|  |
| --- |
|  |

**Shader in OpenGL**

**Computer Graphik Projekt**

des Studiengangs Informationstechnik an

der Dualen Hochschule Baden-Württemberg Stuttgart

21.12.2022

|  |  |
| --- | --- |
| **Matrikelnummer** | 8803945, XXX |
| **Kurs** | TINF20IN |

Inhaltsverzeichnis

[Abkürzungsverzeichnis III](#_Toc122593120)

[1 Einleitung 1](#_Toc122593121)

[1.1 Ausgangssituation und Problembeschreibung 1](#_Toc122593122)

[1.2 Zielsetzung 1](#_Toc122593123)

[1.3 Vorgehensweise 1](#_Toc122593124)

[1.4 Aufbau der Arbeit 1](#_Toc122593125)

[2 Grundlagen 2](#_Toc122593126)

[3 Anforderungen 3](#_Toc122593127)

[4 Implementierung 4](#_Toc122593128)

[5 Zusammenfassung und Ausblick 5](#_Toc122593129)

[Abbildungsverzeichnis IV](#_Toc122593130)

[Tabellenverzeichnis V](#_Toc122593131)

[Literaturverzeichnis VI](#_Toc122593132)

[Anhang VII](#_Toc122593133)

Abkürzungsverzeichnis

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | PLM |  | Product Lifecycle Management |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Einleitung

## Zielsetzung

## Vorgehensweise

## Aufbau der Arbeit

## Anforderungen

## Schreiben Sie einen GLSL Shader in WebGL und dokumentieren Sie seine Funktionsweise. Bewertet wird Kreativität, Verständlichkeit der Dokumentation und die Implementierung. Aus der Dokumentation sollte hervorgehen, auf welchen theoretischen Prinzipien das bildgebende Verfahren basiert, wie es algorithmisch umgesetzt werden kann und wie es implementiert wurde. Als Orientierung für eine sehr gute Dokumentation dient <https://thebookofshaders.com/>. Ein Shader wird als nicht sonderlich kreativ bewertet, wenn er nur eine geringe Variabilität zu Standard Beispielen aus gängiger Literatur oder Seiten wie <https://thebookofshaders.com/> oder https://www.shadertoy.com/ aufweist. (100 Punkte)

Stichpunkte

* Shader implementieren
* Funktionsweise Dokumentieren
  + Theoretische Prinzipien
  + Algorithmische Umsetzung
  + Implementierung

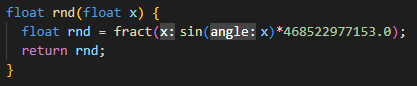
# Grundlagen

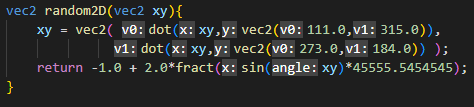
## Zufalls Zahlen

Das erste theoretische Prinzip ist das generieren von Zufälligen Zahlen. Dieser Begriff bedeutet dass jede Zahl aus einem Feld mit der gleichen Wahrscheinlichkeit ausgewählt wird. Es existieren nur dann Zufällige Zahlen, wenn Werte der Zahlen gleichmäßig über einen Bereich verteilt sind und es unmöglich ist auf Zukünftige Zahlen zu schließen mit den schon erhaltenen Zahlen. Zufällig generierte Zahlen finden in vielen Bereichen großen Einsatz, für diese Dokumentation ist allerdings nur wichtig wie diese mit den Einschränkungen einer Programmiersprache umsetzbar sind. In vielen Programmiersprachen ist dies bereits durch eine hinterlegte Funktion einfach lösbar. In WebGL muss dieses Problem zuerst gelöst werden.

Algorithmisch: Es gibt mehrere Möglichkeiten wie Algorithmisch Zufällige Zahlen generiert werden können. Die in dieser Dokumentation gewählte verwendet den sin() um einen Bereich an Zahlen zu haben, aus dem die Zufälligen Zahlen ausgewählt werden können. Dieser besteht aus Flies Komma-Werten. Von diesen Werten wird nur der Bruchteil der Flies Komma Zahlen benötigt. Dies sind noch keine Zufälligen Zahlen, aber wenn die Sinuswelle mit Zahlen größer 100000 Multipliziert wird kann die Sinuswelle nicht mehr erkannt werden und man erhält pseudo Zufällige Zahlen. Pseudo Zufällige Zahlen reichen für diesen Shader vollständig aus.

(vgl. Book of Shaders)

Implementiert: Um die Zufälligen Zahlen zu Programmieren wurde eine Funktion erstellt, diese besteht aus einem sin() der mit einer Zahl über 100000 multipliziert wird und der schon verfügbaren Funktion fract(). Diese Funktion gibt den Bruchteil der Float Zahl zurück.

Die Zufallsmethode reicht für diesen Shader in dieser Form nicht aus, da diese Funktion nur 1-Dimensionelle Zufallswerte generieren kann und 2-Dimensionale Werte also Vektoren benötigt werden. Um dies zu erreichen wird die dot() Funktion verwendet. Diese gibt einen float Wert zwischen 0.0 und 1.0 zurück je nachdem wo sich zwei Vektoren im Verhältnis zueinander befinden. Nachdem einer Variablen der Wert der Funktion dot() zugewiesen wird, ist der Rest für die 2D Zufalls Funktion gleich wie darüber beschrieben.

Durch diese Funktion werden die Zufälligen 2D Werte für den Shader generiert.

## Noise

Das nächste verwendete Prinizip ist Noise. Noise ist eine Technik, die in der Shaderentwicklung verwendet wird, um zufällige Muster und Texturen zu generieren. Diese Technik ähnelt der Verwendung von Random-Funktionen, da das Verhalten von Noise häufig als zufällig betrachtet wird. Die Generierung von Noise erfolgt durch Zerlegung eines float-Werts in seinen Ganzzahl- und Bruchteilteil mithilfe der Funktionen floor() und fract(). Anschließend wird die Zufallsfunktion mit diesen Werten aufgerufen und das Ergebnis wird mithilfe der Funktion mix() berechnet. Noise kann in vielen Bereichen des täglichen Lebens beobachtet werden, wie beispielsweise in Wellen im Meer oder Wolken am Himmel.

\*\*\*Bild von Noise-Algorithmus\*\*\*

Allerdings wird für diesen Shader ein 2-Dimentionaler Noise Wert benötigt der durch diese Funktion nicht generiert werden kann. Um einen solchen 2D-Noise-Wert in einem Shader zu generieren, ist es notwendig, einen vec2 anstatt eines float-Parameters zu verwenden. Der grundlegende Ablauf der Noise-Generierung bleibt jedoch gleich, wie im folgenden Bild dargestellt ist. Es werden lediglich Anpassungen vorgenommen, um den vec2 statt des float zu verwenden. Die Funktionen floor() und fract() werden zur Zerlegung des vec2 in seine Ganzzahl- und Bruchteilteile verwendet, bevor die Zufallsfunktion aufgerufen und das Ergebnis mithilfe der Funktion mix() berechnet wird. Durch die Verwendung eines 2D-Noise-Werts können realistischere Texturen und Muster in einem Shader erzeugt werden.

\*\*\*Bild von 2D-Noise-Algorithmus\*\*

## Formen

Theorie:

### Es gibt viele verschiedene Geometrische Formen, dazu zählen zum Beispiel verschiedene Vielecke aber auch Ellipsen oder ein Kreis. Um diese Formen zu erstellen oder zu beschreiben gibt es in der Mathematik verschiedene Formeln. Diese Formen können analog aber auch digital erstellt werden. Dies ändert nichts an den Mathematischen Formeln oder den Eigenschaften für diese Formen. Um einen Kreis zeichnen zu können wird in der analogen Welt ein Zirkel verwendet. Durch das einstechen der Nadelspitze des Zirkels wird der Mittelpunkt des Kreises bestimmt und durch bewegen des Zirkelschenkels wird der Radius des Kreises festgelegt. Im Digitalen geschieht das auf ähnliche Weise nur wird anstatt eines Zirkels Funktionen verwendet um auf die gewünschten Formeln zu kommen.

### Um einen Kreis in WebGL zu generieren wird wie in der analogen Welt ein Mittelpunkt benötigt. Um danach bei jedem Pixel feststellen zu können ob es innerhalb des Kreises liegt. Um diese Aussage treffen zu können wird die Distanz, oder auch Radius genannt, gebraucht. Diese Distanz kann in WebGL auf verschiedene Weise berechnet werden. Sie könnte zum Beispiel über die Mathematische Formel berechnet werden, allerdings kann auch eine schon existierende Funktion zur Berechnung einer Distanz verwendet werden wie distance() oder length(). Als letzten Schritt muss eine Step() oder Smoothstep() Funktion verwendet werden um den Bereich in dem der Kreis angezeigt werden soll auf den Radius des Kreises zu beschränken.

Algorithmisch:

Algorithmisch gesehen besteht der Kreis aus einem Mittelpunkt, der über eine Formel bestimmt wird, und den Abständen der Pixel zu diesem Mittelpunkt. Um diese Abstände zu bestimmen kann die Formel bestimmt werden.

# Bausteine des Shaders

## Hintergrund

Verwendete theoretische Prinzipien: Random, Noise, brownianMotion, fraktioneleBrownianMotion

## Viel-Ecke

Um die Viel-Ecke des Shaders zu generieren werden keine weiteren Theoretischen Prinzipe außer den im zweiten Kapitel beschriebenen benötigt. Die Funktion CalculateForms berechnet die Viel-Ecke in einem 2D-Raum anhand der gegeben Seitenanzahl und der Koordinaten. Diese werden als Parameter der Funktion mitgegeben. Als erstes verschiebt die Funktion die Koordinaten des Parameters currentCoords auf das Koordinatenfeld so, dass diese den Bereich (-1|1) abdecken. Dies ist wichtig um das Viel-Eck wiederholt über das Koordinatensystem abbilden zu können. Hiernach werden zwei Winkel berechnet. Der erste Winkel ist zwischen dem Ursprung und den gegebenen Koordinaten. (BILD)

Als zweites muss der Winkel zwischen den Seiten des Viel-Ecks berechnet werden. (BILD) Dieser Winkel ist besonders wichtig, da dieser dafür sorgt, dass die Form des Vielecks Geometrisch gleichmäßig ist. Also alle Winkel innerhalb des Viel-Ecks gleich groß sind.

Im letzten Schritt wird die Distanz zwischen dem Ursprung und den übergebenen Koordinaten bestimmt. Die Distanz wird dazu verwendet, dass das Viel-Eck an der richtigen Position erstellt wird. (BILD)

Die Funktion gibt einen vec3 zurück. Dieser besteht aus einem smoothstep, der die eben berechnete Distanz verwendet um einen schöneren Übergang zwischen dem Viel-Eck und dem Hintergrund zu generieren. (BILD)

## Bewegtes Muster

- Funktionsweise Dokumentieren

o Gedanke hinter Baustein

o Theoretische Prinzipien

o Algorithmische Umsetzung

o Implementierung

Die Idee hinter dem bewegten Muster war, die Dynamik ständiger Bewegung mit der Schönheit eines gleichmäßigen Musters zu verbinden. Wir kamen daher auf die Idee, ein Raster aus Formen anzuzeigen. Jede dieser Formen sollte ähnlich, aber nicht gleich aussehen. Dadurch kamen wir auch auf die Füllung der Formen mit Inhalt. Die Formen sollten daher gleich aussehen. Um dies Umzusetzen, überlegten wir uns ein Raster aus eigenen kleinen Koordinatensystemen zu erstellen. Die Dynamik dieser Formen war die eigentliche Herausforderung, da sie nicht zu gleichmäßig sein sollte, da größere Abwechslung spannender ist. Es war also klar, dass sich die Richtung der Bewegung ändern sollte, wobei eine 180 Grad Wendung vermieden werden sollte, da es eine zu große Unterbrechung darstellen würde.

Das Muster ist in zwei Teile aufgeteilt, die Erstellung eines Rasters und die Bewegung dieses Rasters.

1.1.1 Rastererstellung

Diesem Schritt liegt die Transformation eines Koordinatensystems in mehrere kleineren Koordinatensysteme zu Grunde. Dabei

1.1.2 Fließenbewegung

## Kunst in den Viel-Ecke

Die theoretische Prinzipen auf der dieser Baustein des Shaders basiert wurden schon im Kapitel Grundlagen genauer beleuchtet. In diesem Kapitel wird die Implementierung genauer betrachtet.

### calculateShapeForm

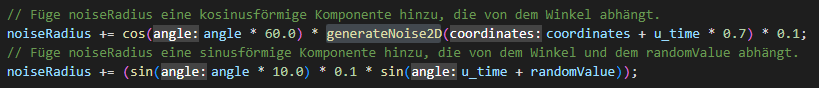
Die „Kunst“ innerhalb der Viel-Ecke basiert auf der Form eines Kreises und wird in der Funktion calculateShapeInForms erstellt. Ein Kreis benötigt einen Mittelpunkt, wie es im Kapitel Grundlagen erklärt ist. Da dieser Kreis sich immer mittig innerhalb des Koordinatenfeldes befinden soll, wird zuerst das Koordinatenfeld um 0.5 nach Links unten verschoben. Damit wird erreicht, dass das Koordinatenfeld nicht mehr, wie standardmäßig, von (0|0) bis (1|1) geht. Sondern von (-0.5|-0.5) bis (0.5|0.5). Das Ergebnis dieser Verschiebung ist das der Mittelpunkt des Kreises in der Mitte des Koordinatensystems bei (0|0) liegt. (BILD)



In der nächsten Zeile "float distanceToOrigin = length(coordinates) \* 2.0;" wird die Distanz des übergebenen Punktes (mit den Koordinaten "coordinates") zum Ursprung (dem Punkt (0,0)) brechnet. Der Rückgabewert der Funktion "length" ist die Länge des Vektors, der durch den Punkt "coordinates" definiert wird. Der Faktor "2.0" wird anschließend multipliziert, um die Distanz zu verdoppeln. Wenn der Faktor größer als 2.0 gestellt wird, wird die Distanz des Punktes zum Ursprung entsprechend vergrößert. Dies hat zur Folge, dass der Rückgabewert der Funktion kleiner wird, da der Rückgabewert der Funktion "smoothstep" (1.0 - smoothstep(noiseRadius, noiseRadius + 0.007, distanceToOrigin)) kleiner wird, wenn die Distanz größer wird.

Die Funktion "smoothstep" gibt einen glatt verlaufenden Wert zwischen 0 und 1 zurück, der auf Basis der übergebenen Grenzen (hier: noiseRadius und noiseRadius + 0.007) und des übergebenen Parameters (hier: distanceToOrigin) berechnet wird. Wenn der übergebene Parameter größer als die obere Grenze ist, wird der Rückgabewert 1.0 sein. Wenn der übergebene Parameter kleiner als die untere Grenze ist, wird der Rückgabewert 0.0 sein. Wenn der übergebene Parameter zwischen den Grenzen liegt, wird ein glatt verlaufender Wert zwischen 0 und 1 zurückgegeben. Durch diesen Schritt werden harte Sprünge zwischen dem Kreis und dem Hintergrund verhindert.

Durch das setzen des Mittelpunkts und der Berechnung des Radius kombiniert mit dem verwenden einer Step Funktion existiert zu diesem Zeitpunkt ein Kreis. Allerdings verändert dieser seine Form noch nicht. Um dies zu erreichen wird der Winkel zum Ursprung berechnet. Dieser ermöglicht es unterschiedliche Punkte des Kreises zu beeinflussen. Der Winkel wird durch die Funktion im folgenden Bild berechnet. 

Anschließend werden in Abhängigkeit des Winkels cos und überlagernde sin Werte auf den noiseRadius addiert oder subtrahiert. Dies bewirkt eine Wellenartige Verzerrung des Kreises. Damit es zu keiner gleichförmigen Welle über den ganzen Kreis kommt, wird der cos mit der generateNoise2D Funktion multipliziert. Ebenfalls wird der sin mit dem Ergebnis der randomValue Funktion multipliziert. In beiden Fällen wird die Zeit also u\_time zusätzlich multipliziert. Dies hat zur Folge, dass sich die nun Zufälligen Wellen des Kreises bewegen und das diese Bewegung sich um den Kreis herum fortbewegt. Als letztes wird der sin und cos noch mit 0.1 multipliziert um die Werte in eine brauchbare Größe zu wandeln. Dies ist im nächsten Bild zu sehen.

### Widerholungen und Pulsieren

Da nicht nur ein Kreis in den Vielecken aufgerufen werden soll, wird eine For-Schleife verwendet. Diese ermöglicht es Kreise innerhalb eines Kreises zu generieren. Hierzu bekommt der erste Kreis den größten Radius, dies entspricht 0.9. Um nun Kreise innerhalb des ersten Kreises zu erstellen muss der Radius kleiner werden. Hierzu wird die Laufvariable i der For-Schleife mit 0.23 multipliziert und von dem Radius abgezogen. Somit hat der zweite Kreis einen Radius von 0.67, der dritte 0.44 und der vierte somit letzte Kreis 0.21. Somit werden insgesamt vier Kreise erstellt. (BILD)

Bei dem Erstellen der Kreise wird die Funktion shapeBorderWidth verwendet. Diese besitzt als Parameter width. Mit dem Parameter wird die breite des Randes von dem Kreis bestimmt. Um nun einen Pulsierenden Rand zu erhalten, wird für den Parameter width der positive Teil der sin Welle verwendet. Der sin verändert sich durchgehen durch das Verwenden von u\_time. Um die Geschwindigkeit des Pulsierens anpassen zu können wird die Zeit u\_time mit der Variable pulseSpeed multipliziert. (BILD)

### Farbenhintergrund

Den Farbenteil im Hintergrund von den Vielecken ist im Internet unter folgendem Link zu finden [https://glslsandbox.com/e#98242.0]. Die Grund Funktion des Farben Shaders wurde größtenteils unverändert übernommen. Allerdings wird für jedes Vieleck andere Farben angezeigt und bei jedem Vieleck werden die Variablen zum verändern des Farben Shaders unterschiedlich gewählt um ein größt möglichen Unterschied zwischen den verschiedenen Vielecken zu generieren.

# Implementierung

VSCode

Wie man es ausführt:

Um den Shader ausführen zu können muss zuerst Visual Studio Code (VSCode) installiert werden. Nachdem VSCode installiert ist, muss die Extension LiveServer installiert werden. Dies geht indem im linken Teil von VSCode die Extensions ausgewählt werden und nach dem LiveServer gesucht wird. Nach dem installieren des Live Servers muss der Ordner mit unserem Shader in VSCode geöffnet werden. Als letztes muss in der unteren rechten Ecke des Fensters, der live Server gestartet werden. In diesem kann der Shader betrachtet werden.

Extentions

LiveServer / glslCanvas / …

Abbildungsverzeichnis

[Abbildung 2‑1: PLM-Kreislauf, basierend auf [2] 2](#_Toc59527846)

Tabellenverzeichnis

[Tabelle 2‑1: Übersicht der HTTP-Methoden, basierend auf [17, S. 604] 2](#_Toc59527850)

Literaturverzeichnis

[1] Werkzeugmaschinenlabor WZL der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen (Hg)., *Product Lifecycle Management (PLM).* [Online]. Verfügbar unter: http://www.plm-info.de/de/d7bc5950658c5c08c1257e99002f4d59.html (Zugriff am: 2. Juli 2020).

[2] Andrew S. Tanenbaum und Maarten van Steen, *Verteilte Systeme Prinzipien und Paradigmen*. Pearson Studium 2008, 2008.

Anhang

[1 Auszug aus … VIII](#_Toc59527951)

[2 Klassendiagramm IX](#_Toc59527952)

1. Auszug aus …