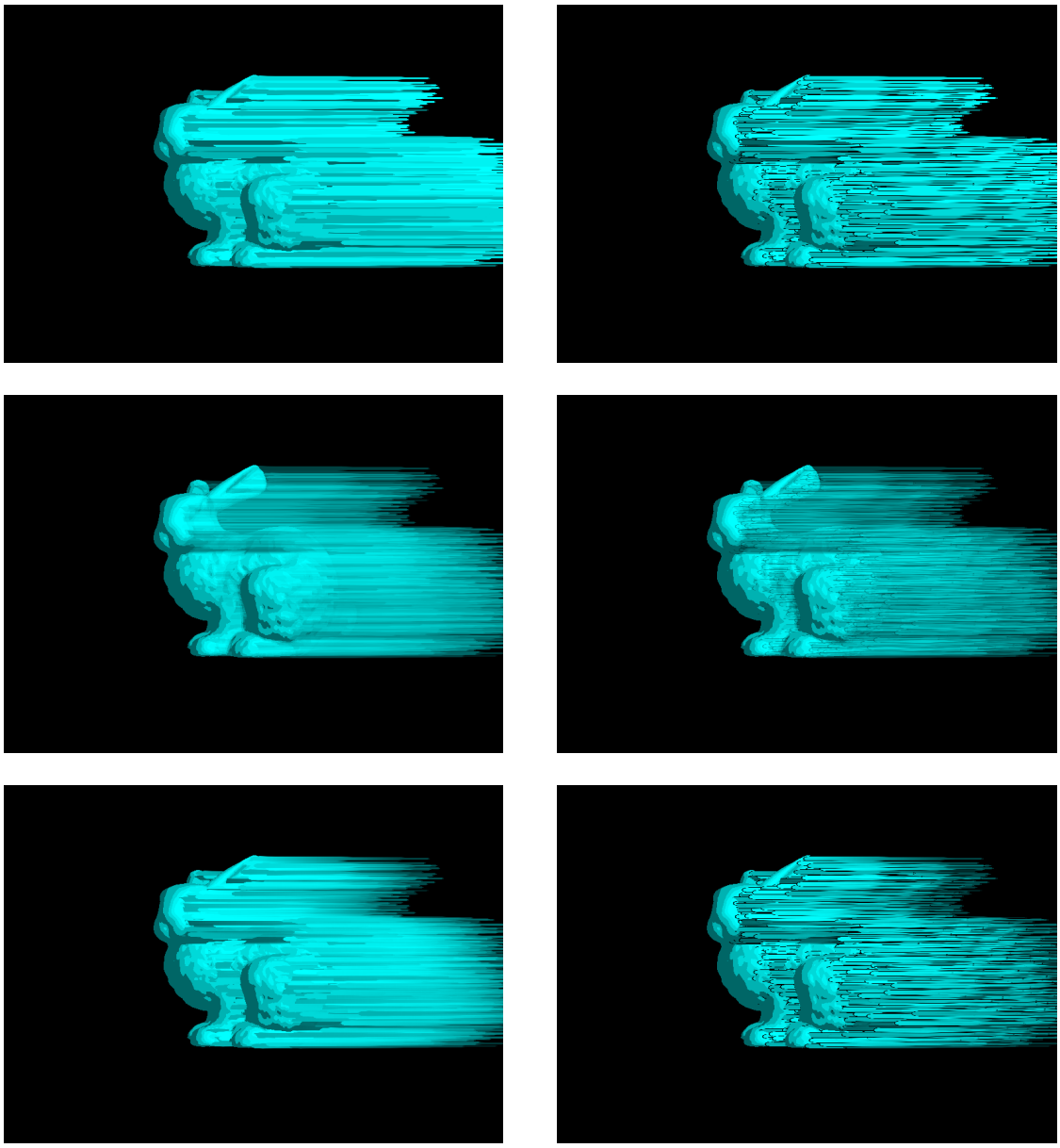

Abbildung 6.6.2.: Verschiedene auf ein nach links verschobenes Hasenmodell angewandte Stricharten. Die Anordnung entspricht der in Abbildung 6.6.1.

Wie aus den gerenderten Bilder zu erkennen ist, kann die Darstellung der Striche stark durch die Textur beeinflusst werden. Besonders schöne Effekte können mit transparenten Strichtexturen erzielt werden.

6.7. Zusammenfassung

Der verfolgte Ansatz wurde erfolgreich umgesetzt und erzeugt Schlieren. Die dazu verwendeten Speed Lines erzeugt die Umsetzung mit relativ geringem Aufwand auf beliebiger Geometrie. Es kann sowohl die Strichdicke definiert werden als auch auf das Aussehen der Striche über eine Textur Einfluss genommen werden. Es ist möglich eine Mindestgeschwindigkeit zu definieren, ab der die Striche anfangen sich langsam einzublenden. Das Gleiche gilt für das Ausblenden von bewegten Objekten.

Das folgende Kapitel 7, fasst nochmal kurz zusammen, wie das komplette Verfahren mit allen Korrekturen implementiert ist.

7. Implementation

Nachdem die vorherigen Kapiteln zwei Ansätze besprochen haben, beschreibt dieses Kapitel die Implementation des vielversprechenden zweiten Ansatzes. Es stellt das endgültige Verfahren vor und geht genauer auf die einzelnen Durchläufe ein. Zur Beschreibung jedes Durchlaufs verwendet es verkürzte Auszüge aus dem Programmcode, der in GLSL geschrieben ist. Auf der beiliegenden CD[1] sind alle Programmcodebeispiele in vollem Umfang enthalten.

Die Implementation ist mit Java geschrieben und verwendet die Core API von OpenGL 3.0 zur Ansteuerung der Graphikkarte. JOGL[2] ist die Graphikbibliothek in Java, die Bindings zu allen OpenGL Versionen bereitstellt.

7.1. Beschreibung des Verfahrens

Das Verfahren wird im Rahmen dieser Arbeit als *Streaking* bezeichnet und nähert bewegte Objekte durch Schlieren an, die aus Speed Lines bestehen. Das bewegte Objekt wird durch Speed Lines in Abhängigkeit zu seiner Geschwindigkeit dargestellt.

Das Verfahren erzeugt Speed Lines durch zufällig über den Bildraum verteilte Seed Points. Diese zufälligen Punkte werden über die Halton-Sequenz erzeugt und lesen an den Position im Bildraum Informationen aus vorgerenderten G-Buffern aus.

7.1.1. Voraussetzung

Die Voraussetzungen für *Streaking* sind programmierbare Shader, besonders der Geometry-Shader, die Speicherung der vorherigen ModelView-Matrix und Multi Render Targets. Außerdem muss es möglich sein Frame Buffer zu verwenden, die mit höheren Bit-Genauigkeiten als 8-Bit pro Farbkanal arbeiten können und als Buffer Texturen verwenden werden können.

7.1.2. Aufbau

Das Verfahren beinhaltet drei Renderdurchläufe und folgt den Ablauf, der im Aktivitätsdiagramm in Abbildung 7.1.1 skizziert ist. Den Kern des Verfahrens bilden die verwendeten Shaderprogramme.

[1] Die Codebeispiele befinden sich im Verzeichnis */listings* auf der CD.
[2] Java Bindings for OpenGL zu finden unter `http://jogamp.org`, zuletzt besucht am 03.01.2010.

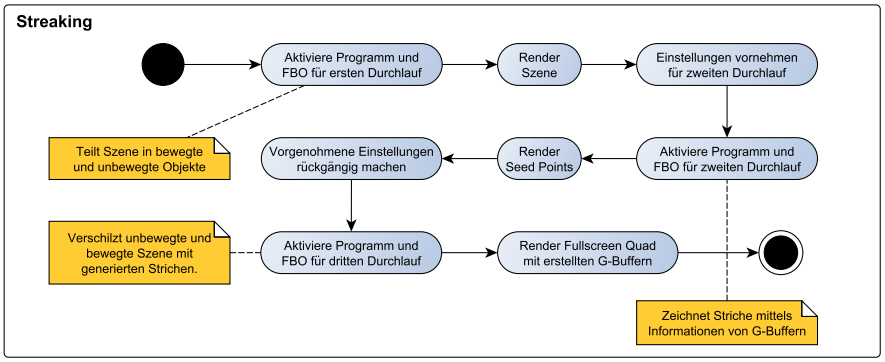

Abbildung 7.1.1.: Ablaufdiagramm der einzelnen Renderdurchläufe um Streaking zu erreichen.

Vor dem ersten Renderdurchlauf wird das entsprechende Shaderprogramm und das FBO gesetzt, welches die zu rendernde Szene in bewegte und unbewegte Objekte aufteilt. Es speichert die jeweilige Farbe, Tiefe, Position und Geschwindigkeit aus Sicht der Kamera und Geschwindigkeit im Bildraum in Texturen ab.

Der zweite Durchlauf hat als Voraussetzung, das Face Culling deaktiviert, das Blending aktiviert und die Vergleichsfunktion für den Tiefentest auf kleiner gleich ändert wird. Damit ist sichergestellt, dass jeder Strich sichtbar, auch wenn Oberflächennormale vom Betrachter wegweist. Das aktivierte Blending lässt transparente Striche zu und der angepasste Tiefentest ermöglicht es die Tiefe der Striche zu ermitteln und sie gleichzeitig ohne Artefakte zu zeichnen. Danach lädt die folgende Aktivität das entsprechende Programm mit FBO und bindet die zuvor gerenderten Texturen an das Programm. Anschließend zeichnet das Programm die Seed Points. Dieser zweite Renderdurchlauf verwendet die angebundenen Texturen und eingehenden Seed Points um Striche zu erzeugen und speichert ihre Farbe sowie ihre Tiefe ebenfalls in Texturen ab. Nach dem Durchlauf werden alle Zustände wieder zurückgesetzt, die zuvor geändert wurden.

Abschließend kombiniert der letzte Durchlauf alle Farbbilder der bewegten und unbewegten Szene sowie der erzeugten Striche miteinander. Damit dies geschehen kann wird ein bildschirmfüllender Quad gezeichnet und die entsprechenden Farb- und Tiefenbuffer angebunden. Dabei wird außerdem festgelegt, bei welcher Geschwindigkeit die bewegten Objekte ausgeblendet werden. Das Verfahren trifft diese Entscheidung pro Pixel.

7.2. Erster Durchlauf: Rendern von bewegten und unbewegten Objekten

Der erste Durchlauf teilt die Szene in bewegte und unbewegte Objekte auf und schreibt diese in verschiedene Ebenen eines Framebuffer, um so für beide Ergebnisse jeweils einen Tiefenbuffer

in nur einen Durchlauf zu erhalten. Als Alternative könnte die Trennung der Objekte auch durch zweimaliges Rendern erfolgen.

Der Vertex-Shader, der in Listing 7.1 aufgeführt ist, benötigt Informationen wie die Position und die derzeitige und vorherige ModelView-Matrix, hier als `modelView` und `previousModelView` bezeichnet. Die Position `element.position` wird mit beiden Matrizen transformiert und so die Geschwindigkeit `velocity` in der Welt als auch `projectedVelocity` im Bildraum berechnet. Der Vertex-Shader reicht die berechnete Geschwindigkeit und ihre Anfangs- und Endpunkte an den Geometry-Shader weiter und schreibt die derzeitige projizierte Position in `gl_Position`. Außerdem wird auch die Normale zur Berechnung der Farbe weitergegeben[3].

```
1  void main() {
2     // Applying modelview matrix
3     position = modelView * element.position;
4     currentProjectedPosition = projection * position;
5
6     // Applying previous modelview matrix
7     vec4 previousPosition = previousModelView * element.position;
8     previousProjectedPosition = projection * previousPosition;
9
10    // Calculating the velocity vector in model view and screen space
11    velocity = (position - previousPosition).xyz;
12    projectedVelocity = ((currentProjectedPosition /
          currentProjectedPosition.w) - (previousProjectedPosition /
          previousProjectedPosition.w)).xyz;
13
14    ...
15
16    gl_Position = currentProjectedPosition;
17 }
```

Listing 7.1: Vertex-Programm zum Berechnen der Geschwindigkeit.

Der Geometry-Shader, den Listing 7.2 enthält, entscheidet zunächst mit der Methode `isMoving()`, ob die übergebenen Geometrie sich bewegt. Dies geschieht, indem der Shader für jeden Vertex des eingehenden Primitives die Länge des Geschwindigkeitsvektors `velocityLength` mit einem vordefinierten Schwellwert `factor` vergleicht. Wenn sich nur ein Punkt bewegt hat, gibt die Methode `isMoving()` den Wert `true` zurück. Das bedeutet die Variable `gl_Layer` wird für unbewegte Geometrie auf `0` gesetzt und sonst auf `1`, wodurch es möglich ist die Szene zu trennen. Die dann für jeden Vertex aufgerufene Methode

[3] Im Kontext eines anderen System können für die Berechnung der Farbe evtl. weitere Information benötigt werden.

`setVertexAttributes()` setzt die restlichen Attribute für jeden Vertex und übergibt den fertigen Vertex an den Fragment-Shader weiter. Indem der Geometry-Programm die Entscheidung, in welche Ebene der Fragment-Shader anschließend schreiben soll, für das gesamte Primitiv trifft, stellt er sicher, dass Vertices nicht in verschiedene Ebenen geschrieben werden und pro Ebene zu wenig Vertices vorhanden sind, um ein Primitiv zu bilden. Andernfalls wäre das auftretende Verhalten undefiniert. Der Geometry-Shader übergibt die Farbe, die Normale, die Position und die Geschwindigkeit aus Sicht der Kamera und die vorherigen und jetzige projizierte Position an den Fragment-Shader.

```
1  void main() {
2     // Determine if this primitive has moved and sets the correct Layer
3     gl_Layer = isMoving() ? 1 : 0;
4
5     foreach vertex {
6        // Set Attributes
7        setVertexAttributes(index);
8        // Emit a new vertex with all set output attribute
9        EmitVertex();
10    }
11 }
12
13 void setVertexAttributes(int index){
14    gl_Position = gl_in[index].gl_Position;
15    ...
16 }
```

Listing 7.2: Das Geometry-Programm zerteilt die Szene in bewegte und unbewegte Objekte.

Der Fragment-Shader berechnet wie üblich die Farbe der Objekte und schreibt alle eingehenden Werte in Texturen. Die Berechnung der Farbe folgt der Farbreduktion des Cartoon-Shading. In den Fragment-Shader eingehende Attribute sind zwischen drei Punkten interpoliert, was zu falschen Werten bei der Geschwindigkeit im Bildraums führt. Dieses Problem hat schon Unterabschnitt 5.3.2 behandelt. Die Geschwindigkeit berechnet sich deswegen aus den übergebenen projizierten Punkten der vorherigen und jetzigen Position. Das Fragment-Programm speichert die Farbe, die Tiefe, die Position und Geschwindigkeit, jeweils aus Sicht der Kamera, und die Geschwindigkeit im Bildraum in Texturen.

7.3. Zweiter Durchlauf: Erzeugung der Striche

Der zweite Durchlauf konstruiert die Striche aus den Werten der vorgerenderten G-Buffern und zeichnet diese mit der vordersten Tiefe in zwei weitere G-Buffer. Der Vertex-Shader reicht dabei die eingehende Position an den Geometry-Shader weiter.

Der Geometry-Shader erhält zusätzlich zur Position noch die vorgerenderten G-Buffer aus dem vorherigen Durchlauf. Sie beinhalten Informationen über die Farbe, die Position und die Geschwindigkeit aus Sicht der Kamera. Aus diesen Informationen werden die Striche konstruiert.

Listing 7.3 zeigt, dass in Zeile 5 die eingehende Position als Texturkoordinate interpretiert wird und damit Information wie Position, Geschwindigkeit und Farbe an den entsprechenden Stellen ausgelesen werden durch die Methoden `getPosition()`, `getVelocity()` und `setColor()`. Die Methode `setColor()` liest die Farbe aus und speichert sie in einer globalen Variable. Die Position `position` und Geschwindigkeit `velocity` werden an die Methode `generateStroke()` übergeben.

```
1  vec4 color;
2
3  void main (void){
4     // Retrieve texture coordinate
5     vec3 coords = gl_in[0].gl_Position.xyz;
6
7     // Read position, velocity and color from attached G-buffers
8     vec4 position = getPosition(coords);
9     vec4 velocity = getVelocity(coords);
10    color = getColor(coords);
11
12    // Generate a stroke
13    generateStroke(position, position - velocity);
14 }
```

Listing 7.3: Die `main()` Methode des Geometry-Programms ließt alle Informationen aus dem G-Buffer aus und lässt durch die Methode `generateStroke` einen Strich erzeugen.

Die Methode `generateStroke()` aus Listing 7.4 implementiert das beschriebene Verhalten aus Unterabschnitt 6.4.3. Sie berechnet zuerst die Strecke `s` und deren Projektion `sProjected` aus den projizierten Punkten `p0` und `p3`. Um das Strichpolygon aufzuspannen wird der Vektor `h` benötigt, der sich aus dem Kreuzprodukt des Streckenvektor `s` und dem Sichtvektor `point0` ergibt und obendrein normalisiert wird. Um eine tiefenunabhängige Strichdicke zu erlangen, ist die projizierten Länge von `h` erforderlich. Mit `point0` als Anfangspunkt und dem Vektor `h`, wird der projizierte Endpunkt `ph` ermittelt. Aus den vorhandenen Werten wird in Zeile 17 der tiefenunabhängige Skalierungfaktor `scaleFactorGamma` berechnet. Danach wird die Sichtbarkeit des Strichs über dessen projizierte Länge ermittelt. Für den Fall, dass die sich ergebene Farbe vollkommen transparent ist, bricht der Geometry-Shader seine Arbeit ab. Abschließend wird aus den berechneten Werten das Polygon erzeugt.

```
1  void generateStroke(in vec4 point0, in vec4 point3) {
```

```
    // Calculate line vector s⃗ with its length.
    vec4 s = point3 - point0;
    float sLength = length(s);

    // Project the start and endpoint
    vec4 p0 = project(point0);
    vec4 p3 = project(point3);

    // Calculate projected line vector s⃗^p
    vec4 sProjected = p3 - p0;
    float sProjectedLength = length(sProjected.xy);

    // Calculation of the depth invariant scaling factor γ
    vec4 h = vec4(normalize(cross(s.xyz, point0.xyz)), 0);
    vec4 ph = project(point0 + h);
    float scaleFactorGamma = thickness / length((ph - p0).xy);
    h *= scaleFactorGamma;

    // Determines the blending factor for the whole stroke
    currentColor.rgba *= getBlendFactor(sProjectedLength);
    if(currentColor.a < 0.0001){
        return;
    }

    ...
}
```

Listing 7.4: Die Methode `generateStroke()` erzeugt die Strichpolygone.

Die Berechnung des Blendfaktors ist in Listing 7.5 aufgeführt. Damit eine korrekt Berechnung durchgeführt werden kann, überprüft die erste Verzweigung, ob der Wert der Variable `beginFading` kleiner gleich `endFading` ist. Sie werden als Uniform-Variablen an den Geometry-Shader übergeben. Die Variable `beginFading` beschreibt die Mindestlänge, die die projizierte Strecke erfüllen muss, damit der Strich zu sehen ist. Hingegen steht `endFading` für die Länge, aber der Strich voll sichtbar ist. Der zweite Teil der Methode berechnet den Blendfaktor und garantiert, dass der Wert im Wertebereich $[0, 1]$ bleibt.

```
float getBlendFactor(float velocity){
    float difference = endFading - beginFading;

    // Checks for valid values
    if(difference <= 0){
        return 1.0;
    }
```

```
8
9     // Calulates the blending factor and ensures, that it stays in the
          range between 0 and 1
10    float factor = (velocity - beginFading) / difference;
11    return clamp(factor, 0.0, 1.0);
12 }
```

Listing 7.5: Die Implementation der Methode `getBlendFactor()` des GeometryProgramms.

Das Fragment-Programm zeichnet schließlich die Striche, indem die eingehende Farbe mit der Farbe der Strichtextur multipliziert wird. Die eingebaute Texture Feedback Loop ermöglicht es die Tiefe des vordersten Punkts zu ermitteln.

Listing 7.6 zeigt wie das Fragment-Programm zuerst die Farbe `fragColor` berechnet und das jedes Fragment außer Acht gelassen wird, bei der der Alphawert von `fragColor` gleich 0 ist. Der anderen Teil des Fragment-Programms überprüft zuerst, ob das zu zeichnende Fragment nicht von der Hintergrundszene verdeckt wird und verwirft es bei Bedarf. Außerdem ermittelt es den derzeitigen und gespeichert Tiefenwert `gl_FragCoord.x` und `storedDepth` und schreibt den kleinsten Wert von beiden in `gl_Depth`. Beim Auslesen von `storedDepths` entsteht die besagt Texture Feedback Loop. Das Fragment-Programm speichert die Farbe und die Tiefe in Texturen.

```
1  void main (void) {
2     // Calculates the final color of the stroke and discards it, in case
          of full transparency
3     fragColor = texture(strokeTexture, textureCoordinate) * color;
4     if(fragColor.a == 0){
5        discard;
6     }
7
8     // Calculates the right texture coordinate to retrieve depth values
9     vec2 fragCoord = gl_FragCoord.xy / textureSize(sceneDepths, 0).xy;
10    vec4 texCoord = vec4(fragCoord, 0, 1);
11    float sceneDepth = texture(sceneDepths, texCoord);
12
13    // Discards fragments, which are hiden by the background scene
14    if(sceneDepth < gl_FragCoord.z){
15       discard;
16    }
17
18    // Always sets the lowest depth value of the current or fragment
          depth as gl_Depth
19    float storedDepth = texture(storedDepths, texCoord.xyw);
20    gl_FragDepth = (storedDepth < gl_FragCoord.z) ? storedDepth :
          gl_FragCoord.z;
```

```
21 }
```

Listing 7.6: Das Fragment-Programm kombiniert die eingehende Farbe mit der Farbe der Strichtextur. Außerdem wird über eine Texture Feedback Loop die kleinste Tiefe gespeichert.

7.4. Dritter Durchlauf: Kombination der Striche mit der Szene

Im letzten Durchlauf werden alle bisherigen Farbbuffer mit ihrer Tiefe ausgelesen und miteinander kombiniert. Dabei wird die anhand der Tiefe die Farbwerte sortiert. Der Vertex-Shader führt keine Berechnungen durch, sondern zeichnet nur einen Quad, der über den gesamten Viewport gelegt wird.

```
1  void main(void){
2     // Read all color and depth values into arrays
3     retrieveValues();
4     // Apply a fading factor to the moving geometry
5     applyFadeFactor();
6     // Sorts all values, so that the farthest color comes first and the
          nearest color is last.
7     sort();
8
9     // Loop over all colors
10    for(int i = 0; i < 3; i++){
11       // Blends the color at position i with the current color
12       blendWithCurrentColor(colors[i]);
13    }
14 }
```

Listing 7.7: Das Fragment-Programm liest alle Farben mit der Tiefe aus, sortiert sie und blendet sie ineinander über.

Das Fragment-Programm aus Listing 7.7 liest mit der Methode `retrieveValues()` zuerst alle Farbbuffer mit ihrer Tiefe aus und schreibt ihre Werte in Arrays. Die Methode `applyFadeFactor()` wendet einen Blendfaktor auf den Farbwert der bewegten Geometrie an, sodass sie ausgeblendet werden kann. Es ist die gleiche Funktion, die schon in Listing 7.5 gezeigt wurde. Der Unterschied ist nur, dass jetzt der Umkehrwert berechnet wird. Anschließend werden die Arrays mit `sort()` entsprechend ihrer Tiefe absteigend sortiert. Dabei bildet die Farbe und die Tiefe ein zu sortierendes Element. Die darauffolgende Schleife bewirkt mit der Methode `blendWithCurrentColor()`, dass alle Farbwerte additiv ineinander übergeblendet werden.

7.5. Zusammenfassung

Dieses Kapitel hat umfassend die Implementation des Verfahrens *Streaking* gezeigt und den Ablauf demonstriert. Die Implementation setzt das Verfahren auf einer gängigen Graphikkarte in Echtzeit um. Erste Ergebnisse der Implementation sind schon in Kapitel 6 und werden im folgenden Kapitel 8 näher analysiert.

8. Ergebnisse

Die hier geschilderten Ergebnisse beziehen sich auf das entwickelte Verfahren aus dem zweiten Ansatz, der im Kapitel 6 vorgestellt wurde. Das Kapitel schildert zunächst Beobachtungen, die der Benutzer während der Ausführung des entstandenen Programms macht. Hieran koppelt der Abschnitt Messungen der mittleren Ausführungsgeschwindigkeit. Sodann analysiert das Kapitel die Vor- und Nachteile von Streaking, sowie beobachtete Fehlerfälle und deren Ursachen. Zum Schluss wird das Verfahren anhand der gestellten Anforderungen bewertet und in den Kontext bereits bekannter Verfahren eingeordnet.

8.1. Beobachtungen

Das umgesetzte Verfahren *Streaking* erzeugt viele Speed Lines, die Objekte und ihre Bewegung andeuten. Das Verfahren kennzeichnet jedes ausreichend schnell bewegte Objekt mit ihnen und ersetzt ab einer gewissen Geschwindigkeit bewegte Objekte durch sie. Die Speed Lines erhalten die Farbe, die an ihrem Startpunkt vorliegt, und deuten durch ihre Ausrichtung und Länge die zurückgelegte Strecke des Objekts an. Die dargestellte Speed Line beschreibt eine Gerade von einem Punkt an seiner neuen Position zu seiner vorherigen Position. Speed Lines sind stilisiert dargestellt. Die Größe der Strichenden ist an die Strichdicke gekoppelt, wobei die Strichdicke wiederum unabhängig von der Tiefenposition ist.

Alle Speed Lines werden im dreidimensionalen Raum gezeichnet und verdecken oder überdecken entsprechend ihrer räumlichen Lage Objekte. Ihre Darstellung ist perspektivisch angepasst und deutet jede Bewegung an. Speed Lines treten nicht mit ihrer Spitze an der anderen Objektseite heraus. Stattdessen wirkt es auf den Benutzer als liegen sie an einem bewegten Objekt an und folgen diesem in all seinen Bewegungen. Wenn sie exakt auf Sichtvektoren liegen, sind Speed Lines unsichtbar, weil das ausgerichtete Polygon nicht genügend Fläche bietet um dargestellt zu werden. Das Verfahren zeichnet Speed Lines aufeinander ohne einen Tiefentest durchzuführen.

Die Anzahl der erzeugten Speed Lines hängt von der Anzahl der Seed Points und der Fläche ab, die das Objekt einnimmt. Seed Points sind gleichmäßig aber nicht regelmäßig über die Bildebene verteilt. Die Positionierung der Speed Lines wirkt natürlich. Durch die gleichmäßige Verteilung entstehen keine sichtbaren Lücken zwischen den Speed Lines, sodass innerhalb eines definierbaren Umkreises mindestens eine Speed Line existiert. Am Bildschirmrand treten

Seed Point Anzahl	Bewe-gungs-stärke	FPS	Gesamt-zeit	1. Durchlauf		2. Durchlauf		3. Durchlauf		Restzeit	
30000	stark	29,42	34,0 ms	9,4 ms	27,65%	22,7 ms	66,82%	1,8 ms	5,22%	0,1 ms	0,31%
30000	mittel	41,39	24,2 ms	9,6 ms	39,72%	12,7 ms	52,53%	1,8 ms	7,32%	0,1 ms	0,43%
30000	leicht	65,27	15,3 ms	9,6 ms	62,44%	3,9 ms	25,41%	1,8 ms	11,52%	0,1 ms	0,64%
30000	N/A	69,89	14,3 ms	9,6 ms	67,32%	2,8 ms	19,44%	1,8 ms	12,53%	0,1 ms	0,71%
10000	stark	50,31	19,9 ms	9,7 ms	48,90%	8,2 ms	41,37%	1,8 ms	9,18%	0,1 ms	0,55%
10000	mittel	62,24	16,1 ms	9,5 ms	59,29%	4,7 ms	29,35%	1,7 ms	10,81%	0,1 ms	0,56%
10000	leicht	71,33	14,0 ms	10,2 ms	72,89%	1,8 ms	12,76%	1,8 ms	13,11%	0,2 ms	1,24%
10000	N/A	79,13	12,6 ms	9,5 ms	75,50%	1,3 ms	10,34%	1,7 ms	13,47%	0,1 ms	0,69%
4000	stark	63,32	15,8 ms	10,1 ms	63,73%	3,7 ms	23,66%	1,8 ms	11,59%	0,2 ms	1,02%
4000	mittel	69,19	14,5 ms	10,2 ms	70,25%	2,3 ms	16,08%	1,8 ms	12,55%	0,2 ms	1,12%
4000	leicht	75,67	13,2 ms	10,2 ms	77,14%	1,1 ms	8,17%	1,8 ms	13,52%	0,2 ms	1,18%
4000	N/A	82,00	12,2 ms	9,6 ms	78,51%	0,9 ms	7,07%	1,7 ms	13,72%	0,1 ms	0,70%

Tabelle 8.1.: Abgebildet sind die Ergebnisse einer Geschwindigkeitsmessungen der Cornel Box mit zwei rotierenden Kugeln. Die angegebene Bewegungsstärke gibt ein Gefühl dafür, wie stark die Verschiebung war.

keine Lücken in der Darstellung von bewegten Objekten auf, die den Bildschirm betreten oder verlassen.

Geschwindigkeitsmessung

Für eine bessere Einschätzung, wie schnell das Verfahren arbeitet, wurde eine Geschwindigkeitsmessung durchgeführt. Die verwendete Graphikkarte war eine NVIDIA GeForce 9600 GT mit 1 GB Graphikspeicher. Damit die Messungen vergleichbar sind, wurde die global zeitgebende Einheit der Implementation so manipuliert, dass zwischen den Renderdurchläufen immer eine konstante Zeitspanne vergeht. Dieses Vorgehen ist wichtig um gleichlange Speed Lines zu erlangen, da die Ausführungszeit die durch ein Objekt zurückgelegte Strecke beeinflusst. Die konstante Zeitspanne betrug $\frac{1}{30}$ einer Sekunde.

Alle gemessenen Werte sind Mittelwerte von insgesamt 1000 Renderdurchläufen. Um zuverlässige Ergebnisse zu bekommen, ist es wichtig nach jedem Durchlauf den OpenGL-Befehl `glFinish()` auszuführen, damit die Messung das Ende des derzeitgen Rendervorgangs abwartet. Die Messung stellte die Gesamtzeit sowie die Zeiten der einzelnen Durchläufe fest. Die angegebene Anzahl der Frames pro Sekunde (FPS) ist die tatsächliche Geschwindigkeit der Darstellung. Die verwendete Szene ist eine Cornel Box mit zwei rotierenden Kugeln. Durch zusätzliche, horizontale Bewegungen der gesamten Szene zu beiden Seiten, entsteht für jeden Seed Point ein Speed Line. Die Stärke der Bewegung ist eingeteilt in *stark* bis *N/A* (*nicht vorhanden*), wobei bei *stark* lange Speed Lines erzeugt werden und bei *kaum vorhanden* nur die rotierenden Kugeln Speed Lines erzeugen. Die gesamte Messung wurde für 30000, 10000 und

Seed Point Anzahl	Bewegungsstärke	FPS	Gesamtzeit	1. Durchlauf		2. Durchlauf		3. Durchlauf		Restzeit	
30000	stark	10,59	94,4 ms	11,8 ms	12,45%	79,8 ms	84,47%	2,7 ms	2,89%	0,2 ms	0,20%
30000	N/A	55,16	18,1 ms	12,0 ms	66,42%	3,2 ms	17,54%	2,7 ms	15,16%	0,2 ms	0,89%
4000	stark	38,66	25,9 ms	11,4 ms	44,09%	11,8 ms	45,55%	2,6 ms	9,96%	0,1 ms	0,39%
4000	N/A	82,00	12,2 ms	9,6 ms	78,51%	0,9 ms	7,07%	1,7 ms	13,72%	0,1 ms	0,70%

Tabelle 8.2.: Abgebildet sind die Ergebnisse einer Geschwindigkeitsmessungen der Cornel Box mit zwei rotierenden Kugeln und einer FBO-Größe von 1600×1020 Pixeln. Die Konfiguration der Messung entspricht ansonsten der vorangegangenen Geschwindigkeitsmessung.

4000 Seed Points und jeweils vier verschiedene Stärkegrade der horizontalen Bewegung bei einer Größe des Frame Buffer Objekts (FBO) von 800×600 Pixeln durchgeführt.

In Tabelle 8.1 sind die Ergebnisse der Geschwindigkeitsmessung aufgeführt. Die Renderzeiten sind für den ersten und dritten Durchlauf bei allen Varianten nahezu konstant. Die sich ergebenden Restzeiten fallen sehr gering aus und sind deshalb zu vernachlässigen. Die Renderzeiten für die Erzeugung von Speed Lines verändern sich mit der Anzahl der Seed Points und der Stärke der Bewegung.

Bei starken Bewegungen dauert der Renderdurchlauf länger, als wenn die Bewegungen nur schwach sind. Bei schwachen Bewegungen bestimmt der erste Durchlauf die Renderzeit, werden die Bewegungen stärker, wächst immer die Berechnungszeit, die der zweite Durchlauf benötigt. Der dritte Durchlauf ist stets sehr zügig abgeschlossen, in keiner Messung fallen auf ihn auch nur fünfzehn Prozent der Gesamtzeit.

Mit einer Erhöhung der Anzahl an Seed Points nimmt die Gesamtzeit zu, wobei besonders der zweite Durchlauf mehr Zeit in Anspruch nimmt. Es ist eine lineare Abhängigkeit von der Anzahl der Seed Points zur Renderzeit des zweiten Durchlaufs zu beobachten.

Die Rendergeschwindkeit liegt immer über 30 FPS, außer für die erste Messung mit 30000 Seed Points und starken Bewegungen. Im Schnitt werden 52.5 FPS erreicht, wenn alle Messungen für 30000 Seed Points zusammen genommen werden. Für 10000 Seed Points ergeben sich 65.7 FPS und für 4000 sind es 72.5 FPS.

Die Tabelle 8.2 zeigt Ergebnisse der gleichen Konfiguration mit dem Unterschied, dass das FBO eine Größe von 1600×1020 Pixeln hat. Die gemessenen Werte unterliegen etwas stärkeren Schwankungen, bestätigen aber im Wesentlichen die vorgestellten Beobachtungen.

Ein Objekt mit höheren Polygonzahlen erhöht die Geschwindigkeit nur für den ersten Durchlauf, weil mehr Polygone gerendert werden müssen. Der Rest des Verfahrens bleibt davon unbeeinflusst. Aus diesem Grund wird keine ausführliche Geschwindigkeitsmessung für Objekte mit höheren Polygonanzahlen durchgeführt.

8.2. Analyse

Beim vorgestellten Verfahren handelt es sich um ein Post-Processing Verfahren, dass als Eingabe nur G-Buffer mit der Position und Geschwindigkeit in der Welt sowie der Geschwindigkeit im Bildraum benötigt. Außerdem ist es erforderlich, dass der Szenegraph die ModelView Matrix des vorherigen Renderdurchlaufs zwischenspeichern oder berechnen kann.

8.2.1. Leistungsvermögen

Das Verfahren erzeugt eine Vielzahl von Speed Lines über den gesamten Sichtbereich. Es zeichnet die Striche dreidimensional in den Raum, wobei es aus G-Buffern alle benötigten Informationen für die Erzeugung von Speed Lines extrahiert. Die Erzeugung von Speed Lines ist robust und arbeitet zuverlässig. Das Verfahren erfordert keine Manipulation der Geometrie, da es die benötigten Informationen in G-Buffer rendert und sie dann als Eingabe für die Generierung der Striche verwendet.

Wie die Geschwindigkeitsmessungen ergeben haben, ermöglicht das Verfahren interaktive Wiederholungsraten, selbst bei starken Bewegungen und einer hohen Rate an Seed Points. Die Geschwindigkeit des Verfahren hängt stark von der Anzahl der verwendeten Seed Points, der Menge der tatsächlich generierten Speed Lines und deren Länge ab. Die Anzahl der Seed Points beeinflusst die Renderzeit, weil dann mehr Speed Lines gezeichnet werden müssen. Stärkere Bewegungen verursachen längere Speed Lines und da der Tiefentest keine Auswirkung hat, wird jeder Strich komplett gezeichnet. Dies bedeutet, dass sich der Zeichenaufwand proportional zur Stärke der Bewegung im Bildraum erhöht. Da die Ergebnisse jedoch die interaktiven Bildwiederholungsraten zeigen und keine Änderung einen solchen Zusammenhang verhindern könnte, sind die Frameraten zufriedenstellend

Die verwendete Strichtextur kann während der Laufzeit ausgetauscht werden, um das Aussehen der erzeugten Striche an die momentanen Bedürfnisse anzupassen.

Das vorgestellte Verfahren kann auch dazu verwendet werden, Speed Lines zu zeichnen ohne Objekte auszublenden. Dadurch kann der vorgestellte Renderablauf entscheidend vereinfacht werden, weil nicht zwischen unbewegten und bewegten Objekten unterschieden werden muss. In diesem Fall fällt das zwei-schichtige FBO weg. Außerdem wird der G-Buffer für Geschwindigkeiten im Bildraum nicht mehr benötigt.

8.2.2. Grenzen

Das entwickelte Verfahren besitzt verschiedene Grenzen. So nähert es Bewegungen linear an und erfasst nur die seit dem vorherigen kompletten Renderdurchlauf erfolgten Bewegungen. Das bedeutet auch, dass die Bewegungslinien sich nicht über mehrere Bilder erstrecken, was Streaking in Comics leisten kann.

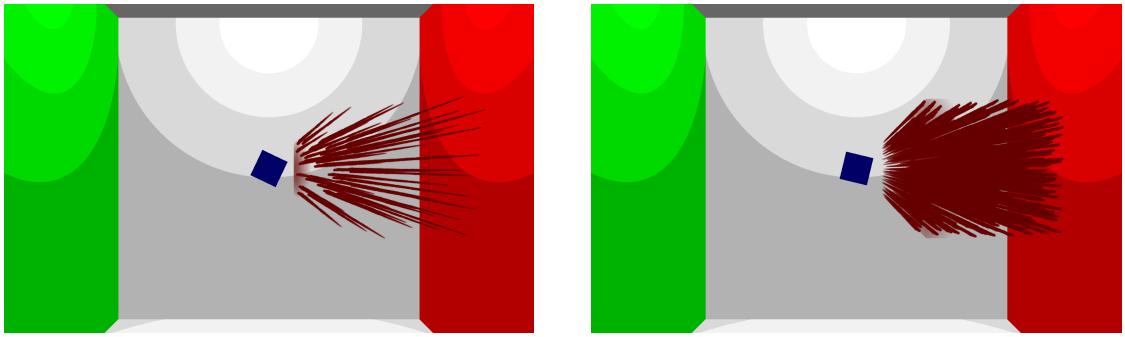

Abbildung 8.2.1.: Die Szene zeigt eine pulsierende rote Box mit einer rotierenden blauen Box in der Mitte. Durch Streaking, ist die rote Box fast völlig ausgeblendet.
links: Die rote Box wird kleiner und bewegt sich damit vom Betrachter weg. Es werden weniger Speed Lines generiert.
rechts: Die rote Box wird größer und bewegt sich auf den Betrachter zu. Es werden viele Speed Lines generiert.

Beim Vergleich einer Bewegung vom Betrachter weg und zu ihm hin fällt auf, dass bei der dem Betrachter abgewandten Bewegung weniger Striche generiert werden, als umgekehrt. Ein Beispiel hierfür ist in Abbildung 8.2.1 in Form einer pulsierenden roten Box zu sehen. Das linke Bild zeigt eine schrumpfende Box, das rechte Bild eine Wachsende. Die Ursache für die wenigen Striche liegt an der kleineren Fläche des sich fortbewegenden Objekts. Durch die kleinere Fläche kann das Objekt nicht von genügend Seed Points getroffen werden, weshalb weniger Striche generiert werden.

Wenn die Seed Points nicht dicht genug liegen, kann es passieren, dass einige kleinere Objekte nicht erfasst werden können und so für diese Objekte keine Speed Lines entstehen. Aus demselben Grund kann nicht garantiert werden, dass markante Stellen eines Objekts getroffen werden. Doch da bewegte Objekte nicht auf der Stelle verweilen, sondern sich in der Bildebene fortbewegen, werden sie irgendwann von einem Seed Point erfasst. Deswegen fällt dieses Problem nur auf, wenn die Phasen der Bewegung als Einzelbilder betrachtet werden.

Das Verfahren arbeitet auf vielen G-Buffern mit hohen Genauigkeiten. Momentan werden insgesamt drei Tiefenbuffer und drei Farbbuffer verwendet. Dazu kommen drei Buffer mit jeweils vier Farbkanälen und 32-Bit Genauigkeit pro Kanal, die sich in einem zwei-schichtigen FBO befinden. Daraus ergeben sich pro Pixel 12 KB für die Tiefe, 12 KB für die Farbe mit Alphakanal und 96 KB für die restlichen Buffer. Für eine FBO Größe von 800×600 wären das 57.6 MB an Graphikspeicher, der nur für das Verfahren zur Verfügung stehen muss. Bei einer Größe von 1600×1020 ergeben sich 195.84 MB.

8.2.3. Fehlerfälle

Streaking blendet Objekte ab einer gewissen Geschwindigkeit aus. Aus dem Grund verwendet es zwei Farbbuffer, einer für bewegte und der andere für unbewegte Objekte. Wenn sich zwei

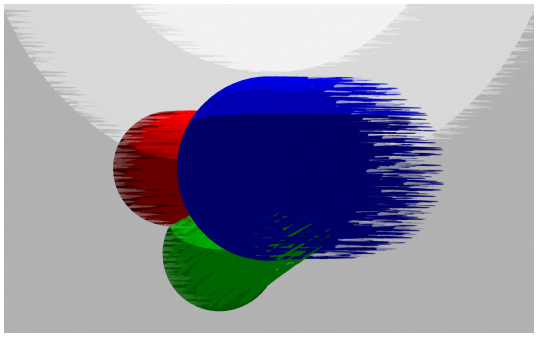

Abbildung 8.2.2.: Eine Szene mit zwei rotierenden Kugeln. Die Szene ist nach links verschoben worden.

bewegte Objekte überlagern und das vordere beider Objekt so schnell ist, dass es ausgeblendet wird, wird es nicht über das dahinter liegende Objekt geblendet, sondern mit dem Farbbuffer für unbewegte Objekte verschmolzen. Unterabschnitt 6.5.3 auf Seite 88 beobachtete diesen Fehler bereits. In Abbildung 8.2.2 ist dieser Fall an der roten Kugel zu sehen, die an einigen Stellen dunkler wirkt. Diese Stellen sollten mit dem anzunehmenden Hintergrund, nämlich der Cornel Box, geblendet werden. Stattdessen werden sie mit dem Farbbuffer für unbewegte Objekte geblendet, der leer und dementsprechend schwarz ist.

Werden Speed Lines übereinander gezeichnet, überlagern sie sich nicht entsprechend ihrer Tiefe, sondern das Speed Line des letzten gezeichneten Seed Points liegten über allen anderen. Dieser Effekt kann zur Folge haben, dass der zweite Renderdurchlauf einen eigentlich verdeckten Strich zuletzt bearbeitet und der anschließende dritte Durchlauf diesen dann unbeabsichtigt als vordersten Strich zeichnet. Das Bild in Unterabschnitt 6.5.3 auf Seite 88 zeigt diesen Sachverhalt. Die rote und grüne Kugel, sind jeweils hinter der blauen Kugel. Die gesamte Szene hat sich etwas nach links bewegt, weshalb auch für die blaue Kugel Speed Lines generiert werden. Die Speed Lines der grünen Kugel treten hervor, obwohl dort nur blaue Striche zu sehen sein sollten.

8.3. Bewertung

Die Bewertung des entwickelten Verfahrens erfolgt anhand der Zielsetzung dieser Arbeit, vgl. Abschnitt 1.2, und der Bewertungskriterien aus Abschnitt 3.4.

Streaking kann Bewegung suggerieren und stellt Bewegungsunschärfe durch Schlieren dar, die durch Speed Lines erreicht werden. Das Verfahren ist echtzeitfähig und erfordert nur, dass der Szenegraph die vorherige ModelView-Matrix des zu zeichnenden Objekts mitführt. Das Verfahren verändert die Geometrie nur im Geometry-Shader. Die Graphikkarte führt sämtliche Berechnungen, die die Schlieren betreffen, aus, was die Zielsetzung aus Abschnitt 1.2 erfüllt.

Anforderungen	Masuch	Song	Schmid	Hanl	Streaking
Beachtet minimale und maximale Ausdehnung (**A1**)	✓		✓	✓	(✓)
Verwendet markante Stellen (**A2**)	✓	(✓)	✓	✓	(✓)
Anzahl einstellbar (**A3**)	✓	✓	✓	✓	(✓)
Keine konstanten Abstände (**A4**)	✓	✓	✓	✓	✓
Keine Ansammlung auf bestimmten Regionen (**A5**)	✓	(✓)	✓	✓	✓
Hinter Objekt / keine Überschneidungen (**A6**)	✓		✓	(✓)	
Länge einstellbar (**A7**)	✓	✓	✓	✓	

Tabelle 8.3.: Gegenüberstellung der genannten Verfahren anhand der Anforderungen aus Unterunterabschnitt 3.3.3.1.
✓ = voll erfüllt; (✓) = zum Teil erfüllt bzw. erwecken den Anschein die Anforderungen erfüllen zu können.

Für einen genaueren Vergleich werden die Bewertungen aus Abschnitt 3.4 herangezogen. Tabelle 8.3 vergleicht die vorgestellten Speed Line Verfahren mit Streaking. Bevor genauer auf den Vergleich eingegangen wird, muss angemerkt werden, dass das entwickelte Verfahren nicht primär Speed Lines zeichnet, sondern bewegte Objekte durch die Annäherung mit Speed Lines undeutlicher aussehen lässt. Da es dabei viele Speed Lines zeichnet, wird es dennoch mit den vorgestellten Verfahren verglichen.

Die ersten drei Anforderungen werden nicht erfüllt, weil Streaking nicht auf der Geometrie selbst arbeitet, sondern auf einer gerenderten bildlichen Repräsentation. Da zufällig verteilte Seed Points die Startpunkte bilden, hängt es im Wesentlichen von ihrer Position und der Lage des Objekts im Bildraum ab, ob die minimal und maximale Ausdehnung beachtet werden. Das Gleiche gilt für **A2**. Die Anzahl der Speed Lines ist für das gesamte Bild einstellbar, aber nicht für das Objekt. Die Anforderungen **A1**, **A2** und **A3** sind aus den Gründen nur zum Teil erfüllt und stehen deswegen in Klammern. Die Anforderung **A4** ist erfüllt, weil die Seed Points gleichmäßig aber nicht regelmäßig über das Bild verteilt sind. Diese Verteilung unterbindet auch die Ansammlung auf bestimmten Regionen entsprechend Anforderung **A5**. Die Anforderung **A6** ist nicht erfüllt, weil Speed Lines das Objekt schneiden und immer auf dem Objekt beginnen. Anforderung **A7** ist ebenfalls nicht erfüllt, weil die Darstellungsdauer für einen kompletten Renderdurchlauf gilt und die Länge eines Speed Line von der zurückgelegten Geschwindigkeit des Objekt abhängt.

Die Tabelle 8.4 stellt die Verfahren abermals einander gegenüber, geht dabei aber auf weitere Eigenschaften ein, die für interaktive Programme wichtig sind. Weitere Informationen über die Kriterien sind in Abschnitt 3.4 auf Seite 36 enthalten.

Abschnitt 8.1 auf Seite 104 zeigt, dass das entwickelte Verfahren echtzeitfähig ist. Die erzeugten Striche sind Polygone, deren Aussehen Texturen bestimmen, weswegen sie stilisierbar sind. Das Verfahren arbeitet nicht auf Basis der Geometrie, sondern auf den Eigenschaften des vor-

Eigenschaften	Masuch	Song	Schmid	Hanl	Streaking
Echtzeitfähig	(✓)			✓	✓
Stilisierbar			✓	gering	✓
Verwendet Geometrie	✓(Kontur)	✓	✓ (TAO)	✓	(G-Buffer)
Unstetige Generierung von Speed Lines möglich	(✓)	✓	(✓)		
Abbildung nicht-linearer Bewegungen	✓	✓	✓	✓	
Speed Lines über mehrere Frames	✓	✓	✓	✓	

Tabelle 8.4.: Gegenüberstellung weiterer Eigenschaften der vorgestellten Verfahren.

dersten Oberflächenpunkts in den gerenderten G-Buffern. Es generiert Speed Lines in Abhängigkeit von der Bewegungsgeschwindigkeit und nur eine stetige Generierung von Speed Lines ist vorgesehen. Streaking beherrscht nur lineare Bewegungen pro Pixel und zeichnet Speed Lines nur für die Dauer eines Frames.

Das entwickelte Verfahren kann in interaktiven Anwendungen und Spielen verwendet werden, in denen Objekte mit zunehmender Geschwindigkeit immer undeutlicher werden. In Spielen, die NPR-Verfahren einsetzen und Bewegungen durch Speed Lines andeuten , kann das Verfahren ebenfalls eingesetzt werden, da es Speed Lines auf jeder Geometrie erzeugen kann und keine Berechnungen vorab vornehmen muss.

9. Fazit

9.1. Zusammenfassung

Diese Arbeit verfolgte zwei eigene Ansätze, die Motion Blur mit Non-Photorealistic Rendering Verfahren annähern.

Der erste Ansatz verwendet den Velocity-Buffer und TAMs um den gewünschten Effekt zu erzielen. Mehrere Schwierigkeiten ergaben sich während seiner Umsetzung, die diese Arbeit nicht in einem realistischen Zeitrahmen lösen hätte können.

Der zweite Ansatz verfolgt einen anderen Lösungsweg. Er verwendet den Geometry-Shader in Verbindung mit Seed Points um Strichpolygone zu erzeugen und zu zeichnen. Während der Umsetzung sind am Konzept aufgetretene Mängel beseitigt worden.

Das Verfahren erzeugt eine Vielzahl von Speed Lines über den gesamten Sichtbereich. Es zeichnet die Striche dreidimensional in den Raum, wobei es alle benötigten Informationen aus G-Buffern extrahiert. Da das Verfahren die benötigten Informationen in G-Buffer rendert und sie dann als Eingabe für den Generierungsprozess der Striche verwendet, erfordert es keine Manipulation von Geometrie. Durch Kombination von drei Farbbuffern von unbewegten und bewegten Objekten sowie den Speed Lines werden Schlieren erreicht. Das Verfahren arbeitet dabei in Echtzeit und ist darauf ausgelegt, die einzelnen Speed Lines über eine Strichtextur zu stilisieren. Die Erzeugung von Speed Lines ist robust und arbeitet zuverlässig.

Die Arbeit hat das Ziel, bewegte Objekte mit Speed Lines anzunähern und einen dem Motion Blur ähnlichen Effekt einzubinden, erreicht. Die zwei weiteren gestellten Anforderungen, nämlich dass das angestrebte Verfahren echtzeitfähig sein soll und nicht direkt auf der Basis von Geometrie arbeitet, erfüllt die Arbeit ebenfalls. Das entwickelte Verfahren heißt *Streaking*.

9.2. Ausblick

Das hier dargestellte Verfahren Streaking besitzt einiges Potential für Weiterentwicklungen. So wäre interessant, ob anstatt gerader Speed Lines auch gebogene Speed Lines möglich wären und besser aussähen.

9.2.1. Weiterentwicklung von Streaking

Der Speicherverbrauch der derzeitigen Implementation ist noch zu hoch und kann reduziert werden. Hier ist von Interesse, wie stark der Speicherverbrauch schrumpfen werden kann, ohne dass erstens die Implementation geändert werden muss, zweitens die Rendergeschwindigkeit darunter leidet und drittens wie weitreichende Änderungen tatsächlich möglich wären.

Diese Arbeit ist nicht auf die Beleuchtung der einzelnen Speed Lines eingegangen. Insbesonders in Bezug auf die Schattengenerierung sind Fragen zu klären, wie: Wie können Schatten über Shadow Mapping erzeugt werden und sähen die Ergebnisse glaubhaft aus?

Order-Independent Transparency ist in dieser Arbeit häufig erwähnt worden. Nachfolgende Arbeiten könnten untersuchen, wie K-Buffer verwendet werden könnten, um die gesamte Szene und gleichzeitig Speed Lines zu zeichnen, während bewegte Objekte ausgeblendet werden.

Eine weitere Idee, die ohne großen Aufwand umzusetzen wäre, ist die Strichstile abhängig vom Objekt zu variieren.

9.2.2. Mögliche Entwicklungen basierend auf Streaking

Diese Arbeit hatte in erster Linie zum Ziel Motion-Blur mit Speed Lines zu vereinen, nicht ein weiteres Speed Line Verfahren zu entwickeln. Dabei ist ein neues Konzept zur Erzeugung von Speed Lines entstanden, welches Erweiterungen ermöglicht.

Das entwickelte Verfahren könnte angepasst werden, um richtige Speed Lines zu generieren, wobei die sieben an Speed Line Verfahren gestellten Anforderungen erfüllt werden sollten. Solche Verfahren erfordert längere Speed Lines, die den Verlauf der Bewegung über mehrere Frames darstellen. Um dies zu erreichen könnte ein solches Verfahren die benutzen Velocity-Buffer und Tiefenbuffer über einen gewissen Zeitraum zusammen mit einer Objekt ID zwischenspeichern. Die Objekt ID dient hierbei zur Identifikation der Objekte in älteren G-Buffern. Anstatt den Velocity-Buffer und den Tiefenbuffer zu speichern, könnte der Transform Feedback des Geometry-Shaders schon gezeichnete Striche zwischenspeichern, um sie später erneut zu zeichnen. Ein zu lösendes Problem ist der immer gleiche Startpunkt, der durch die Seed Points gegeben ist und sich nicht verändert. Gewünscht wäre aber, das ältere Speed Lines da beginnen, wo die aktuellen Speed Lines enden.

Ein weitere Möglichkeit Streaking zu erweitern ist es Speed Lines nur an bestimmten Stellen zu zeichnen. Gültigkeitsbereiche können festgelegen, wo Speed Lines gerendert werden dürfen. Diese Bereiche könnte das Verfahren zum Beispiel über Boolean-Texturen an den Geometry-Shader weitergeben. Eine solche Veränderung würde das Verfahren flexibel halten.

Streaking kann also als Ansatzpunkt für viele neue Entwicklungen dienen.

A. OpenGL 3.0

Die Open Graphics Library (OpenGL) ist eine API um Graphikkarten anzusprechen. Die verwendete OpenGL Version ist 3.0. Diese Version teilt sich in zwei Profile: Compatibility und Core. Das Compatibility-Profil enthält alle Methoden der Vorgängerversion 2.1, aber hat diese als deprecated[1] markiert. Im Core-Profil sind alle deprecated Methoden nicht mehr enthalten. Für die Implementation kam das Core-Profil zum Einsatz. Gegenüber der vorherigen Version 2.1 sind einige Änderungen erfolgt.

A.1. Wesentliche Unterschiede zur Vorgängerversion

In OpenGL 2.1 sind viele Funktionen enthalten, die den Umgang mit OpenGL erleicht, wie z.B. Methoden zur Manipulation des Matrixstacks. In OpenGL 3.0 sind viele dieser Funktionen deprecated und können im Core-Profil nicht genutzt werden.

Aus dem OpenGL 3.0 Core-Profil sind

- die Matrixstacks mit den dazugehörigen Methoden,
- der Immediate-Mode mit den Display-Listen,
- die Fixed-Function Pipeline mit der Beleuchtung und
- der Accumulation Buffer

entfernt worden. Stattdessen sind die erforderlichen Strukturen und Methoden selbst zu implementieren.

A.2. Vordefinierte Datentypen in verwendeten Shaderprogrammen

Bei Codebeispielen von Shaderprogrammen werden einige Eingabe- und Ausgabevariablen, sowie uniforme Variablen als gegeben angenommen, obwohl sie nicht explizit angegeben werden.

[1] Deprecated ist eine Anmerkung in APIs, um Methoden oder Klassen als nicht mehr gebräuchlich zu markieren, weil sie in naher Zukunft aus der API zu entfernen.

```
uniform matrices {
   mat4 projection;
   mat4 modelView;
   mat4 normalMatrix;
   mat4 previousModelView;
};

uniform light{
   vec4 color;
   vec4 position;
   vec3 direction;
} light;

uniform material {
   vec4 color;
};

in vertex {
   vec4 position;
   vec3 normal;
   vec3 vertexColor;
} element;
```

Listing A.1: Mehrere Variablen, die bei Codebeispielen als gegeben angenommen werden.

In Listing A.1 sind Variablen abgedruckt, die in vielen Codebeispielen dieser Arbeit vorkommen, aber nicht angegeben sind. Die Uniform Block Variable `matrices` zeigt, wie alle verwendeten Matrizen heißen. Die Variablen `light` und `material` stehen für das verwendete Licht und Material. Die Eingabevariablen des Shaders sind in einen Eingabeblock definiert, der `vertex` heißt. Sie stehen für mehrere Attribute, die ein Vertex besitzt und an sie werden Vertex Buffer Objekte angebunden.

Blöcke von Uniform Variablen können als Buffer im Speicher der Graphikkarte existieren. Ihre Verwendung bewirkt, dass ein Shaderprogramm wissen muss, was für Typen die Variablen besitzen und in welcher Reihenfolge sie auftreten. Die Verwendung im Programmcode wird dadurch vereinfacht, weil der Shader nur die Adresse des Buffers bekommt und das auszuführende Programm nur während der Konstruktion wissen muss, wie die Variable im Shader heißt, um sie mit dem Buffer zu verknüpfen. Würde eine einfach Uniform Variable zur Anwendung kommen, dann müsste der verwendete Szenegraph für jedes Programm den Namen der Projektionsmatrix kennen.

A.3. Mehr-schichtige Frame Buffer Objekte

OpenGL 3.0 beherrscht mehr-schichtige Frame Buffer Objekte (FBOs). Um sie zu verwenden werden FBOs statt 2D Texturen 3D Texturen angehangen. Eine 3D Textur ist in OpenGL nichts anderes als mehrere 2D Texturen, die als Stapel betrachtet werden.

Um in die verschiedenen Ebenen zu schreiben, muss im Geometry-Shader die Integer-Variable `gl_Layer` belegt werden, sonst wird nur in die erste Ebene geschrieben.

Zu 3D Texturen zählen auch 2D Textur Arrays. Der Unterschied zwischen ihnen ist, dass bei 2D Textur Arrays beim Auslesen im Fragment-Shader nur eine Ebene angesprochen werden kann, hingegen wird bei der 3D Textur zwischen den Ebenen interpoliert, sofern dies eingestellt wurde. Für den Zugriff auf die dritte Ebene einer 3D Textur mit vier Ebenen, bekommt die Texturkoordinate t_z den Wert $0,75$. Bei einem 2D Textur Array wäre es $t_z = 3$.

Außerdem nimmt die Tiefe eines 2D Textur Arrays, also die Anzahl der Ebenen, mit wachsenden MipMap-Level nicht ab, hingegen bei einer 3D Textur schon. So hat eine 3D Textur mit insgesamt vier Ebenen auf der nächsten MipMap-Stufe nur noch zwei Ebenen.

A.4. Kopieren von Frame Buffern

Um den Inhalt von FBOs zu kopieren, kann die Methode `glBlitFramebuffer()` vewendet werden. Bevor diese Methode aber aufgerufen werden kann, muss der richtige Frame Buffer des FBOs ausgewählt werden. Dafür werden die Methode `glReadBuffer()` und `glDrawBuffer()` verwendet, die den angegeben Frame Buffer des derzeitig gebunden FBOs verwenden. Sie bekommen als einziges Argument die ID des zu verwendenen Frame Buffers, die über `GL_COLOR_ATTACHMENT`*i* benutzt werden kann, wobei *i* für die ID des Objekts steht.

Die Methode `glBlitFramebuffer()` verwendet die ersten vier Integerwerte, um den zu kopierenden Ausschnitt festzulegen. Die darauffolgenden vier Intergerwerte spezifizieren den Ausschnitt, indem der zu kopierende Ausschnitt geschrieben werden soll. Der neunte Parameter legt fest, welche Masken zum Kopieren verwendet werden sollen. Es werden zum Beispiel die Werte `GL_COLOR_BUFFER_BIT` oder `GL_DEPTH_BUFFER_BIT` erwartet. Mit dieser Maske ist es erst möglich die Tiefe zu kopieren, denn der Tiefenbuffer kann nicht direkt zum Lesen oder Schreiben gebraucht werden. Wenn der Quell- und Zielausschnitt nicht gleich groß sind, bestimmt der letzte Parameter, wie zwischen den Farbwerten interpoliert wird. Ein möglicher Wert wäre z.B. `GL_NEAREST`.

A.5. Blending

Das Blending geschieht nachdem der Fragment-Shader seine Arbeit beendet hat und sein Ergebnis in den Farbbuffer schreibt. Das Blending muss vor seiner Verwendung erstmal über

`glEnable()` mit dem Wert `GL_BLEND` aktiviert werden. Danach kann das Programm über die Methoden `glBlendEquation()` und `glBlendFunc()` festgelegen, wie die Farbwerte geblendet werden.

Die Methode `glBlendEquation()` bestimmt, welche Gleichung zum Blenden der Quellfarbe, die die Ergebnisfarbe vom Fragment-Shader ist, und der Zielfarbe zum Einsatz kommt. Zum Beispiel kann zwischen additiven und multiplikativen Blenden gewählt werden. Die Methode `glBlendFunc()` beeinflusst die Gewichtungen der Quelle- und Zielfarbe.

Beide Methoden legen die Gleichung und Gewichtungsfaktoren für die Farbe und den Alphawert fest. Wenn die Situation es erfordert, kann mit den Mothoden `glBlendEquationSeparate()` und `glBlendFuncSeparate()` die Gleichung und die Gewichtungsfaktoren separat geändert werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[AMHH08] AKENINE-MÖLLER, Tomas ; HAINES, Eric ; HOFFMAN, Naty: *Real-Time Rendering, Third Edition*. 3. A K Peters/CRC Press, 2008. – ISBN 9781568814247

[AMMH07] AKENINE-MÖLLER, Tomas ; MUNKBERG, Jacob ; HASSELGREN, Jon: Stochastic rasterization using time-continuous triangles. In: *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2007. – ISBN 978–1–59593–625–7, 7–16. – http://fileadmin.cs.lth.se/graphics/research/papers/2007/stochrast/stochrast2007.pdf

[BL04] BAXTER, William V. ; LIN, Ming C.: A Versatile Interactive 3D Brush Model. In: *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*. Washington, DC, USA : IEEE Computer Society, 2004 (PG '04). – ISBN 0–7695–2234–3, 319–328. – http://gamma.cs.unc.edu/BRUSH/

[BS00] BUCHANAN, John W. ; SOUSA, Mario C.: The edge buffer: a data structure for easy silhouette rendering. In: *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM, 2000 (NPAR '00). – ISBN 1–58113–277–8, S. 39–42. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.1877&rep=rep1&type=pdf

[CL93] CABRAL, Brian ; LEEDOM, Leith C.: Imaging vector fields using line integral convolution. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1993 (SIGGRAPH '93). – ISBN 0–89791–601–8, 263–270. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.8134&rep=rep1&type=pdf

[CPIS02] CHENNEY, Stephen ; PINGEL, Mark ; IVERSON, Rob ; SZYMANSKI, Marcin: Simulating cartoon style animation. In: *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM, 2002 (NPAR '02). – ISBN 1–58113–494–0, 133–138. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.1844&rep=rep1&type=pdf

[Cut02] CUTTING, James E.: Representing motion in a static image: constraints and parallels in art, science, and popular culture. In: *Perception* 31 (2002), Nr. 10, 1165-93. http://www.psy.vanderbilt.edu/faculty/blake/Honors_183/PDFs/CuttingOnMotion.pdf

[Dec96] DECAUDIN, Philippe: Cartoon Looking Rendering of 3D Scenes / INRIA. Version: Juni 1996. www.antisphere.com/Research/RR-2919.php. 1996 (2919). – Research Report

[Fer] FERNANDES, António R.: *GLSL Tutorial - Toon Shading*. http://www.lighthouse3d.com/opengl/glsl/index.php?toon

[Gei07] *Kapitel* Generating Complex Procedural Terrains Using the GPU. In: GEISS, Ryan: *GPU Gems 3*. Addison-Wesley Professional, 2007. – ISBN 9780321515261

[GG01] GOOCH, Bruce ; GOOCH, Amy: *Non-Photorealistic Rendering*. A K Peters/CRC Press, 2001 http://amazon.com/o/ASIN/1568811330/. – ISBN 9781568811338

[GGSC98] GOOCH, Amy ; GOOCH, Bruce ; SHIRLEY, Peter ; COHEN, Elaine: A non-photorealistic lighting model for automatic technical illustration. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1998 (SIGGRAPH '98). – ISBN 0–89791–999–8, 447–452. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.6762&rep=rep1&type=pdf

[Gra10] GRAIS, Stuart: *Movement*. http://facweb.cs.depaul.edu/sgrais/. Version: 11 2010. – last visited: 2010-12-13

[Gre03] GREEN, Simon: *Stupid OpenGL Shader Tricks*. Game Developers Conference. http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf. Version: March 2003

[GT07] GEISS, Ryan ; THOMPSON, Michael: *NVIDIA Demo Team Secrets: Cascades*. NVidia Developer Zone. http://developer.nvidia.com/object/gdc-2007.htm. Version: March 2007. – last visited: 2010-12-02

[HA90] HAEBERLI, Paul ; AKELEY, Kurt: The accumulation buffer: hardware support for high-quality rendering. In: *SIGGRAPH Comput. Graph.* 24 (1990), September, 309–318. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4497&rep=rep1&type=pdf. – ISSN 0097–8930

[Han04] HANL, Christian: *Echtzeit-Rendering von Bewegung im Comic-Stil*, Fachhochschule Hagenberg - Fakultät für Informatik, Kommunikation und Medien, Diplomathesis, Juni 2004. http://bl002ged.edis.at/thesis.pdf

[HHD04] HALLER, Michael ; HANL, Christian ; DIEPHUIS, Jeremiah: Non-Photorealistic Rendering Techniques for Motion in Computer Games. In: *In ACM Computers in Entertainment 2(4* 2 (2004), 2004. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.1389&rep=rep1&type=pdf

[HS04] HALLER, Michael ; SPERL, Daniel: Real-time painterly rendering for MR applications. In: *Proceedings of the 2nd international conference on Computer gra-*

phics and interactive techniques in Australasia and South East Asia. New York, NY, USA : ACM, 2004 (GRAPHITE '04). – ISBN 1–58113–883–0, 30–38. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.6784&rep=rep1&type=pdf

[HV10] *Kapitel* III Rendering Techniques - 2 NPR Effects Using the Geometry Shader. In: HERMOSILLA, Pedro ; VÁZQUEZ, Pere-Pau: *GPU Pro: Advanced Rendering Techniques*. A K Peters/CRC Press, 2010. – ISBN 9781568814728, 149-165

[HWSE99] HEIDRICH, Wolfgang ; WESTERMANN, Rüdiger ; SEIDEL, Hans-Peter ; ERTL, Thomas: Applications of pixel textures in visualization and realistic image synthesis. In: *Proceedings of the 1999 symposium on Interactive 3D graphics*. New York, NY, USA : ACM, 1999 (I3D '99). – ISBN 1–58113–082–1, 127–134. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.3135&rep=rep1&type=pdf

[HZ00] HERTZMANN, Aaron ; ZORIN, Denis: Illustrating smooth surfaces. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000 (SIGGRAPH '00). – ISBN 1–58113–208–5, 517–526. – http://mrl.nyu.edu/publications/illustrating-smooth/hertzmann-zorin.pdf

[Ihm04] IHME, Burkhard: *Comic! Jahrbuch 2005*. 1. Interessenverband Comic e.V. ICOM, 2004 http://amazon.de/o/ASIN/3888349354/. – ISBN 9783888349355

[Imm] IMMANUEL-KANT-GYMNASIUM - FACHSCHAFT KUNST: *Impressionismus (etwa 1860- 1900)*. http://www.kunstwissen.de/fach/f-kuns/01.htm. – last visited: 2010-12-13

[Iow] IOWA STATE UNIVERSITY: *The Halton Sequence*. http://orion.math.iastate.edu/reu/2001/voronoi/halton_sequence.html

[KHK03] KAWAGISHI, Yuya ; HATSUYAMA, Kazuhide ; KONDO, Kunio: Cartoon Blur: Non-Photorealistic Motion Blur. In: *Computer Graphics International Conference* 0 (2003), 276. http://doi.ieeecomputersociety.org/10.1109/CGI.2003.1214482. – ISSN 1530–1052

[LMHB00] LAKE, Adam ; MARSHALL, Carl ; HARRIS, Mark ; BLACKSTEIN, Marc: Stylized rendering techniques for scalable real-time 3D animation. In: *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM, 2000 (NPAR '00). – ISBN 1–58113–277–8, 13–20. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.858&rep=rep1&type=pdf

[MB05] MCREYNOLDS, Tom ; BLYTHE, David: *Advanced Graphics Programming Using OpenGL (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 2005. – ISBN 9781558606593

[MB07a] MYERS, Kevin ; BAVOIL, Louis: Stencil routed A-Buffer. In: *ACM SIGGRAPH 2007 sketches*. New York, NY, USA : ACM, 2007 (SIGGRAPH '07). – http://staffwww.itn.liu.se/~andyn/courses/tncg08/sketches/content/sketches/0518.pdf and http://developer.download.nvidia.com/presentations/2007/siggraph/stencil_routed_a-Buffer_sigg07.ppt

[MB07b] MYERS, Kevin ; BAVOIL, Louis: *Stencil Routed K-Buffer*. NVidia. http://developer.download.nvidia.com/SDK/10/direct3d/Source/StencilRoutedKBuffer/doc/StencilRoutedKBuffer.pdf. Version: November 2007

[Mic07] MICROSOFT COOPERATION: *The DirectX Software Development Kit*. Microsoft Cooperation. http://msdn.microsoft.com/en-us/library/ee416417(VS.85).aspx. Version: November 2007. – last visited: 2010-12-05

[Miu03] MIURA, Kentaro: *Berserk 22*. 1. Panini Manga und Comic, 2003 http://amazon.de/o/ASIN/3899214293/. – ISBN 9783899214291

[MSS99] MASUCH, Maic ; SCHLECHTWEG, Stefan ; SCHULZ, Ronny: Speedlines: Depicting Motion in Motionless Pictures. In: *SIGGRAPH*. New York : ACM Press/ACM SIGGRAPH, 1999, 277. – http://isgwww.cs.uni-magdeburg.de/~masuch/anewgallery/speedlines.pdf

[Mud10] MUDERS, Thomas: *Cool 2D lines with openGL*. dev.spunkmeyer.de. http://dev.spunkmeyer.de/?e=33. Version: August 2010. – last visite: 2010-12-02

[Nie06] NIENHAUS, Marc: *Real-Time Non-Photorealistic Rendering Techniques for Illustrating 3D Scenes and their Dynamics*, University of Potsdam, Computer Science Department and Hasso Plattner Institute, Diss., 2006. http://www.hpi.uni-potsdam.de/fileadmin/hpi/Forschung/Publikationen/Dissertationen/Nienhaus/Diss_Nienhaus.pdf

[Oh10] OH, Jhon: *Pixel Motion Blur*. www.slideshare.net. http://www.slideshare.net/ozlael/motionblur-3252650. Version: March 2010. – last visited: 2010-12-02

[OKKI05] OBAYASHI, Syoichi ; KONDO, Kunio ; KONMA, Toshihiro ; IWAMOTO, Ken-ichi: Non-photorealistic motion blur for 3D animation. In: *ACM SIGGRAPH 2005 Sketches*. New York, NY, USA : ACM, 2005 (SIGGRAPH '05). – http://doi.acm.org/10.1145/1187112.1187224

[Oxf10] OXFORD LASERS: *High Speed Imaging*. http://www.oxfordlasers.com/imaging/high_speed. Version: 2010

[Pad09] *Kapitel* Efficient Real-Time Motion Blur for Multiple Rigid Objects. In: PADGET, Ben: *ShaderX7: Advanced Rendering Techniques*. 1. Charles River Media, 2009. – ISBN 9781584505983, S. 277–283

[PBN07] PATIDAR, Suryakant ; BHATTACHARJEE, Shiben ; NARAYANAN, Jag Mohan Singh P. J.: Exploiting the Shader Model 4.0 Architecture / Center for Visual Information Technology, IIIT Hyderabad. Version: March 2007. http://researchweb.iiit.ac.in/~shiben/docs/SM4_Skp-Shiben-Jag-PJN_draft.pdf. 2007 (145). – Forschungsbericht

[PFH00] PRAUN, Emil ; FINKELSTEIN, Adam ; HOPPE, Hugues: Lapped textures. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000 (SIGGRAPH '00). – ISBN 1–58113–208–5, 465–470. – http://research.microsoft.com/en-us/um/people/hoppe/proj/lapped/

[PHWF01] PRAUN, Emil ; HOPPE, Hugues ; WEBB, Matthew ; FINKELSTEIN, Adam: *Real-Time Hatching*. http://www.cs.princeton.edu/gfx/proj/hatching/. Version: 2001

[Por07] PORCINO, Nick: *Lost Planet Parallel Rendering*. Meshula.net website. http://meshula.net/wordpress/?p=124. Version: October 2007

[RC99] RASKAR, Ramesh ; COHEN, Michael: Image precision silhouette edges. In: *Proceedings of the 1999 symposium on Interactive 3D graphics*. New York, NY, USA : ACM, 1999 (I3D '99). – ISBN 1–58113–082–1, S. 135–140. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.434&rep=rep1&type=pdf

[RC10] RING, Kevin ; COZZI, Patrick: *Geometry Shader Silhouettes without Adjacency Information*. virtualglobeandterrainrendering.blogspot.com. http://virtualglobeandterrainrendering.blogspot.com/2010/07/geometry-shader-silhouettes-without.html. Version: July 2010. – last visited: 2010-12-03

[RMM10] RITCHIE, Matt ; MODERN, Greg ; MITCHELL, Kenny: Split second motion blur. In: *ACM SIGGRAPH 2010 Talks*. New York, NY, USA : ACM, 2010 (SIGGRAPH '10). – ISBN 978–1–4503–0394–1, S. 17:1–17:1. – http://doi.acm.org/10.1145/1837026.1837048

[Ros07] *Kapitel* Motion Blur as a Post-Processing Effect. In: ROSADO, Gilberto: *GPU Gems 3*. Addison-Wesley Professional, 2007. – ISBN 9780321515261, 575-581

[SA10] SEGAL, Mark ; AKELEY, Kurt: *The OpenGL Graphics System: A Specification*. http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf. Version: March 2010

[Sch99] SCHULZ, Ronny: *Visualisierung von Bewegungen in Liniengrafiken*, Otto-von-Guericke-Univ. Magdeburg, Fakultät für Informatik, Diplomathesis, March 1999

[Sch08] SCHÜWER, Martin: *Wie Comics erzählen*. 1. Wvt Wissenschaftlicher Verlag Trier, 2008 http://amazon.de/o/ASIN/386821030X/. – ISBN 9783868210309

[Sco94] SCOTT MCCLOUD: *Understanding Comics: The Invisible Art*. Harper Paperbacks, 1994 http://amazon.com/o/ASIN/006097625X/. – ISBN 9780060976255

[Shi05] SHIRLEY, Peter: *Fundamentals of Computer Graphics*. 2nd rev. ed. Peters, Wellesley, 2005 http://amazon.de/o/ASIN/1568812698/. – ISBN 9781568812694

[Sim] SIMMONS, Ryan: *Animating arm movements using speed lines*. Cartoon Solutions, Inc. http://www.cartoonsolutions.com/store/catalog/Animating-with-speedlines-sp-10.html

[Son05] SONG, Won C.: *Speed-line for 3D animation*, Texas A&M University, Master Thesis, December 2005. http://repository.tamu.edu/handle/1969.1/4730. – http://repository.tamu.edu/handle/1969.1/4730

[SS90] SCHNELLE-SCHNEYDER, Marlene: *Photographie und Wahrnehmung: Am Beispiel der Bewegungsdarstellung im 19. Jahrhundert (German Edition)*. Jonas, 1990 http://amazon.com/o/ASIN/3922561969/. – ISBN 9783922561965

[SS02] STROTHOTTE, Thomas ; SCHLECHTWEG, Stefan: *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation (The Morgan Kaufmann Series in Computer Graphics)*. 1st. Morgan Kaufmann, 2002 http://amazon.com/o/ASIN/1558607870/. – ISBN 9781558607873

[SSBG10] SCHMID, Johannes ; SUMNER, Robert W. ; BOWLES, Huw ; GROSS, Markus: Programmable motion effects. In: *ACM Trans. Graph.* 29 (2010), July, 57:1–57:9. http://graphics.ethz.ch/publications/papers/paperSchm10.php. – ISSN 0730–0301

[SSC03] SHIMIZU, Clement ; SHESH, Amit ; CHEN, Baoquan: Hardware-Accelerated-Motion-Blur-Generation. In: *University of Minnesota Computer Science Department Technical Report 2003-01*, 2003. – http://www.clementshimizu.com/0018-Hardware-Accelerated-Motion-Blur-Generation/ClementShimizuMotionBlur.pdf

[ST90] SAITO, Takafumi ; TAKAHASHI, Tokiichiro: Comprehensible rendering of 3-D shapes. In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1990 (SIGGRAPH '90). – ISBN 0–89791–344–2, 197–206. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.4139&rep=rep1&type=pdf

[TBI03] TATARCHUK, Natalya ; BRENNAN, Chris ; ISIDORO, John R.: Motion Blur Using Geometry and Shading Distortion. Version: 2003.

http://developer.amd.com/media/gpu_assets/ShaderX2_MotionBlurUsingGeometryAndShadingDistortion.pdf. In: ENGEL, Wolfgang (Hrsg.): *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Plano, Texas : Wordware, 2003

[UCB] UCBERKELEY: *IMAGES AND MOVIES*. http://goldberg.berkeley.edu/courses/S06/IEOR-QE-S06/images.html

[Vix08] VIX, Christian: *Real-Time Hatching auf gescannten 3D-Objekten*, Technische Universitaẗ Chemnitz - Fakultẗ fuẗr Informatik, diploma thesis, März 2008. http://nbn-resolving.de/urn:nbn:de:bsz:ch1-200800294

[Wei] WEISSTEIN, Eric W.: *Hammersley Point Set*. MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HammersleyPointSet.html

[Wika] WIKIPEDIA: *Constructions of low-discrepancy sequences*. Wikipedia. http://en.wikipedia.org/wiki/Constructions_of_low-discrepancy_sequences

[Wikb] WIKIPEDIA: *Halton sequence*. Wikipedia. http://en.wikipedia.org/wiki/Halton_sequence

[Wik06] WIKIMEDIA COMMONS: *The Horse in Motion*. http://commons.wikimedia.org/wiki/File:The_Horse_in_Motion.jpg. Version: August 2006. – last visited: 2010-12-13

[Wik08] WIKIPEDIA: *Unique Forms of Continuity in Space*. http://de.wikipedia.org/w/index.php?title=Datei:%27Unique_Forms_of_Continuity_in_Space%27,_1913_bronze_by_Umberto_Boccioni.jpg&filetimestamp=20070618202551. Version: 2008. – last visited: 2010-12-13

[Wik10] WIKIPEDIA: *Anisotropic filtering*. Wikipedia. http://en.wikipedia.org/wiki/Anisotropic_filtering. Version: September 2010

[Wlo01] WLOKA, Matthias: *Implementing Motion Blur & Depth of Field using DirectX 8*. NVidia. http://www.usf3.de/downloads/usf2001/mathias_wloka_motion_blur_depth_of_field.pdf. Version: July 2001

[WPFH02] WEBB, Matthew ; PRAUN, Emil ; FINKELSTEIN, Adam ; HOPPE, Hugues: Fine tone control in hardware hatching. In: *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*. New York, NY, USA : ACM, 2002 (NPAR '02). – ISBN 1–58113–494–0, 53–ff. – http://www.cs.utah.edu/~emilp/research.html

[WS94] WINKENBACH, Georges ; SALESIN, David H.: Computer-generated pen-and-ink illustration. In: *Proceedings of the 21st annual conference on Computer graphics*

and interactive techniques. New York, NY, USA : ACM, 1994 (SIGGRAPH '94). – ISBN 0–89791–667–0, 91–100. – http://artis.imag.fr/~Cyril.Soler/DEA/NonPhotoRealisticRendering/Papers/p91-winkenbach.pdf

[WZ96] WLOKA, Matthias M. ; ZELEZNIK, Robert C.: Interactive Real-Time Motion Blur. In: *The Visual Computer* 12 (1996), 283–295. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.8099&rep=rep1&type=pdf

[ZISS04] ZANDER, Johannes ; ISENBERG, Tobias ; SCHLECHTWEG, Stefan ; STROTHOTTE, Thomas: High Quality Hatching. In: *EUROGRAPHICS* 23 (2004), September, Nr. 3, 421–430. http://dx.doi.org/10.1111/j.1467-8659.2004.00773.x