Cumputergraphik Abgabe

Aufgabe1: Shader

von

Gregori Daiger, Lenz Blattner

Abgabedatum 13.06.2024

Matrikelnummer, Kurs 2395489, 5611072 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 bewegen und miteinander interagieren können. Ein Highlight sind die Kollisionen und Explosionen, die für visuelle Spektakel sorgen. Hinzu kommt ein durch die Maus steuerbarer PacMan, mit dem man sich durch das Universum bewegen kann. 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 ermöglicht eine erweiterte Steuerung und Interaktivität, indem zusätzliche Variablen und ein effektiveres Zustandsmanagement integriert werden können. Der GLSL-Code des Fragment-Shaders bleibt dabei das Herzstück des Projekts, allerdings werden dem Shader Variablen von Javascript übergeben.

3. Umsetzung

Hinweis

Die Lösung besteht aus zwei Dateien:

1. fiktivesUniversum.frag
2. fiktivesUniversum.html

Die erste Datei zeigt die Implementierung des Universums im Book-of-Shaders Editor. Um das Universum anzuzeigen, einfach den Code im Book-of-Shaders-Editor einfügen. (http://editor.thebookofshaders.com/).

Die zweite Datei enthält die Erweiterung des Fragement-Shader-Codes aus der ersten Datei. Hier wurde JavaScript verwendet um weitere Effekte einzubauen, auf die in der folgenden Dokumentation eingegangen wird. Zum Anzeigen des Universums einfach das Verzeichnis der Datei in einem Browser aufrufen.

Schaffung des Universums in GLSL

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. [1]). Für das Projekt soll damit das Universum gerendert werden.

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 (vgl. Abbildung 1):

```
vec2 st = gl_FragCoord.xy / u_resolution.xy;
st.x *= u_resolution.x / u_resolution.y;
```

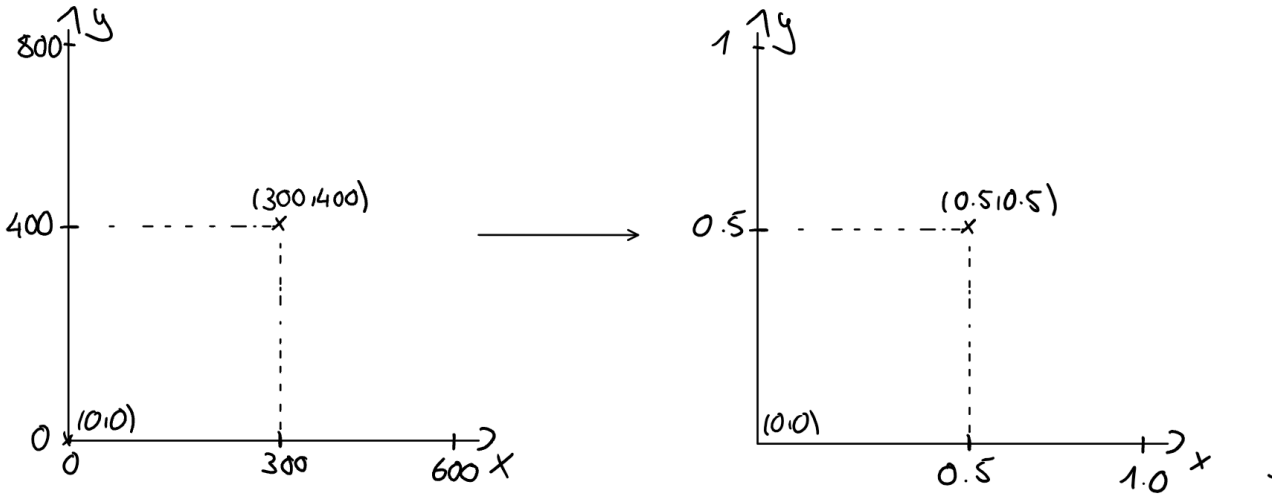
Abbildung 1: Fragment-Shader: Normalisierung, GLSL-Code

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. [2]). 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. Für den schwarzen Hintergrund wird der Wert **color** vom Datentyp vec3 initial auf schwarz gesetzt (RGB-Wert (0, 0, 0)). Im Code wird für jeden Pixel mit verschiedenen Methoden berechnet, welchen Wert die Variable haben soll, bevor sie am Ende **g1_FragColor** zugewiesen wird.

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 Pixelposition des aktuell betrachteten Pixels eine Farbe zugewiesen werden und so die Welt nur auf Basis der Farben des Fragement-Shaders gerendert werden. Initial wird jeder Pixel auf schwarz gesetzt und anschließend werden Funktionen und Bedingungen aufgerufen, die abhängig vom aktuell betrachteten Pixel eine neue Farbe für den Pixel bestimmen.

Die Transformation durch Normalisierung kann man sich so vorstellen (vgl. Abbildung 2): Durch die Normalisierung liegt jeder Pixelwert für x und y zwischen null und eins, wobei der Punkt (x=0.5, y=0.5), den Mittelpunkt darstellt.



Vorher: u_resolution (600, 800) **Nachher: st(1.0, 1.0)**

Abbildung 2: Normalisierung des Fragment-Shaders, Koordinatensystem

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 geplanten Universums um Planeten, Sonnen, Sterne, Kometen und einen 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 Funktion verwendet (vgl. Abbildung 3).

```
vec3 drawObject(vec2 st, vec2 position, float radius, vec3 color){
    if(distance(st,position) < radius){
        return color;
    }
    return vec3(0.0);
}
```

Abbildung 3: drawObjekt-Funktion, GLSL-Code

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

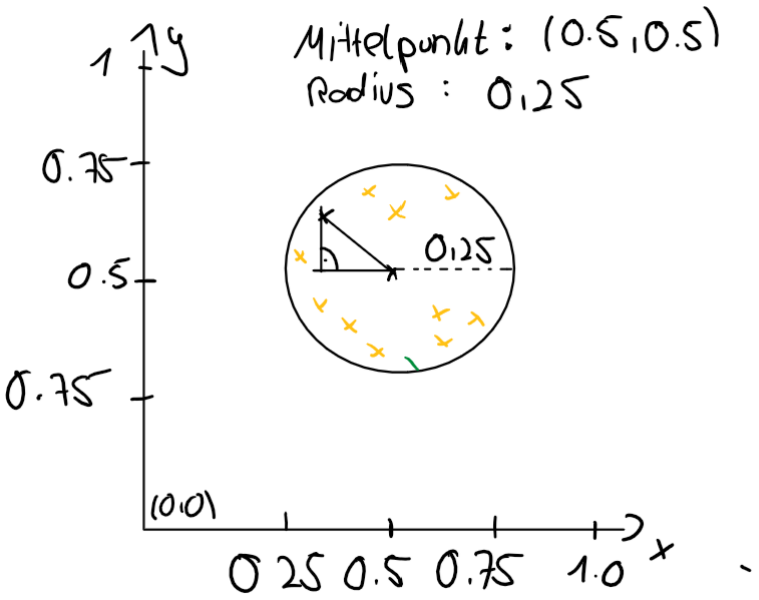


Abbildung 4: Kreis im 2D-Koordinatensystem

Die Skizze aus Abbildung 4 soll verdeutlichen wie damit ein Kreis gezeichnet werden kann. Die vom book-of-shaders eingebaute Funktion(**distance**) kann den Abstand von zwei Punkten berechnen (vgl. [3]) . Die Funktion drawObject überprüft für den einkommenden Pixel (**st**) ob er im Radius (**radius**) des Mittelspunkts (**position**) liegt. In der Abbildung stellen die gelben Kreuze Punkte da, deren Abstand vom Mittelpunkt kleiner als der Radius ist. 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 des virtuellen Kosmos zur Geltung kommen. Um den Eindruck eines Universums auf der 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. Das wird erreicht, indem für jeden Pixel zufällig entschieden wird, ob er mit einem gelblichen Schimmer erstrahlt oder nicht. Dazu wird eine **random**-Funktion definiert, die als Eingabe die Koordinate des Pixel bekommt und davon abhängig einen Wert zurückgibt (vgl. Abbildung 5).

```
float random(vec2 st){
    return fract(sin(dot(st.xy ,vec2(12.9898,78.233))) * 43758.5453);
}
```

Abbildung 5: random-Funktion: Gernerierung einer Zufallszahl

Die Random-Methode zur Erzeugung der Zufallszahl stammt aus der Dokumentaion aus Book of Shaders (vgl. [4]). 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 und wird in GLSL oft benutzt um eine Zufallszahl zu generieren.

```
float generateStars(vec2 coordinates) {
        float starValue = random(coordinates * 100.0);
        return starValue > 0.98 ? 1.0 : 0.0;
}
```

Abbildung 6: generateStars-Funktion

Die Funktion **generateStars (**vgl. Abbildung 6) ruft für den aktuell betrachteten Pixel die random-Funktion auf. Ist der generierte Wert größer als der Schwellenwert, 0.98, soll an diesem Pixel der Wert 1.0 zurückgegeben werden, sonst 0.0. Durch die Erstellung eines Pseudzufallswerts können so viele kleine Sterne an unterschiedlichen Punkten auf dem schwarzen Hintergrund dargestellt werden. Das geschieht, indem der Rückgabewert mit der gewünschten Farbe multipliziert wird, was bei 0.0 einfach den RGB-Wert (0,0,0) ergibt, ansonsten werden wird die die gewünschte Farbe (gelber Farbton) einfach mit 1 multipliziert, womit ein Punkt in diesem Fall als Stern erscheint.

```
// Einen Sternenhintergrund hinzufügen (gelbliche Punkte)
float stars = generateStars(st);
color += vec3(stars) * vec3(0.500,0.465,0.033);
```

Abbildung 7: Erzeugung der Sterne durch generateStars

Nebel und Perlin Noise

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.(vgl 5]).

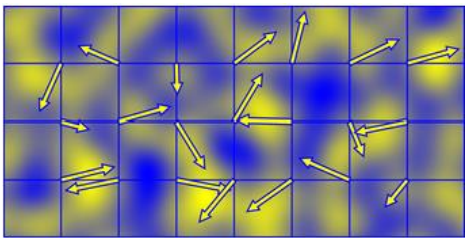


Abbildung 8: Perlin Noise Visualisierung

Abbildung 8 aus [12] zeigt eine Visualisierung von Perlin-Noise im 2D-Raum. Jedes Quadrat stellt eine Zelle des Rauschens dar, mit zufälligen Gradienten an den Ecken, dargestellt durch Pfeile. Die Farbintensität in jedem Quadrat ergibt sich aus der Interpolation dieser Gradienten, was zu sanften

Übergängen zwischen den Zellen führt. Dieses Verfahren erzeugt die Illusion von Tiefe und Flüssigkeit. Das daraus entstehende Muster simuliert den Nebel des Universums

```
float noise(vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

    // Vier Ecken im 2D ganzzahligen Raum
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));

    // Interpolieren
    vec2 u = f*f*(3.0-2.0*f);
    return mix(a, b, u.x) +
            (c - a)* u.y * (1.0 - u.x) +
            (d - b) * u.x * u.y;
}
```

Abbildung 9: noise-Funktion

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, mithilfe der **mix**-Funktion (vgl. 7) entstehen glatte Übergänge innerhalb jedes Quadrats, wodurch ein natürlich aussehendes, kontinuierliches Rauschen erzeugt wird. Die Funktion in GLSL ist in Abbildung 9 zu sehen.

```
// Einen Nebelhintergrund hinzufügen
float n = noise(st * 15.0 + u_time);
color += vec3(n) * vec3(0.1, 0.1, 0.2);
```

Abbildung 10: Erzeugung des Nebels durch noise

Der Aufruf der noise Funktion (vgl. Abbildung 10) mit **u_time** dient zur Erzeugung eines dynamischen, zeitabhängigen Nebels verwendet, während die Multiplikation der Pixelkoordinaten mit 15.0 das Rauschmuster feiner und dichter macht, was einen lebendigen Nebelhintergrund schafft.

Bewegungen der Objekte

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 ausgehend vom Zentrum innerhalb eines Radius eine Linie gezeichnet werden, das funktioniert analog zur drawObject-Funktion, nur dass hier nur am Rand die Farbe übergeben soll. Es wird also geschaut ob ein Punkt auf dem Kreis mit der Liniendicke (**lineWidth**) liegt (vgl. Abbildung 10).

```glsl
vec3 drawOrbit(vec2 Coordinate, vec2 center, float orbitRadius, float lineWidth, vec3 color) {
    if (abs(distance(Coordinate, center) - orbitRadius) < lineWidth) {
        return color;
    }
    return vec3(0.0);
}
```

Abbildung 11: drawOrbit, Zeichnung einer Kreisbahn

Bewegung auf Bahnen

Folgende Skizze (Abbildung 11) verdeutlicht wie mithilfe trigonometrischer Funktionen und dem Einheitskreisprinzip die Position eines Objekts auf einer Kreisbahn berechnet werden kann.

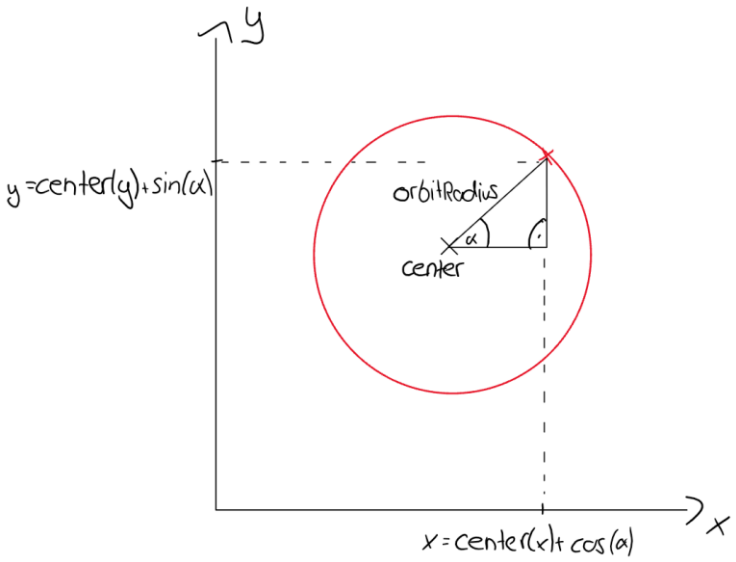


Abbildung 12: Bewegung im Kreis

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

$$x = center(x) + cos(\alpha) * orbitRadius$$

Formel 1: Berechung der x-Koordinate bei der Kreisbewegung

$$y = center(y) + \sin(\alpha) * orbitRadius$$

Formel 2: Berechnung der y-Koordinate bei der Kreisbewegung

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. Durch die Eigenschaften von Kosinus und Sinus ergibt sich damit die Kreisbahnbewegung wie man sie aus dem Einheitskreis kennt. Bei einem Winkel von 0 Grad gilt also beispielsweise befindet sich der Punkt des Kreises auf der x-Achse (cos(0) = 1, sin(0) = 0) und bei 90 Grad auf der y-Achse (cos(90) = 0, sin(90) = 1). Der **orbitRadius** entspricht der Hypotenuse und gibt den Radius des Kreises an, während der Mittelpunkt beim **center** (x=0.5, y=0.5) liegt (vgl. Formel 1 und Formel 2).

Die GLSL-Methode sieht folgendermaßen aus (Abbildung 13):

```
vec2 getCirclePosition(float u_time, float orbitRadius, vec2 center, float speed){
    float angle2 = u_time * speed;
    return center + vec2(cos(angle2), sin(angle2)) * orbitRadius;
}
```

Abbildung 13: getCirclePosition

Um die Planeten letztendlich zu zeichnen wird die getCirclePosition-Position aufgerufen und abhängig von den Parametern der von der Zeit abhängige Mittelpunkt berechnet und als vec2-Koordinate zurückgegeben. Als **center** wird der Mittelpunkt des Koordinatensystems (was dem Sonnenmitttelpunkt entspricht, übergeben. Auf diese Weise können einfach weitere Planeten, die sich auf Kreisbahnen bewegen sollen hinzugefügt werden.

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

$$x = center(x) + cos(\alpha) * majorAxis$$

$$y = center(y) + \sin(\alpha) * minorAxis$$

Umgesetzt in GLSL sieht das folgendermaßen aus (Abbildung 14):

```
vec2 getEllipsePosition(float u_time, float majorAxis, float minorAxis, vec2 center, float speed) {
  float angle = u_time * speed;
  float x = center.x + majorAxis * cos(angle);
  float y = center.y + minorAxis * sin(angle);
  return vec2(x, y);
}
```

Abbildung 14: getEllipsePosition

Mond

Wie auch in unserem Sonnensystem vorhanden, soll das Konzept von Monden eingebaut werden. Das heißt, dass um einen der Planten ein Mond kreisen soll. Er umkreist also einen sich bewegenden Planeten. Durch die Implementierung der Berechnung der Kreisbewegung als Funktion (vgl. Abbildung 13), die unteranderem abhängig vom Mittelpunkt (**center**) der Kreisbewegung ist, kann diese Feature ganz einfach hinzugefügt werden. Beim Aufruf der Methode wird für den Parameter **center**, einfach die Position des Planeten übergeben, die

vorher auch über einen Aufruf der **getCirclePosition**-Funktion berechnet wurde. Für die Position des Mondes ergibt sich damit eine Kreisbewegung um einen Mittelpunkt der von der Zeit abhängig sich bewegt. Schaut man sich in Abbildung 12 das Prinzip der Kreisbewegung an und nimmt für den Mittelpunkt, keinen statischen Punkt, sondern einen sich bewegenden Punkt, wird klar, wie sich damit ein Mond nachbilden lässt.

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 hinzugefügt 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.

```
vec3 drawSupernova(vec2 st, vec2 center, float time, float period, float duration, float maxSize, float normalSize) {
    float distanceToCenter = distance(st, center);
    float cycleTime = mod(time, period);
    bool isSupernova = cycleTime < duration;
    float progress = min(cycleTime / duration, 1.0);
    float currentSize = isSupernova ? (maxSize * progress) : normalSize;
    float glowStrength = isSupernova ? (1.0 - progress) : smoothstep(normalSize, normalSize + 0.05, distanceToCenter);

    if (distanceToCenter < currentSize) {
        vec3 color = isSupernova ? vec3(1.0, 0.5, 0.0) : vec3(1.000, 0.841, 0.092);
        return mix(color, vec3(0.0), glowStrength);
    }
    return vec3(0.0);
}
```

Abbildung 15: drawSupernova

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** (vgl. [6]) und **mix** (vgl. [7]) 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.

Das nächste Element, das 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.

```
vec2 getCometPosition(float u_time, float cometSpeedX, float cometSpeedY, float cometRadius) {
    // Modifizierte Zeitvariable für X- und Y-Bewegung
    float modifiedTimeX = mod(u_time * cometSpeedX, 2.0 - 2.0 * cometRadius);
    float modifiedTimeY = mod(u_time * cometSpeedY, 2.0 - 2.0 * cometRadius);

    // Berechnung der Kometenposition mit korrigierter Randabpralllogik
    vec2 cometPath;
    cometPath.x = modifiedTimeX > 1.0 - cometRadius ? 2.0 * (1.0 - cometRadius) - modifiedTimeX : modifiedTimeX;
    cometPath.y = modifiedTimeY > 1.0 - cometRadius ? 2.0 * (1.0 - cometRadius) - modifiedTimeY : modifiedTimeY;

    // Anpassen der Kometenposition für Randberücksichtigung
    cometPath = cometPath * (1.0 - cometRadius) + cometRadius;

    return cometPath;
}
```

Abbildung 16: getCometPoistion

Die Methode **getCometPosition** (vgl. Abbildung 16) 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:

$$x, y < 1 - cometRadius$$

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:

$$x = 2.0 * (1.0 - cometRadius) - modifiedTimeX$$

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.

Schwarze Löcher

```
// Funktion zur Erstellung eines schwarzen Lochs
vec3 applyBlackHoleEffect(vec2 st, vec2 blackHolePosition, float blackHoleRadius, vec3 color) {
    float distanceToBlackHole = distance(st, blackHolePosition);
    float effectRadius = blackHoleRadius * 2.0; // Effektradius doppelt so groß wie der Radius des Schwarzen Lochs

    if (distanceToBlackHole < effectRadius) {
        float effectStrength = smoothstep(effectRadius, blackHoleRadius, distanceToBlackHole);
        color = mix(color, vec3(0.0), effectStrength);
    }

    return color;
}
```

Abbildung 17: Funktion zur Erzeugung eines schwarzen Lochs

Die Funktion **applyBlackHoleEffect** (vgl. Abbildung 17) simuliert das visuelle Erscheinungsbild eines Schwarzen Lochs im Universum. Die Funktion erhält die aktuelle Position (**st**) eines Pixels, die Position des Schwarzen Lochs (**blackHolePosition**), den Radius des Schwarzen Lochs (**blackHoleRadius**) und die aktuelle Farbe des Pixels (**color**).

Die grundlegende Idee besteht darin, die Pixel in der Nähe des Schwarzen Lochs abzudunkeln, um den Effekt eines starken Gravitationsfelds zu simulieren, das das Licht absorbiert. Der **effectRadius**, der doppelt so groß wie der **blackHoleRadius** ist, definiert den Einflussbereich des Schwarzen Lochs. Die Variable **distanceToBlackHole** enthält den Abstand zwischen dem betrachteten Pixel und dem Zentrum des Schwarzen Lochs.

Wenn sich ein Pixel innerhalb des **effectRadius** befindet, wird die Funktion smoothstep verwendet, um einen weichen Übergang der Farbintensität zu erzeugen. Die mix-Funktion mischt daraufhin die ursprüngliche Farbe des Pixels mit Schwarz (vec3(0.0)) entsprechend der Stärke des Effekts (**effectStrength**). Dadurch entsteht ein visueller Effekt, bei dem die Pixel in der Nähe des Schwarzen Lochs allmählich dunkler werden, je näher sie dem Zentrum des Schwarzen Lochs kommen, wodurch der Eindruck eines absorbierenden, dunklen Bereichs im Universum entsteht.

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. [8]). Da der Bildschirm zu Beginn in Kapitel 3 wurden die Koordinaten des Fragment-Shaders normalisiert. Das muss auch bei der Maus berücksichtigt werden um sie richtig auf dem Bildschirm bewegen zu können und besser mit Kollisionen umgehen zu können (vgl. Abbildung 18).

```
vec2 pacmanPosition = u_mouse / u_resolution;
pacmanPosition.x *= u_resolution.x / u_resolution.y;
```

Abbildung 18: u_mouse Normierung

```
// Pacman mouth parameters
float mouthAngle = radians(45.0); // Mouth opening angle in degrees
float mouthDirection = radians(u_time * 50.0); // Mouth rotates over time

float distPAC = distance(st, pacmanPosition);
if (distPAC < pacmanRadius) {
    float angle = atan(st.y - pacmanPosition.y, st.x - pacmanPosition.x);

    if (angle < 0.0) angle += 6.28318530718;

    float mouthRotationSpeed = 1.0;
    float mouthStartAngle = mod(u_time * mouthRotationSpeed, 6.28318530718);
    float mouthEndAngle = mouthStartAngle + radians(90.0);

    if (!(angle > mouthStartAngle && angle < mouthEndAngle)) {
        color = vec3(1.0, 1.0, 0.0);
    }
}

if (distance(pacmanPosition, PlanetPosition2) < pacmanRadius + planetRadius2){
    if(distance(st, PlanetPosition2) < planetRadius2){
        color = vec3(1.0, 0.0, 0.0);
    }
}
```

Abbildung 19: Pacman zeichnen

Der Code in Abbildung 19 zeigt wie PacMan mit einem sich öffnenden und schließenden Maul dargestellt wird. Jeder Pixel wird daraufhin überprüft, ob er innerhalb des Maulöffnungsbereichs von Pac-Man liegt. Dies wird erreicht, indem man zuerst den Winkel(**angle**) zwischen dem Pac-Man-Mittelpunkt (**pacmanPosition**) und jedem Pixel mithilfe der **atan**-Funktion berechnet (vgl. [9]). 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(vgl. [10]) 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 Position des Mittelpunkt berechnet wird, kann im nächsten Schritt eine Kollisionserkennung programmiert werden.

```
bool isCollision(vec2 pos1, vec2 pos2, float radius1, float radius2){
    if (distance(pos1, pos2) <= radius1 + radius2){
        return true;
    }
    return false;
}
```

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.

4. Erweiterung mit ThreeJS

Durch die Auslagerung der Positionsbestimmung in JavaScript konnte die gleiche Logik wie im Shader verwendet, aber flexibler gestaltet werden. Die berechneten Positionen der Himmelskörper wurden nun als **Uniforms** in den Shader übertragen, was eine reibungslose und effiziente Anpassung und Erweiterung des Universums ermöglicht. Diese Herangehensweise nutzt die Vorteile von JavaScript, um ein dynamisches, interaktives Erlebnis zu schaffen, das über die Möglichkeiten des reinen Shader-Codes hinausgeht, aber letztendlich doch auf GLSL-Code als Basis basiert. Für den Übergang zu WebGL kann das Framework Three.js verwendet werden (vgl. [11]).
Der bereits bestehende Code musste nur an wenigen Stellen angepasst werden um beispielsweise die Normalisierung auch im Browser auf die 2D-Koordinaten in einem quadratischen Bildrahmen zu erzeugen.

```
uniforms = {
  u_time: { type: "f", value: 1.0 },
  u_resolution: { type: "v2", value: new THREE.Vector2() },
  u_mouse: { type: "v2", value: new THREE.Vector2() },
  u_planetPosition2: { type: "v2", value: new THREE.Vector2() },
  u_planetPosition1: { type: "v2", value: new THREE.Vector2() },
  u_planetPosition3: { type: "v2", value: new THREE.Vector2() },
  u_kometPosition: { type: "v2", value: new THREE.Vector2() },
  u_pacmanPosition: { type: "v2", value: new THREE.Vector2() },
```

Abbildung 20: uniforms

Innerhalb der **init()**-Methode können die Uniforms übergeben werden (vgl Abbildung 20), auf die dann im eingebetteten Fragment-Shader Teil zugegriffen werden kann. Die Abbildung zeigt, wie die Uniforms definiert werden. Beispielsweise bietet Three.js die 2D-Vekoren (vec2) als Nachbildung in Form der Klasse **THREE.Vector2** an. Zu erkennen ist, dass die Positionsberechnung jetzt in Javascript-stattfindet und dann an den Fragment-Shader übergeben werden soll.

Auf den JavaScript-Code soll im Folgenden nicht so detailreich wie zuvor eingegangen werden. Es soll beispielhaft für die Positionsbestimmung des Planeten 2 gezeigt werden, wie diese aus dem Fragment-Shader in Javascript ausgelagert werden können. Das Gleiche wird analog für alle anderen Positionsbestimmungen durchgeführt.

```
function getCirclePosition(time, orbitRadius, center, speed, positionUniform) {
  var angle = time * speed;
  var x = center.x + Math.cos(angle) * orbitRadius;
  var y = center.y + Math.sin(angle) * orbitRadius;

  positionUniform.value = new THREE.Vector2(x, y);
}
```

Abbildung 21: getCirclePosition: Bestimmung der Poistion auf Kreisbahn mit Javascript

Die Javascript-Methode **getCirclePosition()** (vgl. Abbidlung 21) berechnet genau wie im Book of Shaders – Editor, die Position eines Planeten auf der Kreisbahn, indem er die dafür nötigen Parameter, für die vergangene Zeit (**time**), den Radius des Orbits (**orbitRadius**), des Zentrums der Sonne (**center**) und die Geschwindigkeit (**speed**) entgegennimmt und dann mit JavaScript-Code die nötigen Berechnungen durchführt. Wie im Abschnitt zur Positionsberechnung (vgl. Formel 1 und Formel 2) werden die x- und y-Koordinate auf der Kreisbahn bestimmt. wird hier als weiteres Argument noch das Objekt aus dem Uniform-Dictionary (**positionUniform**) übergeben, damit die Änderungen dann auch an den Fragment-Shader übergeben werden, wo sie letztendlich gerendert werden.

Erzeugung von Explosionen

Die Idee hinter einer Explosion, ist das temporäre animieren einer Explosion ähnlich wie bei der Supernova, nachdem eine Kollision stattgefunden hat. Um das zu erreichen, kann die Methode für die Kollisionserkennung in Javascript (vgl. Abbildung 22) nachgebaut werden.

```
// Calculate Distance between to Points
function calculateDistance(vec2a, vec2b) {
  var dx = vec2a.x - vec2b.x;
  var dy = vec2a.y - vec2b.y;
  return Math.sqrt(dx * dx + dy * dy);
}

// Check if two object of the universe collide
function checkCollision(center1, center2, radius1, radius2) {
  return calculateDistance(center1, center2) < radius1 + radius2;
}
```

Abbildung 22: calculateDistance and checkCollision in Javascript

Bei jedem Durchlauf der render-Methode wird überprüft ob einer der Planeten mit dem Kometen kollidiert. Falls ja soll eine Uniform (**isExplosion**) für den entsprechenden Planeten gesetzt werden. Mithilfe einer weiteren Variable kann die Dauer der Explosion eingestellt werden.

```
// Handle-Funktino bei Kollison und Auslösung der Explosion
function handleCollisionAndExplosion(
  planetIndex,
  isCollision,
  explosionStartTime
) {
  // Beginn der Explosion
  if (isCollision && !uniforms[`u_isExplosion${planetIndex}`].value) {
    uniforms[`u_isExplosion${planetIndex}`].value = 1;
    // Startzeitpunkt wird in explosionStartTime gestgehalten
    explosionStartTime = Date.now();
  }

  if (explosionStartTime) {
    // Explosion hält an bis seit Explosionsbeginn 3sek (explosionDuration) vergangen ist
    if (Date.now() - explosionStartTime > explosionDuration) {
      uniforms[`u_isExplosion${planetIndex}`].value = 0;
      explosionStartTime = null;
    }
  }
  return explosionStartTime;
}
```

Abbildung 23: handleCollisionAndExplosion

Die Funktion **handleCollisionAndExplosion** (vgl. Abbildung 23) setzt, sobald eine Kollision erkannt wird die Uniform **isExplosion** für den entsprechenden Planeten, unzwar so lange, bis die Explosionszeit abgelaufen ist.

```
vec3 drawExplosion(vec2 st, vec2 explosionCenter, bool isExplosionActive, float explosionRadius) {
   if (!isExplosionActive) return vec3(0.0); // No Explosion

   float distToExplosion = distance(st, explosionCenter);
   float glowStrength = smoothstep(explosionRadius, explosionRadius + 0.05, distToExplosion);

   return mix(vec3(1.0, 0.5, 0.0), vec3(0.0), glowStrength);
}
```

Abbildung 24: drawExplosion

Die Funktion **drawEploxsion** (vgl. Abbildung 24) stellt die Explosion mithilfe der Uniform (**isExplosion**), die als Parameter übergeben wird, dar. Wenn dieser Boolean nicht erfüllt ist, soll einfach nichts geschehen, andernfalls soll analog wie bei der Supernova eine Explosion erzeugt werden, indem die **mix** und **smoothstep** Funktionen verwendet werden.

Aufruf der Funktionen beim Rendern

Nachdem alle Funktionen, die für die Implementierung von Zustandsabhängigen Animationen nötig sind, ausgelagert wurden, gilt es diese an der passenden Stelle aufzurufen. In ThreeJS spiel hierfür die **render**()-Funktion die entscheidende Rolle. Diese Funktion wird nämlich für jeden Pixel aufgerufen und ist verantwortlich für das Rendern der Szene und aktualisiert unter Anderem die Positionen, die vom Fragment-Shader als uniforms verarbeitet werden. Um die gewünschten Effekte zu erreichen werden die Funktionen zur Positionsberechnung hier aufgerufen.

```
function render() {
  uniforms.u_time.value += clock.getDelta();
  var elapsedTime = clock.getElapsedTime();

  // Berechnung der Positionen für alle Objekte im Univserum
  // Planet 1
  getCirclePosition(
    elapsedTime,
    orbitRadius1,
    new THREE.Vector2(0.5, 0.5),
    0.6,
    uniforms.u_planetPosition1
  );
```

Abbildung 25: Aufruf der Funktionen zur Positionsberechnung innerhalb der render-Funktion

Abbildung 25 zeigt wie das realisiert wird. Innerhalb der render-Funktion werden die Zeitvariablen neu beschrieben durch die von Javascript angebotene clock-Klasse. Anschließend können mit den aktualisierten Zeiten die Positionsberechnungen stattfinden. In der Abbildung wird beispielsweise die Position für den ersten Planeten überschrieben, indem getCirclePosition (vgl. Abbildung 21) aufgerufen wird.

4. Fazit

Auf weitere Details des JavaScript-Codes wird nicht eingegangen, da das Herzstück der Fragment-Shader ist. Die JavaScript-Auslagerung sollte nur ein wenig mehr Flexibilität einbauen, was erreicht wurde. Die gezeigten Implementierungen im Abschnitt zu ThreeJS, geben einen ausreichenden Überblick um zu verstehen, wie dieser Schritt erfolgt ist und wie damit zeitlich begrenzte Explosionen erzeugt werden können (Kollisionserkennung und Zustandsspeicherung). Um ein wenig mit dem Universum herumzuspielen, können einfach die Variablen für die Parameter der unterschiedlichen Objekte angepasst werden. Das kann im oberen Teil des Fragment-Shaders gemacht werden. Die Implementierung der Logik in Form von Funktionen ermöglicht ein einfaches Hinzufügen weiterer Planeten und Umlaufbahnen.

Quellenverzeichnis

[1] The Book of shaders, *fragment-shader*, Verfügbar unter https://shadertutorial.dev/basics/fragment-shader/).

[2] The Book of shaders, *gl_FragCoord*, https://thebookofshaders.com/03/

[3] The Book of shaders, *Distance*, https://thebookofshaders.com/glossary/?search=distance

[4] The Book of shaders, *Random*, https://thebookofshaders.com/10/

[5] The Book of shaders, *Noise*, https://thebookofshaders.com/11/

[6] The Book of shaders, *Smoothstep,* https://thebookofshaders.com/glossary/?search=smoothstep

[7] The Book of shaders, *mix*, https://thebookofshaders.com/glossary/?search=mix

[8] The Book of shaders, *u_mouse*, https://thebookofshaders.com/03/).

[9] The Book of shaders, atan, https://thebookofshaders.com/glossary/?search=atan

[10] The Book of shaders, *radians*, https://thebookofshaders.com/glossary/?search=radians

[11] The Book of shaders, In Three.js, *https://thebookofshaders.com/04/*

[12] StackExchange, https://gamedev.stackexchange.com/questions/23625/how-do-you-generate-tileable-perlin-noise