Entwicklung eines interaktiven Shaders

Dokumentation

für die Vorlesung:

Computergraphik

des Studienganges Informationstechnik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Chaitanya Agrawal

Januar 2025

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Ein Shader ist ein Programm, das auf der Grafikkarte ausgeführt wird und zur realistischen Darstellung von 3D-Grafiken sowie visuellen Effekten beiträgt. Seine Hauptaufgabe besteht darin, die Darstellung von Oberflächen, Licht, Schatten und Farben in einer Szene zu berechnen. Durch den Einsatz von Shadern können Eigenschaften wie Glanz, Transparenz oder die Reflexion von Licht auf Objekten in Echtzeit ermittelt werden. Dies ermöglicht eine detaillierte und dynamische visuelle Darstellung, die zur Schaffung lebendiger und überzeugender Grafiken beiträgt.

Dieses Projekt ist eine interaktive, GPU-beschleunigte Simulation, die mit WebGL und Three.js realisiert wurde. Die Anwendung kombiniert dynamische Shader-Effekte, Wellenverzerrung, und Partikel-Systeme, die durch Maus- und Touch-Eingaben beeinflusst werden.

Diese Dokumentation beschreibt den Entwicklungsprozess des Projekts und erläutert die verwendeten Algorithmen sowie deren Implementierung in WebGL und JavaScript. Ein besonderer Fokus liegt auf der Entwicklung des Shaders, da dieser einen zentralen Bestandteil des Projekts darstellt.

Ziel des Projekts ist es, fortschrittliche Techniken der Computergraphik zu demonstrieren, die für Echtzeit-Simulationen geeignet sind.

2 Mathematische Algorithmen & Konzepte

2.1 Rotation eines Vektors

Ein Vektor in einer Ebene kann um einen Winkel rotiert werden, indem man auf diesen eine 2D-Rotationsmatrix angewendet. Für den Vektor $\vec{v} = (x, y)$ und dem Winkel θ kann daher folgende Rotationsmatrix angewendet werden:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Diese Matrix dreht die Koordinaten des Vektors um den Winkel θ. Die neuen Koordinaten x' und y' stellen die rotierten Werte da.

2.2 Hexagonales Gitter

Ein hexagonales Gitter wird basierend auf einem Hexagon aufgebaut, wobei es eine besondere Anordnung dieser aufweist. Um die Entfernung eines Punktes zur nächstgelegenen Kante eines Hexagons zu berechnen wird folgende Formel genutzt:

$$d = \max(0, |p_x| \cdot \cos(30^\circ) + |p_y| \cdot \sin(30^\circ) - \text{radius})$$

Dabei sind p_x und p_y die Abstände des Punktes vom Mittelpunkt des Hexagons. Der **radius** beschreibt den Abstand vom Mittelpunkt eines Hexagons zu seinen Ecken.

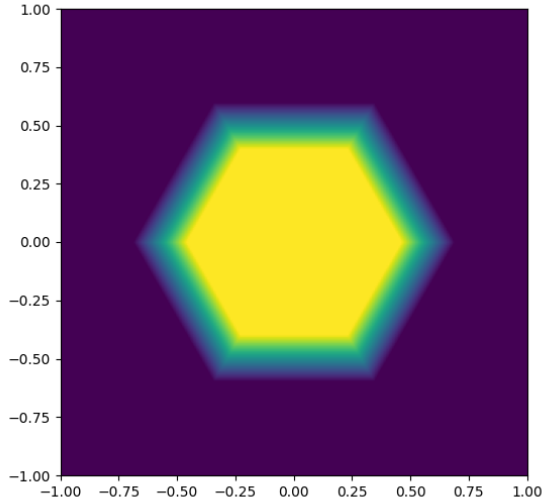

Abbildung 2.1: Hexagon

In Abbildung 2.1 ist die geometrische Struktur eines Hexagons dargestellt, das als Grundlage für die Anordnung des hexagonalen Gitters dient. Die Abbildung zeigt die charakteristischen Kanten und Ecken des Hexagons, die durch die mathematische Berechnung der Abstände und Winkel miteinander verbunden sind und somit die Grundlage des Gittermusters definieren.

2.3 HSV zu RGB Umrechnung

Um von HSV (Farbton, Sättigung, Helligkeit) zu RGB zu konvertieren, wird der Farbton H zuerst einmal auf einen Wert zwischen 0 und 6 skaliert:

$$H' = H \cdot 6$$

Danach werden die RGB-Werte durch Interpolation zu den entsprechenden Sektoren des Farbkreises berechnet. Dabei wird für jeden Farbkanal (Rot, Grün, Blau) ein Wert berechnet, der auf einen versetzten Sektor basiert:

$$\text{mod}_{rgb} = \text{mod}(H' + \text{Offset}, 6)$$

Anschließend wird der Abstand zu den Mittelpunkten der Sektoren berechnet:

$$\text{abs}_{rgb} = |\text{mod}_{rgb} - 3|$$

Dann werden die Werte auf einen Bereich von [0,1] skaliert:

$$\text{rgb} = \text{clamp}(\text{abs}_{rgb} - 1, 0, 1)$$

Nun wird der interpolierten Farbe die Sättigung S hinzugefügt:

$$\text{interpoliert} = \text{mix}(\text{weiß}, \text{rgb}, S)$$

Schließlich wird der resultierende Farbwert mit der Helligkeit V multipliziert, um den endgültigen RGB-Farbwert zu berechnen:

$$\text{finalColor} = V \cdot \text{interpoliert}$$

Das ergibt dann die RGB-Repräsentation der HSV-Farbe.

2.4 Schwarzes Loch

Das verzerren eines Bildes wird mithilfe einer dynamischen Komponente durchgeführt welche auf den UV-Koordinaten basiert. Mithilfe von d welcher den Abstand des aktuellen Punktes zu einem Zielpunkt (in diesem Fall der Maus) darstellt kann mithilfe der Folgenden Formel die Verzerrung dargestellt werden:

$$\vec{u}_{new} = \vec{u} + \frac{D}{0.1 + d^2} \cdot \sin(t) \cdot \hat{d}$$

Dabei ist $\vec{u}$ der ursprüngliche UV-Wert, D die Verzerrungsstärke, $d = \|\vec{u} - \vec{u}_{mouse}\|$ der Abstand zu einem Punkt (hierbei der Mausposition), t die Zeit und $\hat{d}$ der normierte Richtungsvektor.

2.5 Wellen Verzerrung

Die Wellen Verzerrung wird hier als eine Funktion beschrieben werden, welche sich je nach Koordinaten der Pixel p und der Zeit t verändert. Dabei werden Sinusfunktionen verwendet, um oszillierende Muster zu erzeugen:

$$P(x, y, t) = \sin(x + \sin(y + t)) \cdot \sin\left(\|(x, y)\| \cdot 4 - t \cdot 3\right)$$

Die Kombination aus Sinus- und Kosinusfunktionen sowie der Abstand $\|(x, y)\|$ sorgt hierbei für das typische dynamische Muster.

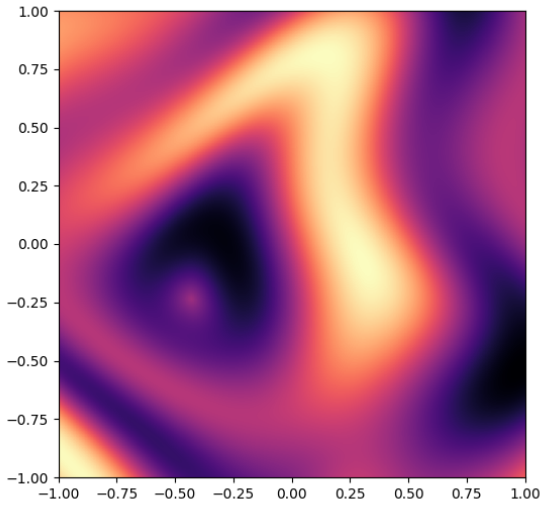

Abbildung 2.2: Wellenverzerrung

In Abbildung 2.2 wird eine Heatmap zur gezeigt welche die Wellenverzerrung repräsentiert.

2.6 Dynamische Partikel

Die Partikel werden als kleine Punkte auf dem Bildschirm dargestellt, die sich über den Bildschirm bewegen. Dieser Algorithmus dient zu Berechnung derer Bewegung und Intensität der einzelnen Partikel.

Dabei wird für jeden Partikel i in einer Menge von Partikeln (mit $i = 1, 2, \ldots, N$) eine Position $\vec{p_i}$ berechnet. Diese wird hierbei relativ zur Mausposition $\vec{m}$ durch das Hinzufügen eines dynamischen Offsets berechnet:

$$\vec{p_i} = \vec{m} + \left(\cos(i \cdot \frac{2\pi}{N} + t) \cdot \sin(i \cdot \frac{2\pi}{N} + t) \right)$$

Die Intensität eines Partikels wird auf der Basis der Inversen quadratischen Entfernung zum aktuellen Pixel ($\vec{u}$) berechnet:

$$I_i(\vec{u}) = \frac{1}{d^2}$$

wobei $d = \|\vec{u} - \vec{p_i}\|$ den Abstand zwischen dem aktuellen Pixel und dem Partikel darstelltt.

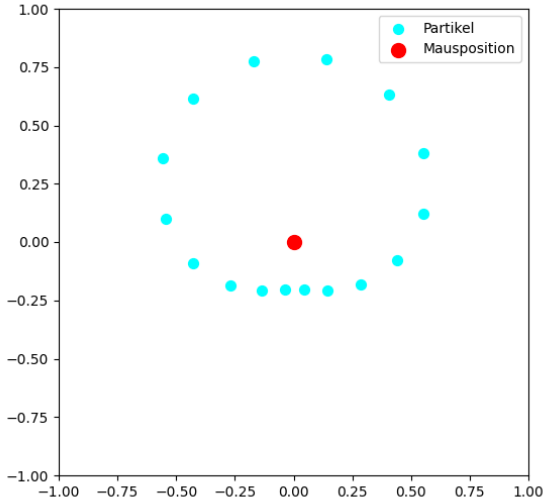

Abbildung 2.3: Partikelverzerrung

In Abbildung 2.3 kann sind die Partikel zu sehen welche die Maus umgeben und wodurch die Partikelanimation geschaft wird.

2.7 Farbkomposition

Die verschiedenen visuellen Elemente (Wellenverzerrung, Partikel, Gitter) werden kombiniert, um das finale Bild zu erstellen. Dabei werden die Farbwerte werden durch eine lineare Interpolation berechnet:

$$C = \alpha \cdot C_1 + (1 - \alpha) \cdot C_2$$

wobei C_1 und C_2 die Farbkomponenten der verschiedenen Effekte (z.B. Wellenverzerrung, Gitter) sind, und α ein Gewichtungsfaktor, der steuert, wie stark die einzelnen Effekte zur endgültigen Farbe beitragen.

Für den Randbereich des Bildes wird eine sanfte Übergangsfunktion verwendet, um den Effekt weicher und realistischer zu gestalten, wie zum Beispiel die **smoothstep**-Funktion:

$$f(x) = \text{smoothstep}(a, b, x) = \frac{1}{2}\left(1 - \cos\left(\pi \cdot \frac{x - a}{b - a}\right)\right)$$

Dadurch wird der Übergang geglättet und es sorgt dafür dass Farben an den Kanten sanft abklingen.

3 JavaScript & WebGL

WebGL (Web Graphics Library) ist eine JavaScript-API zur Darstellung von interaktiven 2D- und 3D-Grafiken im Webbrowser. Durch die direkte Nutzung der Grafikhardware (GPU) ermöglicht WebGL leistungsstarkes Rendering ohne zusätzliche Plugins oder Softwareinstallationen. Diese Technologie eignet sich insbesondere für die Entwicklung von Grafikanwendungen, Animationen, Simulationen und Spielen innerhalb des Browsers.

3.1 JavaScript

JavaScript ist die am weitesten verbreitete Programmiersprache für WebGL-Anwendungen. Sie bietet viele Vorteile für die Entwicklung von Grafik- und 3D-Anwendungen im Web. Zudem ist sie in allen modernen Webbrowsern integriert, so dass keine zusätzlichen Installationen oder Plugins notwendig sind. Dadurch können WebGL-Anwendungen unabhängig vom Betriebssystem plattformübergreifend eingesetzt werden.

WebGL kann auch mit HTML5 und CSS kombiniert werden, um interaktive Webseiten und Anwendungen zu erstellen. Durch diese Kombination ist eine einfache Integration von 3D-Grafiken in bestehende Webprojekte möglich und verbessert so die User Experience durch dynamische Inhalte.

JavaScript verfügt zudem über eine große Entwicklergemeinde, die eine Vielzahl von Ressourcen zur Verfügung stellt, darunter Frameworks wie Three.js. Diese Bibliothek erleichtert die Arbeit mit WebGL erheblich und ermöglicht eine schnellere Entwicklung komplexer grafischer Anwendungen. Darüber hinaus können WebGL-Anwendungen über JavaScript effizient mit Benutzereingaben und anderen Web-Technologien verknüpft werden. Dies erleichtert die Implementierung interaktiver Funktionen wie Maus- und Tastatureingaben, Animationen und Echtzeit-Rendering. Damit stellt JavaScript

eine leistungsfähige Lösung für die Entwicklung moderner, grafisch anspruchsvoller Webanwendungen dar.

3.2 WebGL Code

3.2.1 Erstellung des Gitters

Der Gitterabschnitt des Codes erzeugt ein rotierendes, hexagonales Muster, das sich über die gesamte Fläche erstreckt. Es beginnt mit der Transformation der UV-Koordinaten:

- **Skalierung und Rotation**: Die UV-Koordinaten werden mit einem Faktor von 4.0 skaliert und um einen festen Winkel $180°$ (hier 0.5 Bogenmaß) rotiert. Das sorgt für eine rotierte Gitterstruktur mit Hexagonen:

```
vec2 gridUV = rotate(uv * 4.0 + vec2(u_time *
    verzerrungsGeschwindigkeit), 0.5);
```

- **Fraktalisierung**: Mit `fract()` wird der Dezimalteil der transformierten UV-Koordinaten extrahiert damit das Gitter periodisch wiederholt werden kann und duch das Abziehen von 0.5 wird das Gitter symmetrisch um den Ursprung verschoben:

```
vec2 g = fract(gridUV) - 0.5;
```

- **Hexagonale Struktur**: Der Abstand eines Punktes von der Mitte jedes Hexagons wird mit der Funktion `hex(g)` berechnet. `smoothstep()` sorgt für weiche Übergänge zwischen den Zellen des Gitters:

```
float hexMuster = 1.0 - smoothstep(0.0, 0.2, hex(g)-0.3);
```

Das Ergebnis ist ein rotiertes, hexagonales Gitter mit weichen Übergängen.

3.2.2 Erstellung der Wellenverzerrung

Die Wellenverzerrung erzeugt das dynamische, pulsierende Muster. Der Prozess umfasst die Modifikation der UV-Koordinaten sowie die Anwendung von Sinusfunktionen für wellenartige Bewegungen:

- **Skalierung und Zeitfaktor**: Die UV-Koordinaten werden mit dem Faktor 1.5 multipliziert, während der Zeitfaktor u_time für die kontinuierliche Bewegung sorgt:

```
vec2 p = uv * 1.5 + vec2(u_time*verzerrungsGeschwindigkeit);
```

- **Verzerrung durch Sinus- und Kosinusfunktionen**: Die Sinus- und Kosinusfunktionen werden in einer Schleife verzerrt, wobei diese mit jeder Iteration abschwächt:

```
for(int i=1; i<4; i++) {
    p.x += 0.3/float(i) * sin(float(i)*p.y + u_time);
    p.y += 0.3/float(i) * cos(float(i)*p.x + u_time);
}
```

- **Verzerrungs-Intensität**: Die Intensität der Verzerrung wird durch die Kombination von Sinusfunktionen und der Entfernung vom Ursprung (`length(p)`) gesteuert:

```
float welle = sin(p.x + sin(p.y + u_time)) * sin(length(p)*4.0 -
    u_time*3.0);
```

Dieser Abschnitt erzeugt ein sich kontinuierlich veränderndes Muster, das durch die Verzerrungen und die Intensitätsberechnungen dynamisch lebendig wirkt.

4 Fazit und Ausblick

Während der Entwicklung des Shaders gab es Herausforderungen, insbesondere bei der Optimierung der Performance und der Abstimmung der Shader-Parameter. Durch gezielte Anpassungen konnten jedoch flüssige Animationen erreicht werden. Jedoch wurden einige Erfolge durch Versuch und Irrtum erzielt, da an bestimmten Stellen das erforderliche Wissen zunächst fehlte. Außerdem wurde das Projekt von verschiedenen Quellen inspiriert, die als kreative Einflüsse dienten (z.B. Shadertoy & The Book of Shaders). Letzendlich zeigt das Projekt die Leistungsfähigkeit moderner GPU-Programmierung für interaktive Simulationen.

Zukünftige Projekte könnten zusätlich komplexere und fortgeschrittenere Shader-Techniken beinhalten. Außerdem wäre eine Erweiterung um Benutzersteuerungselemente denkbar.