Shader-Programmierung 1: Grundlagen und Sprites

Description

Shader sind aus modernen Spielen nicht mehr wegzudenken. Selbst in 2D-Spielen werden sie zahlreich eingesetzt. In dieser kleinen Serie sollen die Grundlagen und Möglichkeiten vermittelt werden. Implementiert wird es in *GameMaker Studio 2*. Da es sich um Grundlagen handelt, ist es dennoch für die meisten Anwendungen und Entwicklungsumgebungen geeignet.

Zielgruppe des Tutorials

Die Shaderprogrammierung ist nichts für Leute, die sonst noch nie etwas im GameMaker getan haben. Wer dem Tutorial folgen will, sollte schon einige Stunden in der Game-Engine verbracht haben und wissen, wie man Ressourcen und Instanzen erstellt. Außerdem ist es von Vorteil, wenn man bereits einige Begrifflichkeiten aus der Programmierung und der Spieleentwicklung kennt, auch wenn ich mich bemühe, die Dinge näher zu erläutern, die man für das Grundverständnis braucht.

Das ist unser Ausgangspunkt. Wir werden zwei der drei Sprites verändern

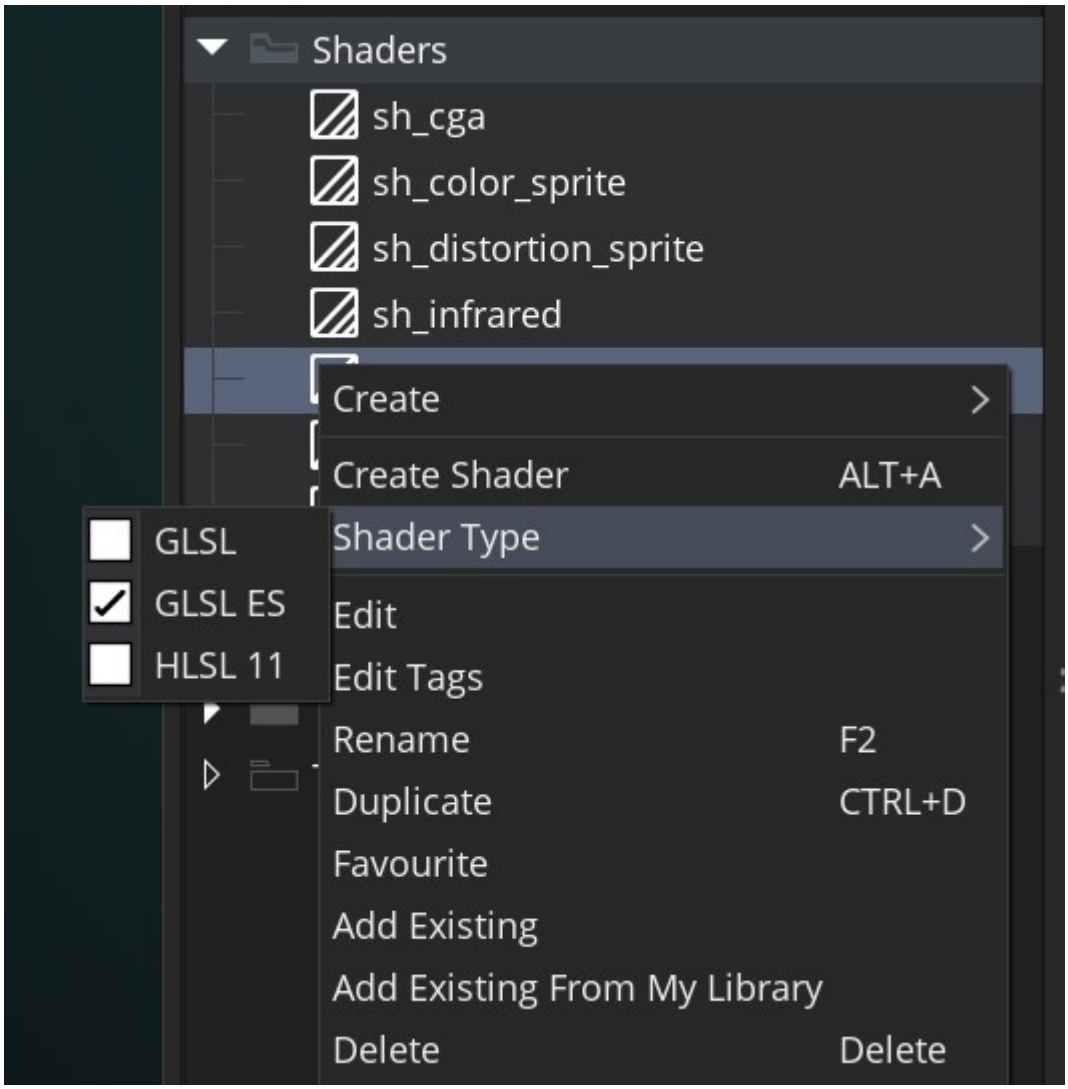
Diese Serie befasst sich mit den Grundlagen und beinhaltet einige praktische Beispiele. Es geht um die Technik, wie man Shader programmiert und sie in GameMaker Studio 2 einbaut. Das bedeutet, dass hier nicht jeder Sonderfall behandelt werden kann. Ich hoffe jedoch, dass die Serie die gängigsten Fälle abdeckt und das nötige Verständnis vermittelt, damit sich bei den kommenden Herausforderungen jeder selbst helfen kann.

Was nicht vermittelt wird, ist die Mathematik hinter den einzelnen Shadern.

Alle verwendeten Grafiken stammen übrigens von Open Gaming Art.

Was sind Shader?

Shader sind spezielle Programme, die in Echtzeit Grafiken generieren und steuern. Ursprünglich wurden sie entwickelt, um Schattierungen für die Beleuchtung zu erzeugen – daher der Name. Inzwischen werden sie für eine Vielzahl von Effekten verwendet. Sie werden normalerweise in einer speziellen Programmiersprache geschrieben, wie z. B. GLSL (Open**GL** **S**hading **L**anguage) oder HLSL (**H**igh-**L**evel **S**hading **L**anguage, die bei der Arbeit mit DirectX verwendet wird). Diese Sprachen ermöglichen es dem Entwickler, detaillierte Anweisungen zu geben, wie ein bestimmtes Grafikelement gerendert werden soll.



Mit einem Rechtsklick auf den Shader kann man in GameMaker Studio 2 den Shader-Typ wählen

Im GameMaker können Shader in GLSL, HLSL und GLSL ES (eine Untergruppe von GLSL, die bei mobilen Geräten üblich ist) geschrieben werden. ES steht übrigens für **E**mbedded **S**ystems. Abgesehen von Syntaxunterschieden sollten die Mathematik und die Techniken in allen drei Sprachen ähnlich sein.

Wie funktioniert ein Shader?

Shader werden direkt auf dem Grafikprozessor ausgeführt und nicht auf der CPU (**C**entral **P**rocessing **U**nit). Sie sind in der Lage, Grafiken in Echtzeit zu erzeugen und zu manipulieren, indem sie die Eigenschaften von Objekten und Oberflächen in einer Szene steuern. Die Grafikkarten sind für solche Berechnungen optimiert, weshalb Shader wesentlich leistungsfähiger laufen, als wenn man diese Berechnungen über den Prozessor ausführen würde.

Ein Shader besteht aus einer Reihe von Anweisungen und Variablen, die es ihm ermöglichen, bestimmte Effekte zu erzeugen. Der Prozess beginnt damit, dass die CPU Daten an die GPU (**G** raphics **P**rocessing **U**nit) sendet. Diese Daten enthalten bspw. Informationen über die Position, Größe,

Farbe und andere Eigenschaften von Objekten in der Szene.

https://www.bytegame.de/wp-content/uploads/2023/05/Plasma-Gruen-Lila.mp4

Ebenfalls ein Shader-Effekt: Dieses schöne Plasma

Der Shader wird dann aufgerufen und beginnt, die übergebenen Daten zu verarbeiten. Es führt mathematische Operationen auf den Daten aus, um sie zu manipulieren und Effekte wie Beleuchtung, Schatten oder Verzerrungen zu erzeugen. Er kann auch Texturen verwenden, um realistische Oberflächen zu simulieren und visuelle Effekte wie Glanz, Spiegelungen oder Transparenz zu erzeugen.

Sobald der Shader seine Berechnungen abgeschlossen hat, sendet er die resultierenden Grafiken zurück an die GPU, wo sie auf dem Bildschirm gerendert werden.

Wo werden Shader eingesetzt?

In GameMaker Studio 2 und anderen Entwicklungsumgebungen können Entwickler Shader nutzen, um die Grafiken in einem Spiel zu verbessern und anspruchsvolle visuelle Effekte erzeugen. Durch die Manipulation von Farben, Texturen und anderen Eigenschaften können Shader eine Vielzahl von Effekten darstellen, die dem Spiel eine visuelle Tiefe und Atmosphäre verleihen. Normalerweise setzt man Shader für folgende Aufgaben ein:

- Post-Processing-Effekte wie Farbkorrektur, Sättigung, Verzerrungen, Weichzeichner, Pixeleffekt, CRT-Effekt etc.
- Manipulation von Sprites (Farbe, Verzerrung etc.)
- Erzeugen von bildschirmfüllenden 2D-Effekten (Plasma, Fraktale u. v. m.)
- Verbesserung von 3D-Szenen (Licht/Schatten, Wasser, Spiegelungen, Texturen etc.)
- Generieren von ganzen 3D-Szenen
- Partikeleffekte wie Rauch, Feuer, Wasser, Blut und Explosionen
- Simulationen von Flüssigkeiten, Wolken, Nebel, Gras, Haare usw.

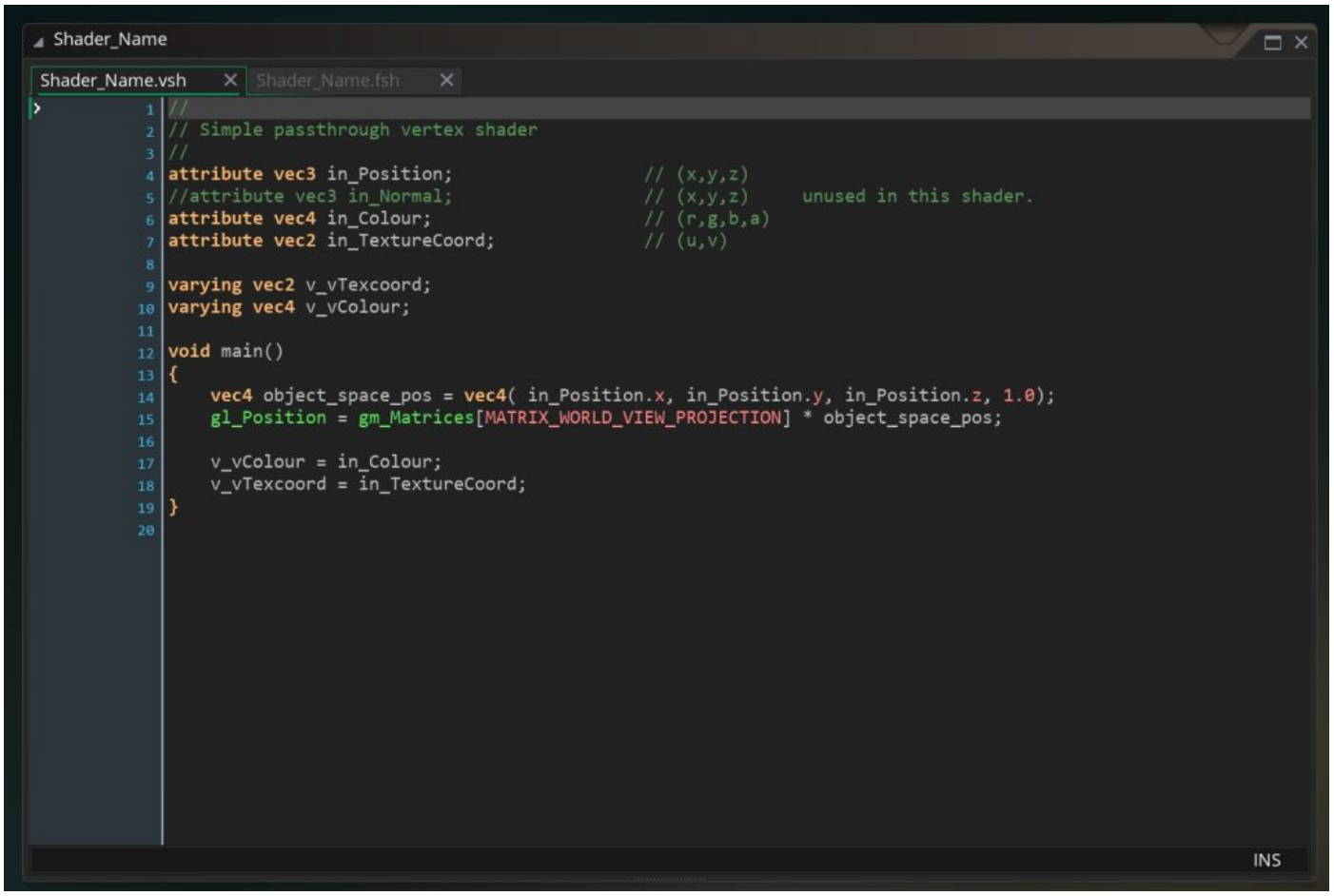
Die Vielzahl der Möglichkeiten macht den Einstieg etwas schwierig. Da man mit Shadern so viel machen kann, erscheint das Thema nicht wirklich greifbar. Ich hoffe, dass dieses Tutorial für mehr Klarheit sorgt.

Einsatz im GameMaker Studio 2

In GMS gibt es zwei Shader. Den „Vertex-Shader“ und den „Fragment-Shader“.

Vertex-Shader

Der Vertex-Shader ist ein Programm, das auf jeder Eingabe-Vertex des zu rendernden Modells ausgeführt wird. Seine Hauptaufgabe besteht darin, die 3D-Position jedes *Vertex* in 2D-Bildschirmkoordinaten zu transformieren und andere Daten wie Farben oder Texturen für die Verwendung im Fragment-Shader zu interpolieren.



Wenn man in GMS einen neuen Shader erstellt, wird der Basiscode für Vertex- und Fragment-Shader automatisch erstellt

In der Computergrafik ist ein Vertex (Mehrzahl: Vertices oder Vertexes) ein Eckpunkt in einem geometrischen Modell, das aus einer Sammlung von Punkten, Linien und Flächen besteht. Jeder Vertex hat eine Position im Raum und kann auch zusätzliche Informationen wie Farbe, Texturkoordinaten oder Normalenvektoren enthalten.

Fragment-Shader

Der Fragment-Shader ist ein Programm, das auf jedem Pixel der gerenderten Oberfläche ausgeführt wird. Seine Hauptaufgabe besteht darin, die Farben und andere visuelle Effekte für jeden Pixel auf der Oberfläche zu berechnen, basierend auf den Daten, die vom Vertex-Shader interpoliert wurden. Dieser wird in 2D-Spielen am häufigsten verwendet.

Die Berechnungen im Fragment-Shader werden für jeden Pixel auf der gerenderten Oberfläche ausgeführt. Die Ergebnisse der Berechnungen werden dann verwendet, um den Pixel im Framebuffer zu setzen, der dann auf dem Bildschirm angezeigt wird.

Insgesamt arbeiten Vertex- und Fragment-Shader eng zusammen, um die Geometrie und Farben der gerenderten Objekte in einer 3D-Szene zu erzeugen. Vertex-Shader führen Berechnungen für jedes Vertex aus, während der Fragment-Shader für jeden Pixel ausgeführt wird.

Float-Nummern

In der Programmierung gibt es verschiedene Datentypen, die zur Speicherung von Zahlenwerten verwendet werden können. Einer der häufigsten Datentypen ist der sogenannte „float“, der zur Speicherung von Fließkommazahlen verwendet wird. Float-Zahlen haben normalerweise eine bestimmte Anzahl an Nachkommastellen, die je nach Anwendung variieren können.

In GLSL wird die Verwendung von Float-Zahlen zur Darstellung von Farben und anderen Grafikdaten häufig verwendet. Um sicherzustellen, dass eine Zahl als Float-Nummer und nicht als Ganzzahl interpretiert wird, muss man das Suffix „.0“ oder „.0f“ hinzufügen. Zum Beispiel würde die Zahl 1 als Integer geschrieben, während 1.0 oder 1.0f als Float geschrieben würden.

Im konkreten Beispiel werden die Float-Werte 1.0, 0.0 und 0.0 als Rotanteil, Grünanteil und Blauanteil einer Farbe verwendet, während der Float-Wert 1.0 als Alpha-Kanal verwendet wird, um die Deckkraft des Objekts zu bestimmen.

Wichtig: Schreibt man versehentlich „1“ statt „1.0“, gibt es bei der Kompilierung eine Fehlermeldung, weil float erwartet wird, aber integer kommt.

https://www.bytegame.de/wp-content/uploads/2023/05/Tunneleffekt.mp4

Der kleine Tunneleffekt wurde ebenfalls als Shader (in GameMaker Studio 2.3) programmiert

Ablauf

Im ersten Tutorial werden wir lediglich einen Sprite manipulieren. Dafür gehen wir wie folgt vor:

1. Neues Projekt erstellen
2. Sprite laden
3. Objekt erstellen
4. Sprite im Objekt setzen
5. Shader erstellen
6. Fragment-Shader programmieren
7. Shader im Draw-Event des Objekts einbinden

Shader erstellen

Da die ersten vier Punkte klar sein sollten, springen wir gleich zu Punkt 5. In der Leiste ganz rechts (Assets) können wir einen Shader anlegen. Im Beispiel nenne ich ihn *sh_color_sprite*. Wenn wir es doppelklicken, sehen wir, dass Fragment- und Vertex-Shader bereits Code enthalten. Im Beispiel werden wir zunächst die Farbe des Sprites manipulieren, also brauchen wir lediglich den Fragment-Shader. Der Vertrex-Shader bleibt, wie er ist.

Farbe ändern

GMS gibt uns beim Fragment-Shader folgenden Code:

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

void main()
{
    gl_FragColor = v_vColour * texture2D( gm_BaseTexture, v_vTexcoord );
}
```

Das ist unser Grundgerüst. Wir fügen lediglich zwei Zeilen hinzu und verändern die letzte Zeile:

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;
uniform vec4 sprite_color;

void main()
{
    vec4 inverted_color = sprite_color;
    gl_FragColor = texture2D(gm_BaseTexture, v_vTexcoord) * inverted_color * v
}
```

Die erste Änderung ist `uniform vec4 sprite_color;`. Mit *uniform* deklarieren wir eine Variable, auf die wir „von Außen“, also dem Objekt mit dem Sprite, zugreifen können. Hier haben wir eine Variable namens `sprite_color` definiert, die eine 4-Komponenten-Vektor ist, der die RGBA-Werte (**R**ot, **G**rün, **B**lau, **A**lpha, also Transparenz) der Farbe des Sprites enthält.

Die zweite Veränderung ist die Zeile `vec4 inverted_color = sprite_color;`. Hier wird der Farbwert übergeben. Damit dieser auch berechnet wird, haben wir die letzte Zeile lediglich ergänzt.

`gl_FragColor` ist eine Ausgabevariable in einem Fragment-Shader-Programm, die verwendet wird, um die Farbe des aktuellen Fragments (Pixel) zu setzen.

In OpenGL-basierten Grafiksystemen, einschließlich GameMaker Studio 2, wird die `gl_FragColor` Variable automatisch vom Grafiktreiber erstellt und initialisiert. Der Fragment-Shader-Code muss lediglich eine Farbe zuweisen, um das entsprechende Fragment zu färben.

Draw-Event im Objekt

Hier brauchen wir nur wenige Zeilen:

```
shader_set(sh_color_sprite);
sh_color = shader_get_uniform(sh_color_sprite, „sprite_color");
shader_set_uniform_f(sh_color, 1.0, 0.0, 0.0, 1.0); // Färbt das Sprite rot
draw_self();
shader_reset();
```

Mit `shader_set()` aktivieren wir den Shader, der auf das aktuelle Objekt angewendet werden soll. `sh_color_sprite` enthält die ID des zuvor erstellten Shaders. Durch das Aktivieren des Shaders wird der Grafikprozessor des Computers angewiesen, diesen Shader für alle folgenden

Grafikoperationen zu verwenden, bis ein anderer Shader aktiviert wird.

Die zweite Zeile ruft die Funktion `shader_get_uniform()` auf, um die Uniform-Variable `sprite_color` aus dem Shader zu holen. In diesem Fall wird die Uniform-Variable aus dem Shader mit der ID `sh_color_sprite` abgerufen und der Variablen `sh_color` zugewiesen.

Die dritte Zeile ruft die Funktion `shader_set_uniform_f()` auf, um den Wert der Uniform-Variable `sh_color` auf `vec4(1.0, 0.0, 0.0, 1.0)` zu setzen. Diese Funktion setzt den Wert der Uniform-Variable, die mit der ID in der ersten Argumentposition übereinstimmt, auf einen Vektor mit 4 float-Werten, die in den nachfolgenden Argumenten angegeben werden. In diesem Fall wird die Uniform-Variable `sh_color`, die in Schritt 2 definiert wurde, auf einen Vektor gesetzt, der Rot als Primärfarbe und eine vollständige Deckkraft (1.0) hat.

Würde man es grün färben wollen, sähe die Zeile so aus:

```
shader_set_uniform_f(sh_color, 0.0, 1.0, 1.0, 1.0);
```

Die nächste Zeile ruft die Funktion `draw_self()` auf, um das aktuelle Objekt zu zeichnen. Der Grafikprozessor des Computers verwendet den aktiven Shader, um das Objekt zu rendern, und verwendet die Uniform-Variable `sprite_color`, um die Farbe des Objekts zu bestimmen.

Links das Original, rechts ist es rot eingefärbt

Die letzte Zeile ruft die Funktion `shader_reset()` auf, um den aktuell aktiven Shader aufzuheben und den Standard-Shader des Systems wiederherzustellen. Dies stellt sicher, dass nach Beendigung des Renderns des Objekts der aktive Shader nicht versehentlich auf andere Objekte oder Grafikoperationen angewendet wird.

Sprite verzerren

Im zweiten Beispiel werden wir das Sprite verzerren. Im Prinzip ist das nicht viel anders, als würde man die Farbe ändern. Der Code für den Fragment-Shader sieht wie folgt aus:

```
varying vec2 v_vTexcoord;
```

```
varying vec4 v_vColour;
uniform vec2 distortion_amount;

void main()
{
    vec2 distortion_offset = vec2(
        sin(v_vTexcoord.y * 10.0) * distortion_amount.x,
        cos(v_vTexcoord.x * 10.0) * distortion_amount.y
    );

    vec2 distorted_texcoord = v_vTexcoord + distortion_offset;

    gl_FragColor = v_vColour * texture2D(gm_BaseTexture, distorted_texcoord);
}
```

Die Verzerrung erfolgt, indem wir jeden Pixel des Sprites in seiner Position verändern. Das Beispiel verwendet eine Uniform-Variable, die `distortion_amount` heißt und einen vec2-Vektor darstellt. Die x-Komponente des Vektors beeinflusst die Verzerrung auf der horizontalen Achse, während die y-Komponente die Verzerrung auf der vertikalen Achse beeinflusst.

Im Fragment-Shader-Bereich wird ein Offset-Vektor `distortion_offset` berechnet, der abhängig von der aktuellen Texturkoordinate (in `v_vTexcoord`) und den Verzerrungsparametern ist. In diesem Beispiel wird der Offset basierend auf der Sinus- und Kosinus-Funktion der Texturkoordinaten berechnet. Man kann jedoch die Formel ändern, um verschiedene Arten von Verzerrungen zu erzeugen.

Der berechnete Offset wird dann auf die Texturkoordinate addiert, um die verzerrte Texturkoordinate `distorted_texcoord` zu erhalten. Schließlich wird die Farbe des verzerrten Pixels aus der Textur gelesen und mit der Vertex-Farbe (in `v_vColour`) multipliziert, um die endgültige Ausgabe-Farbe (`gl_FragColor`) zu generieren.

Draw-Event

Hier ist der Code nahezu identisch zur Färbung:

```
shader_set(sh_distortion_sprite);
sh_distortion = shader_get_uniform(sh_distortion_sprite, „distortion_amount");
shader_set_uniform_f(sh_distortion, 0.02, 0.04);
draw_self();
shader_reset();
```

Wir setzen nur andere Namen ein (der Shader heißt bei mir `sh_distortion_sprite`) und übergeben lediglich zwei Variablen statt der vier bei den Farben. Da ich im Beispiel ein kleines Sprite verwende, reichen geringe Werte, wie 0.02 und 0.04, um eine deutliche Änderung zu sehen.

Dynamische Werte

Die Verzerrung, aber auch die Verfärbung, lassen sich natürlich auch variabel gestalten. So kann man bspw. die Farbe von Bedingungen im Spiel abhängig machen, etwa den Abstand zu bestimmten Objekten.

Verzerrungseffekt

Bei der Verzerrung wäre eine Animation sehr schön. So könnte man zwei Variablen mit `disX` und `disY` anlegen und diese bis zu einem bestimmten Wert hochzählen lassen.

Beispiel:

Create-Event

```
disX = 0.0;
disY = 0.0;
```

Draw-Event

```
if (disX < 0.8)
{
 disX += 0.01;
 disY += 0.02;
}

shader_set_uniform_f(sh_distortion, disX, disY);
```

Einfach die entsprechenden Zeilen einfügen bzw. die letzte Zeile austauschen, schon erhalten wir eine hübsche Bewegung.

Fazit und Ausblick

Was wir bisher gemacht haben, war nur die oberste Schneeflocke auf dem Shader-Eisberg. Das Thema ist so umfangreich, dass sich damit zahllose Bücher befassen.

Dieser Auftakt hilft, den Einstieg in das Thema zu finden und offenbart dabei zahlreiche Möglichkeiten. Wir können nun Farben und Pixel einzelner Sprites manipulieren, womit man sehr viele Dinge anfangen kann. Eines der Vorteile ist, dass wir nicht für jede Kleinigkeit neue Sprites brauchen. Wir färben und verformen es über den Shader und können so zum Beispiel auch Objekte explodieren lassen.

Im nächsten Teil gehen wir einen Schritt weiter und manipulieren alles, was auf dem Bildschirm zu sehen ist. Solche Post-Processing-Effekte können in Spielen sehr oft eingesetzt werden.

Verwendete Sprites

Magier
Hintergrund

Weiterführende Links

Shader-Programmierung 2: Post-Processing
Shader-Programmierung 3: Effekte
Shader-Effekt: Warping
Textscroller: Wellen und einzelne Farben
Raster bar Effekt
Sonnenblumen und der goldene Schnitt

Date Created
26. Mai 2023
Author
sven