****

**Cumputergraphik Abgabe**

**Aufgabe1: Shader**

von

**Gregori Daiger**

Abgabedatum 13.06.2024

**Matrikelnummer, Kurs** 5611072, 2395489 STG-INF21IN

**1. Idee und Zielsetzung**

Das Ziel des Projekts war die Schaffung eines interaktiven, visuell ansprechenden, fiktiven Universums. Dieses Universum beinhaltet Himmelskörper wie Planeten und Kometen, die sich realistisch bewegen und miteinander interagieren können. Ein Highlight sind die Kollisionen und Explosionen, die für visuelle Spektakel sorgen. Das Projekt ist so gestaltet, dass es mit weiteren Ideen flexibel erweitert werden kann.

**2. Verwendete Technologien**

Die Entwicklung begann mit dem "Book of Shaders Editor" für die grundlegende Gestaltung des Universums. Aufgrund der Einschränkungen dieses Editors, insbesondere bei der Implementierung von dynamischen Effekten wie Explosionen, wurde später JavaScript in Verbindung mit Three.js integriert. Diese Kombination ermöglichte eine erweiterte Steuerung und Interaktivität, indem zusätzliche Variablen und ein effektiveres Zustandsmanagement integriert wurden. Der GLSL-Code des Fragment-Shaders blieb dabei das Herzstück des Projekts.

**3. Thoeretische Grundlagen und Konzeption der Welt**

In diesem Projekt wird ein fiktives Universum als 2D-Welt umgesetzt. Das Ziel ist es, Planeten als farbige Kreise innerhalb eines 2D-Koordinatensystems darzustellen. Dieses Koordinatensystem bildet den Bildschirm nach, wobei die untere linke Ecke den Punkt (x = 0.0, y = 0.0) und die obere rechte Ecke den Punkt (x= 1.0, y= 1.0) repräsentiert.

Um dies zu realisieren, wird hauptsächlich ein Fragment Shader verwendet. Ein Fragment Shader ist Teil der Grafik-Pipeline in WebGL und dient dazu, die Farbe und andere Eigenschaften jedes Pixels auf dem Bildschirm zu bestimmen (vgl. <https://shader-tutorial.dev/basics/fragment-shader/>). In unserem Fall wird er verwendet, um das 2D-Universum zu rendern.

Im Code wird das 2D-Koordinatensystem durch die Variable **st** repräsentiert, welche die Koordinaten jedes Pixels auf dem Bildschirm enthält. Die ersten zwei Zeilen in der main-Funktion des Fragment Shaders berechnen diese Koordinaten:

Ein Bild, das Text, Schrift, Screenshot, Grafiken enthält.

Automatisch generierte Beschreibung

Hier wird die Position jedes Fragments (**gl\_FragCoord**) durch die Auflösung des Bildschirms (**u\_resolution**) geteilt, um die Koordinaten im Bereich von 0 bis 1 zu normalisieren (vgl. <https://thebookofshaders.com/03/>). Die Anpassung des x-Wertes ist notwendig, um das Seitenverhältnis des Bildschirms zu berücksichtigen und Verzerrungen zu vermeiden.

Der Schlüssel zur Darstellung der Welt liegt in der **gl\_FragColor**-Variable. Sie bestimmt die Farbe jedes Pixels und somit das Erscheinungsbild des gesamten Universums. Durch die Manipulation davon im Fragment Shader können verschiedene visuelle Elemente wie Planeten, Sterne und andere kosmische Objekte erzeugt werden.

Also zusammenfassend kann man sagen, dass jeder Pixel den Code betritt, dann normalisiert wird um es so auf dem Koordinatensystem abzubilden. Mit der auf dem 2D-Koordinatensystem basierenden Logik kann dann abhängig von der Pixelpositioion des aktuell betrachteten Pixels eine Farbe zugewiesen werden und so die Welt nur auf Basis der Farben des Fragement-Shaders gerendert werden.

Die Transforation durch Normalisierung kann man sich so vorstellen:

**Ein Bild, das Text, Diagramm, Reihe, Schrift enthält.

Automatisch generierte Beschreibung**

**Darstellung von Objekten in dem Universum**

Mithilfe der Normalisierung der Pixel in ein 2D-Koordinatensystem das quadratisch ist, können Pixel nun verarbeitet und mathematisch so berechnet werden, dass Objekte und Bewegungen simuliert werden können.

Da es sich bei den Objekten des geplaneten Universums um Planeten, Sonnen, Sterne, Kometen und einen einfachen PacMan handeln soll, können für die Darstellung Kreise verwendet werden, die sich durch ihre Farben und Farbeffekte unterscheiden.

Für die Darstellung der Kreise wird folgende selbst-definierte Methode verwendet

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Die drawObject-Methode bekommt die Posisition des aktuell betrachteten Pixel im normierten Format (**st),** den Mittelpunkt des Kreise (**position)** als Datentyp vec2, was einer 2D-Koordinate entspricht. Der Radius des Kreises wird als float übergeben (**radius**) und die gewünschte Farbe als vec3, was einer RGB-Farbe entspricht.

Ein Bild, das Text, Schrift, Diagramm, Handschrift enthält.

Automatisch generierte Beschreibung

Die obenstehende Skizze soll verdeutlichen wie damit ein Kreis gezeichnet werden kann.   
Die vom book-of-shaders eingebaute Funktion(distance) kann den Abstand von zwei Punkten berechnen. Das wird ganz einfach mithilfe des Satz des Pythagoras gemacht. Die Funktion drawObject überprüft für den einkommenden Pixel (**st**) ob er im Radius (**radius**) des Mittelspunkts (**position**) liegt, wenn das der Fall ist wird für diesen Punkt die übergebene Farbe (**color**) festgelegt und später gerendert. Wenn das nicht der Fall ist, dann wird die Farbe schwarz zurückgegeben, was der Hintergrundfarbe entspricht und somit, als unsichtbar erscheint.

**Erzeugung des Hintergrunds**

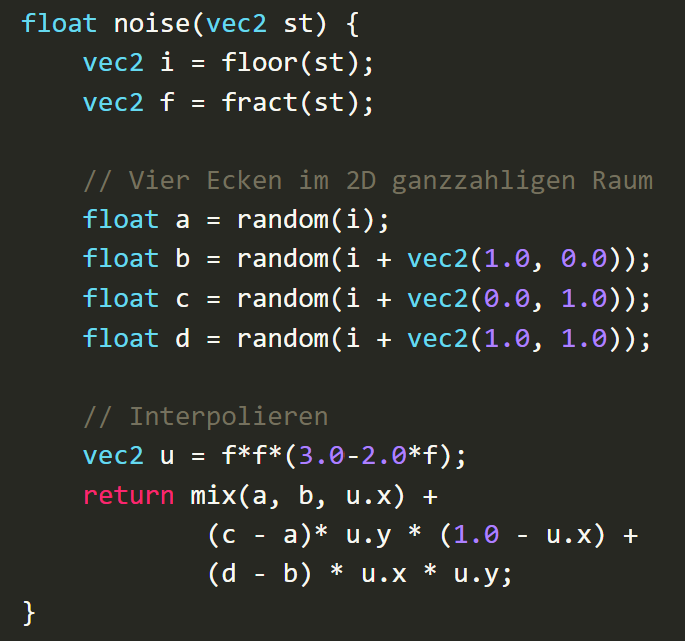
In der Schaffung eines Universums spielt der Hintergrund eine entscheidende Rolle. Er ist das Fundament, auf dem die Elemente unseres virtuellen Kosmos zur Geltung kommen. Um den Eindruck eines Universums auf unserer 2D-Szene zu schaffen, sollen zwei Effekte eingebaut werden: Viele kleine Sterne und ein schimmernder Nebel.

Die Sterne entstehen durch eine zufällige Verteilung von kleinen, leuchtenden Punkten, die Sterne darstellen. Dies wird erreicht, indem für jeden Pixel zufällig entschieden wird, ob er mit einem gelblichen Schimmer erstrahlt oder nicht. Dazu wird eine rand-Funktion definiert, die als Eingabe die Koordinate des Pixel bekommt und davon abhängig einen Wert zurückgibt. Ist dieser Wert größer als ein vorgegebener Schwellenwert, in unseren Fall 0.98, soll an diesem Pixel ein Stern aufleuchten. Durch die Erstellung eines Pseudzufallswerts können so viele kleine Sterne an unterschiedlichen Punkten auf dem schwarzen Hintergrund dargestellt werden.

Ein Bild, das Text, Screenshot, Schrift, Grafiken enthält.

Automatisch generierte Beschreibung

Die Random-Methode zur Erzeugung der Zufallszahl stammt aus der Dokumentaion aus Book of Shaders (<https://thebookofshaders.com/10/>). Die Zufallszahlen basieren auf einer Mischung aus mathematischen Eigenschaften und empirischer Erprobung. Die Zahlen 12.9898 und 78.233 sind dabei die Eingabewerte für die **sin**- und **dot**-Funktionen, die zusammen mit der Zahl 43758.5453 eine Pseudozufallszahl erzeugen. Diese spezielle Kombination von Zahlen erzeugt eine verteilte Sequenz von Werten, die für visuelle Effekte nützlich ist, insbesondere in einem Kontext, in dem echte Zufallszahlen nicht verfügbar sind. Das ist auch beim Book-of-Shaders-Editor der Fall, weshalb diese Methode die optimale Lösung für das Erschaffen der zufälligen Sterne ist.

Nachdem die Sterne mithilfe der random-Funktion simuliert werden können, gilt es noch den Nebel nachzustellen. In der Welt der Shader-Programmierung gibt es dafür das Prinzip **noise**, womit man beispielsweise Naturereignisse aus der realen Welt gut nachbilden kann. Für den Nebel kann daher das von Ken Perlin entwickelte Noise-Prinzip in der 2D-Welt optimal genutzt werden. (https://thebookofshaders.com/11/).  


Der obenstehende Code zeigt wie mihilfe des Pearl Noise ein Nebeleffekt in dem Unisversum erzeugt werden kann. Der Perlin-Noise basiert auf der Generierung von Zufallswerten an den Ecken von Quadraten (a-d) mithilfe der random-Funktion, die ein Gitter bilden. Durch die Interpolation zwischen diesen Werten entstehen glatte Übergänge innerhalb jedes Quadrats, wodurch ein natürlich aussehendes, kontinuierliches Rauschen erzeugt wird.

**Bewegungen der Objekte**

Nachdem nun der Hintergrund für die Atmosphäre sowie die Darstellung der Objekte als Kreise steht, geht es darum in der Szene die Objekte zu bewegen und geeignete Algorithmen zu verwendnen.  
Es sollen drei Planeten um die Sonne kreisen, zwei davon in einer Kreisbahn und der dritte in einer Ellipsenbahn.

Für die Erzeugung der Kreisbahn kann nach einem ähnlichen Prinzip ausgehend vom Zentrum aus innerhalb eines Radius eine Linie gezeichnet werden.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Folgende Skizze verdeutlicht wie mithilfe trigonometrischer Funktionen und dem Einheitskreisprinzip ein Objekt die Position eines Objekts auf einer Kreisbahn berechnet werden kann.  
Ein Bild, das Text, Diagramm, Handschrift, Schrift enthält.

Automatisch generierte Beschreibung

Jeder Punkt auf dem Kreis kann durch seinen Winkel am Kreismittelpunkt und dem Radius (was der Hypotenuse entspricht) dargestellt werden.

Die kontinuierliche Bewegung eines Punktes auf einer Kreisbahn kann durch die Verwendung der Zeitvariable **u\_time** als Winkelparameter simuliert werden. Dabei repräsentiert **u\_time** den aktuellen Winkel, und durch Multiplikation mit einem Geschwindigkeitsfaktor lässt sich die Umlaufgeschwindigkeit steuern. So wird der Winkel über die Zeit kontinuierlich verändert, was die Animation eines Objekts entlang der Kreisbahn ermöglicht. Die glsl-Methode sieht folgendermaßen aus:

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Auch für die Ellipsenbahn und die Bewegung darauf können die mathematischen Funktionen für Ellipsen im 2D-Koordinatensystem herangezogen werden:

Umgesetzt in glsl sieht das folgendermaßen aus:

Ein Bild, das Text, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

**Komet und Supernova**

Bisher konnte mit der Verwendung der oben genannten Prinzipien ein einfaches Universum dargestellt werden. Als Zwischenergebnis können mithilfe der oben beschriebenen Methoden beliebig Planeten hinzugefügt werden, die durch die Angabe des Radius der Farbe und der Wahl der Art der Orbittypen(Ellipse oder Kreis) Planeten um die Sonne kreisen, die durch die Erstellung eines Hintergrunds mit der random-Funktion und dem noise-Prinzip in einer realistsichen 2D-Univserumsatmosphäre ein Sonnensystem bilden.  
  
Um dem Unisversum mehr Dynamik und Diversität zu verleihen soll eine Supernova hinzugefüt werden und ein Komet der durch die Szene fliegt und an den Rändern abprallt.

Die Idee hinter der Simulation einer Supernova im Shader ist es, einen Stern zu schaffen, der zyklisch eine explosive Expansion durchläuft und anschließend wieder auf seine normale Größe zurückkehrt. Dies wird in einer kontinuierlichen Schleife wiederholt, sodass der Stern periodisch zwischen einem normalen und einem explosiven Zustand wechselt. Das wird doch folgende Funktion erreicht.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

**mod(time, period)** teilt die Zeit **u\_time** durch die Periodendauer und nutzt den Restwert, um den Zeitpunkt im Zyklus zu bestimmen. Wenn **cycleTime** kleiner als die **duration** ist, befindet sich der Stern in der Expansionsphase der Supernova (**isSupernova** ist wahr). Der Fortschritt dieser Phase (**progress**) wird als Verhältnis der vergangenen Zeit zur Gesamtdauer berechnet und begrenzt den Wert auf maximal 1.0.(durch **min-Funktion**).Die aktuelle Größe des Sterns (**currentSize**) wird dann basierend auf diesem Fortschritt berechnet, wobei sie während der Explosion mit der Zeit anwächst und danach wieder zur normalen Größe zurückkehrt.

Um den visuellen Effekt der Explosion zu erzeugen, werden die Funktionen **smoothstep** (https://thebookofshaders.com/glossary/?search=smoothstep) und **mix** (<https://thebookofshaders.com/glossary/?search=mix>) verwendet. **smoothstep** wird eingesetzt, um einen weichen Übergang zwischen der Stern- und Hintergrundfarbe zu schaffen, insbesondere am Rand des Sterns. Dies verleiht dem Stern ein glühendes Aussehen. Die **mix**-Funktion wird verwendet, um die Farbe des Sterns während der Explosionsphase zu mischen, wobei die Farbintensität basierend auf der **progress**-Variable abnimmt, was eine abnehmende Leuchtkraft simuliert.

Diese beiden Funktionen sind Standard-Tools in der Shader-Programmierung, die helfen, realistische visuelle Effekte zu erzielen und stammen aus der Doku vom The book of shaders.

Das nächste Element, dass dem Universum hinzugefügt werden soll ist ein freifliegender Komet, der am Bildschirmrand abprallt und so im Bild bleibt. Später soll er dem Universum mehr Dynamik verleihen, in dem Planeten die mit ihm kollidieren, ähnlich wie bei der Supernova explodieren.

Der Komet soll sich innerhalb des Bildbereichs bewegen und an den Rändern abprallen. Hierbei muss auch der Radius des Kometen berücksichtigt werden, damit der Komet nicht halb aus dem sichtbaren Bereich herausragt.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Die Methode **getCometPosition** berechnet die Position des Kometen in Abhängigkeit von der Zeit und berücksichtigt dabei die Logik für das Abprallen am Bildschirmrand. Mithilfe der Modulo-Berechung für die X und Y-Bewegung (**modifiedTimeX, modifiedTimeY**) wird dafür gesorgt dass die Koordinatenwerte zwischen 0.0 bis 2.0 – 2.0 \* **cometRadius** bleiben. Anschließend findet die Berechnung der Position auf Basis der zeitabhängigen Bewegungen statt.

Eine Koordinate befindet sich noch im Bildschirm wenn folgende mathematische Bedingung gegeben ist:

Wird der Wert überschritten heißt das, dass der Komet am Rand abprallen soll. Das wird realisiert, indem der Komet auf der entsprechende Achse sich nun in die entgegengesetzte Richtung bewegen soll. Dazu wird folgende Formel (beispielhaft an der X-Koordinate) verwendet:

Mithilfe dieser Logik lässt sich der Komet nur in Abhängigkeit von der Zeit **u\_time** bewegen und unter Berücksichtigung seines Radius am Rand abprallen. Die Parameter **cometSpeedX** und **cometSpeedY** beeinflussen nur die Geschwindigkeit des Kometen auf den entsprechenden Achsen, in dem sie mit der Zeit multipliziert werden.

**PacMan**

Um der Szene etwas Interaktivität mit dem Zuschauer zu verleihen, soll ein PacMan hinzugefügt werden der mit der Maus gesteuert werden kann. Dazu kann die vom book-of-shaders automatisch bereitgestellte uniform **u\_mouse** verwendet werden. (vgl. <https://thebookofshaders.com/03/>). Da der Bildschirm zu Beginn in Kapitel 3 wurden die Koordinaten des Fragment-Shaders normalisiert. Das muss auch bei der Maus berücksichtig werden um sie richtig auf dem Bildschirm bewegen zu können und besser mit Kollisionen umgehen zu können.

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

Das wird mit diesen Zeilen erreicht.

Ein Bild, das Text, Screenshot, Software, Schrift enthält.

Automatisch generierte Beschreibung

Dieser Code zeigt wie PacMan mit einem sich öffnenden und schließenden Maul dargestellt wird. Die Animation von Pac-Man in unserem Shader-Universum basiert auf der Anwendung von Trigonometrie in einem 2D-Raum. Jeder Pixel wird daraufhin überprüft, ob er innerhalb des Maulöffnungsbereichs von Pac-Man liegt. Dies wird erreicht, indem wir zuerst den Winkel(**angle**) dem Pac-Man und jedem Pixel berechnen (Verwendung von **atan** https://thebookofshaders.com/glossary/?search=atan). Der Winkel wird normalisiert, um einen reibungslosen Übergang zwischen 0 und 2\*PI (Kreiswinkel in rad) zu gewährleisten. Durch die zeitabhängige Änderung des Winkels, repräsentiert durch die Variable **u\_time** wird die Rotation des Mauls erreicht. Diese Methode ermöglicht eine lebendige Darstellung von Pac-Man, der sein Maul im Laufe der Zeit öffnet und schließt. Die **radians**-Methode(https://thebookofshaders.com/glossary/?search=radians) konvertiert einen Winkel in Grad angegeben und wird verwendet um den Öffnungsgrad des Mauls mit der Variable **maulEndWinke**l zu bestimmen.

**Kollisionserkennung**

Nachdem für jedes Objekt des Universums eine Funktion besteht, mit der die Positions des Mittelpunkt berechnet wird, kann im nächsten Schritt eine Kollisionserkennung programmiert werden.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Da es sich bei den Objekten ausschließlich um Kreise handelt, kann die Funktion **isCollision** mithilfe der Mittelpunktpositionen(**pos1, pos2**) und den Radien (**radius1, radius 2**)der zwei betrachteten Objekte eine Kollision erkennen.  
Hierfür wird der Abstand der beiden Mittelpunkte berechnet und anschließend geschaut ob dieser kleiner ist als die beiden Radien addiert. Wenn das der Fall ist überschneiden sich die Kreisränder der Objekte, was in unserem Zusammenhang eine Kollision bedeuten soll.