Shader-Programmierung 3: Effekte

Description

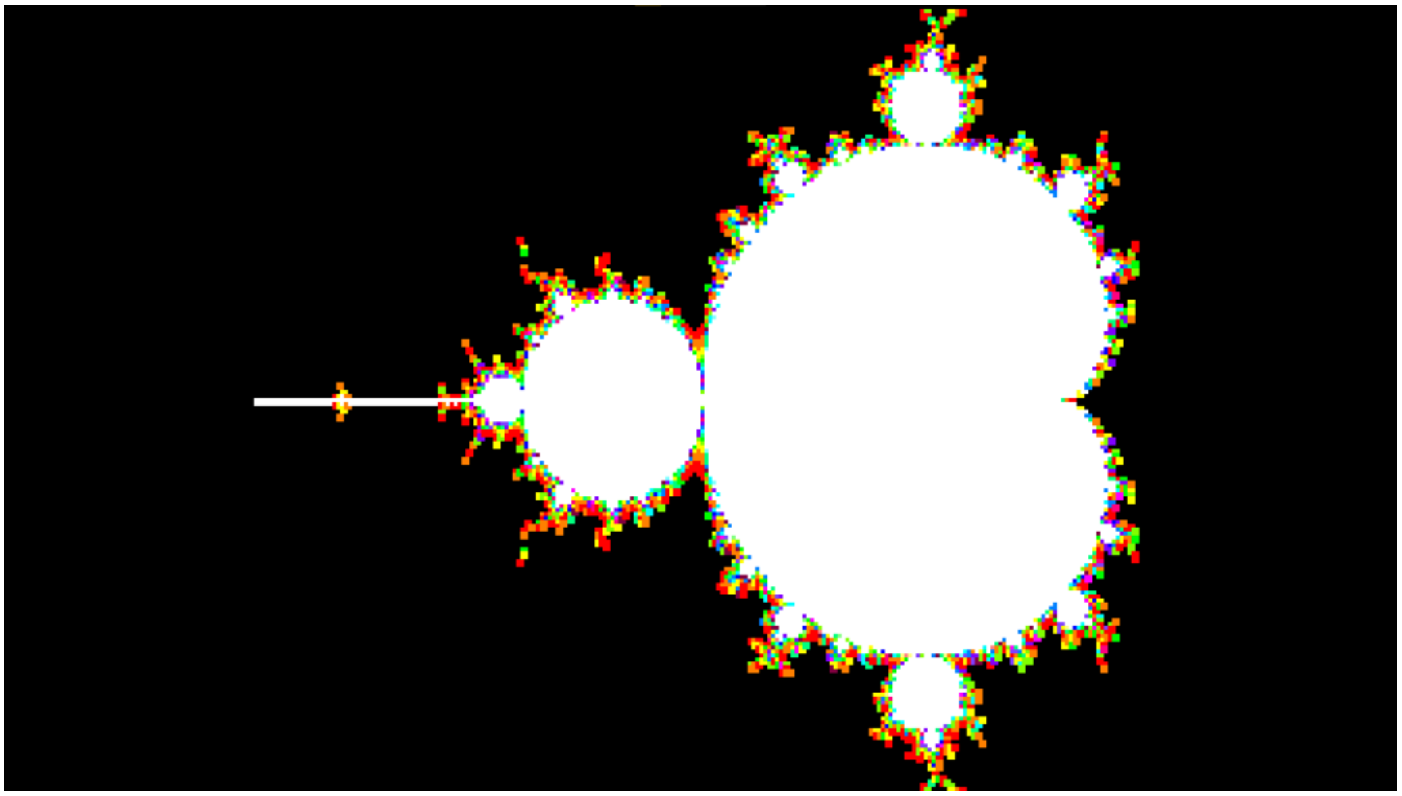
In den ersten beiden Teilen dieser Serie haben wir sehr viel über Shader gelernt. Neben der Manipulation von Sprites und Post-Processing gibt es aber noch andere Anwendungsmöglichkeiten. Mit Shadern kann man auch bildschirmfüllende Effekte programmieren.

Vorteile und Anwendung

Während wir beim Post-Processing lediglich das manipulieren, was bereits auf dem Bildschirm sichtbar ist, werden wie im dritten Teil ganz neue Inhalte generieren. Der Vorteil, dies mit Shadern zu machen, ist einfach erklärt: Mit Shadern sind Dinge möglich, die sich mit den Mitteln von Game Maker überhaupt nicht, oder nur sehr schwer umsetzen lassen. Die Zeichenfunktionen sind diesbezüglich sehr limitiert und vergleichsweise langsam.

Mandelbrot-Fraktal mit GM-Mitteln

Als Benchmark ein kleiner Test. Das folgende Beispiel zeichnet ein Mandelbrot-Fraktal mit `draw_rectangle()` auf ein Surface. Es wird im Create-Event vorberechnet und dauert, je nach Prozessor, vergleichsweise lange und sieht dazu noch recht bescheiden aus. Und nicht vergessen: Am Ende haben wir nur einen Frame.



Mandelbrot Fraktal im Game Maker

Create-Event

```
x = 0;
y = 0;

// Funktion zur Berechnung des Mandelbrot-Sets für eine gegebene komplexe Zahl
function calculate_mandelbrot(cx, cy)
{
 var zx = 0;
 var zy = 0;
 var value = 0;

 // Iterationen durchführen, bis der Wert des Mandelbrot-Sets bestimmt werden
 while (value < 256 && zx * zx + zy * zy < 4)
 {
     // Neue Z-Koordinaten berechnen
     var nzx = zx * zx - zy * zy + cx;
     var nzy = 2 * zx * zy + cy;
     zx = nzx;
     zy = nzy;
     value++;
 }

 // Wert des Mandelbrot-Sets zurückgeben
 return value;
}
```

```
width = 320;
height = 200;
surface = surface_create(width, height);

// Transparentes schwarzes Surface erstellen
transparent_black = surface_create(1, 1);
surface_set_target(transparent_black);
draw_set_color(c_black);
draw_set_alpha(1);  // Alpha-Wert auf 1 setzen
draw_rectangle(0, 0, 1, 1, false);
surface_reset_target();

for (var xx = 0; xx < width; xx++)
{
    for (var yy = 0; yy < height; yy++)
    {
        var cx = xx / width * 3.5 - 2.5;
        var cy = yy / height * 2 - 1;
        var value = calculate_mandelbrot(cx, cy);
  var color = c_white;
  if (value > 16)
  {
        if (value < 20) color = make_color_rgb(255, 0, 0);
        else if (value < 24) color = make_color_rgb(255, 128, 0);
        else if (value < 28) color = make_color_rgb(255, 255, 0);
        else if (value < 32) color = make_color_rgb(128, 255, 0);
        else if (value < 36) color = make_color_rgb(0, 255, 0);
        else if (value < 40) color = make_color_rgb(0, 255, 128);
        else if (value < 44) color = make_color_rgb(0, 255, 255);
        else if (value < 48) color = make_color_rgb(0, 128, 255);
        else if (value < 52) color = make_color_rgb(0, 0, 255);
        else if (value < 56) color = make_color_rgb(128, 0, 255);
        else if (value < 60) color = make_color_rgb(255, 0, 255);
        else if (value < 64) color = make_color_rgb(255, 0, 128);
   else if (value < 72) color = make_color_rgb(128, 0, 64);
   else if (value < 80) color = make_color_rgb(128, 0, 255);
   else if (value < 96) color = make_color_rgb(128, 64, 255);
   else if (value < 112) color = make_color_rgb(0, 128, 255);
   else if (value < 128) color = make_color_rgb(0, 255, 255);
   else if (value < 144) color = make_color_rgb(0, 255, 128);
   else if (value < 160) color = make_color_rgb(128, 255, 0);
   else if (value < 176) color = make_color_rgb(255, 128, 0);
   else if (value < 192) color = make_color_rgb(255, 255, 0);
   else if (value < 208) color = make_color_rgb(255, 128, 128);
   else if (value < 224) color = make_color_rgb(255, 255, 128);
   else if (value < 240) color = make_color_rgb(128, 255, 128);
   else if (value < 256) color = make_color_rgb(128, 255, 255);
        else color = c_white;
  } else {
   color = c_black;
  }

        surface_set_target(surface);
```

```
        if (color == c_black)
        {
            // Transparentes schwarzes Surface zeichnen
            draw_surface_ext(transparent_black, xx, yy, 1, 1, 0, color, 0);
        } else {
            // Nicht-schwarzen Pixel zeichnen
            draw_set_color(color);
            draw_rectangle(xx, yy, xx + 1, yy + 1, false);
        }
        surface_reset_target();
    }
}
```

Draw-Event

```
// Surface auf dem Bildschirm anzeigen
draw_surface(surface, 0, 0);
```

Wie man im Code sehen kann, hat das Fraktal nur eine Größe von 320×200 Pixel – und dauert dennoch recht lange für die Berechnung. Wie flott das im Shader geht, werden wir später sehen.

So weit zu den Vorteilen, aber wozu sind solche Effekte gut? Einerseits, wie so oft in solchen Tutorials, sind sie einfach schön anzusehen. Egal ob Intro, Fake-Cracktro, Hauptmenü oder Credits: Mit solchen Effekten lassen sich Spieler beeindrucken. Doch sie können auch spielrelevant sein. Man muss sich nur mal einen ausgefeilten Tunneleffekt im Hintergrund vorstellen, durch den ein Sprite-Raumschiff fliegt, um nur ein Beispiel zu nennen.

Abgesehen davon kann es extrem Spaß machen – und letztlich geht es genau darum bei der Programmierung.

Projektionsfläche

Es gibt mehrere Möglichkeiten, einen Shader-Effekt auf dem ganzen Bildschirm anzuzeigen. Im Tutorial möchte ich zwei davon zeigen, wobei ich in den Beispielen nur noch die zweite, einfachere Varianten anwenden werde.

Die Vertex-Texture-Methode

In Game Maker gibt es die Funktion `draw_vertex_texture()`. Sie definiert die Position eines texturierten Scheitelpunktes für ein Primitiv. Zur Verwendung müssen wir zunächst eine Sprite-Textur erstellen, wofür wir die Funktion `sprite_get_texture()` brauchen. Kurz gesagt: Wir werden eine Textur erstellen, die aus zwei Polygonen besteht. Darauf rendern wir den Shader Effekt.

Zur Verdeutlichung etwas Code:

```
shader_set(sh_shader);
// Wir legen die Parameter des Shaders fest
```

```
// Hier erstellen wir die Textur mit den beiden Polygonen
var tex = sprite_get_texture(s_shader, 0);
draw_primitive_begin_texture(pr_trianglestrip, tex);
draw_vertex_texture(offset_x, offset_y, 0, 0);
draw_vertex_texture(offset_x + width, offset_y, 1, 0);
draw_vertex_texture(offset_x + width, offset_y + height, 1, 1);

draw_vertex_texture(offset_x, offset_y + height, 0, 1);
draw_vertex_texture(offset_x, offset_y, 0, 0);
draw_vertex_texture(offset_x + width, offset_y + height, 1, 1);
draw_primitive_end();
shader_reset();
```

Der eigentliche Shader-Code fehlt hier, weil ich den Fokus auf die Textur legen möchte. Der Ablauf ist wie folgt:

1. Wir setzen den Shader.
2. Wir definieren die Parameter (`shader_set_uniform_f()` etc.)
3. Mit `sprite_get_texture()` wird eine Textur erstellt.
4. Mit `draw_vertex_texture()` definieren wir je drei Punkte eines Dreiecks. Das ist unser Polygon. Die beiden Dreiecke ergeben ein Rechteck.
5. Das Zeichnen wird mit `draw_primitive_end()` und `shader_reset()` beendet.

Viel Aufwand für wenig Effekt?

Tatsächlich ist es mehr Code, als die Methode, die gleich gezeigt wird. Sie ist wahrscheinlich auch etwas langsamer. Der Vorteil ist, dass man den Effekt mit dieser Methode etwas besser kontrollieren kann. Man kann Größe und Position zur Laufzeit ganz einfach ändern. Wenn man das Rechteck in viele weitere Polygone unterteilt, kann man so einen Effekt auch auf kreative Weise ein- und ausblenden.

Außerdem haben wir so einmal mit einer Textur gearbeitet, was man vor allem im 3D-Bereich gut gebrauchen kann.

Die Surface-Methode

Wer die ersten beiden Teile bereits durchgearbeitet hat, dem wird dies vertraut vorkommen.

https://www.bytegame.de/wp-content/uploads/2023/07/Klassischer-Plasma-Effekt.m

Create-Event

```
paletteShift = 0;
width = 320;
height = 200;
offset_x = 0;
offset_y = 0;
zoom = 0.75;
time = 0;
```

```
time_add = 0.01;
waveStrength = 0.02;
time_max = 20;
```

Draw-Event

```
paletteShift++;

if(time > time_max) {
    time_add = -time_add;
    waveStrength += 0.1;
}

time += time_add;

shader_set(sh_plasma01);
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"offset"),offset_x, offset
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"elapsedTime"),time);
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"waveStrength"),waveStreng
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"zoom"),zoom);
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"width"),width);
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"height"),height);
shader_set_uniform_f(shader_get_uniform(sh_plasma01,"paletteShift"),paletteShi

var tex = sprite_get_texture(s_shader, 0);
draw_primitive_begin_texture(pr_trianglestrip, tex);
draw_vertex_texture(offset_x, offset_y, 0, 0);
draw_vertex_texture(offset_x + width, offset_y, 1, 0);
draw_vertex_texture(offset_x + width, offset_y + height, 1, 1);

draw_vertex_texture(offset_x, offset_y + height, 0, 1);
draw_vertex_texture(offset_x, offset_y, 0, 0);
draw_vertex_texture(offset_x + width, offset_y + height, 1, 1);
draw_primitive_end();
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

#define PI 3.14159265359

uniform float waveStrength;
uniform float elapsedTime;
uniform float paletteShift;
uniform float width;
uniform float height;
uniform vec2  offset;
uniform float zoom;

float spectrum = 256.0;
```

```
void main()
{
 // Skalierung von v_vTexcoord mit dem Zoomfaktor
 vec2 texCoord = v_vTexcoord * zoom;
 // Hinzufügen des Offsets zu v_vTexcoord
 texCoord += offset;
 texCoord.x += sin(texCoord.y + elapsedTime) * waveStrength;

 // Berechnet den Farbwert
 float color = (sin(texCoord.x * width) + sin(texCoord.y * height)) * spectrum
 // Farbe „index" verschieben
 color = mod(color+paletteShift,spectrum);

 // Grün und Lila
 float r = (128.0 + (spectrum) * sin(color * PI / 128.0)) / 256.0;
 float g = (128.0 + (spectrum) * sin(color * PI / 64.0)) / 256.0;
 float b = (128.0 + (spectrum) * sin(color * PI / 32.0)) / 256.0;

 r = r * 0.5;
 b = b * 0.5;

 // Farbvektor erstellen und dem aktuellen Pixel bei x/y zuweisen
 vec4 finalColour = vec4(r,g,b,1.0);
 gl_FragColor = finalColour;
}
```

Die ursprünglichen Texturkoordinaten `v_vTexcoord` werden mit einem Zoom-Faktor multipliziert und um einen Offset-Wert verschoben. Dadurch kann der Effekt gezoomt und verschoben werden.

Die x-Komponente der verschobenen Texturkoordinaten wird modifiziert, indem eine Sinus-Funktion auf den y-Komponentenwert und die vergangene Zeit (`elapsedTime`) angewendet wird. Dadurch entsteht eine wellenartige Bewegung entlang der x-Achse. Die Farbe jedes Pixels wird basierend auf den modifizierten Texturkoordinaten berechnet. Es werden Sinus-Funktionen auf den x- und y-Komponentenwerten angewendet, und das Ergebnis wird mit einem Skalierungsfaktor (`spectrum`) und einem Wertebereich (`4.0`) multipliziert. Dadurch entsteht ein Farbwert, der zwischen -spectrum/4.0 und spectrum/4.0 liegt.

Der berechnete Farbwert (`color`) wird mit einem Verschiebungsfaktor (`paletteShift`) addiert und dann mit dem Spektrum-Wert (`spectrum`) modulo gerechnet. Dadurch entsteht eine Verschiebung in der Farbpalette. asierend auf dem verschobenen und modulierten Farbwert werden Sinus-Funktionen auf den einzelnen RGB-Komponenten angewendet, um die finale Farbe (`finalColour`) zu berechnen. Die Sinus-Funktionen verwenden verschiedene Perioden (PI / 128.0, PI / 64.0, PI / 32.0), um unterschiedliche Farbmuster zu erzeugen.

Die rote und blaue Komponente der finalen Farbe werden jeweils mit 0,5 multipliziert, um ihre Intensität zu verringern. Die finale Farbe (`finalColour`) wird dem aktuellen Pixel (`gl_FragColor`) zugewiesen, um es zu rendern.

Dank der Variable zoom im Create-Event kann man das Ganze auch vergrößern. So sieht es aus, wenn man statt 0.75 0.25 verwendet:

https://www.bytegame.de/wp-content/uploads/2023/07/Klassischer-Plasma-Effekt-Zoom.mp4

Realcolor Plasma

Der Effekt verwendet das Prinzip des Fractional Brownian Motion (fbm) mit mehreren Oktaven, um ein detailreiches und sich veränderndes Muster zu erzeugen. Der Shader nutzt Noise-Funktionen, Rotationen und Transformationen, um die Formen zu generieren. Je nach Farbwahl können somit Feuer, Nebel oder andere, sehr kreative Muster erzeugt werden. Mit der Variable `time_add` im Create-Event können wir die Geschwindigkeit steuern. Setzt man sie auf 0, erhält man ein Standbild.

https://www.bytegame.de/wp-content/uploads/2023/07/Realcolor-Plasma.mp4

Create-Event

```
time = 0;
time_add = 0.02;
```

Draw-Event

```
time += time_add;

shader_set(sh_plasma02);
shader_set_uniform_f(shader_get_uniform(sh_plasma02,"resolution"), display_get
shader_set_uniform_f(shader_get_uniform(sh_plasma02,"time"),time);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

#ifdef GL_ES
precision highp float;
#endif

// #extension GL_OES_standard_derivatives : enable

#define NUM_OCTAVES 14

uniform float time;
uniform vec2 resolution;

mat3 rotX(float a) {
    float c = cos(a);
    float s = sin(a);
    return mat3(
        1, 0, 0,
        0, c, -s,
```

```
        0, s, c
    );
}
mat3 rotY(float a) {
    float c = cos(a);
    float s = sin(a);
    return mat3(
        c, 0, -s,
        0, 1, 0,
        s, 0, c
    );
}

float random(vec2 pos) {
    return fract(sin(dot(pos.xy, vec2(1349.9898, 78.233))) * 43758.5453123);
}

float noise(vec2 pos) {
    vec2 i = floor(pos);
    vec2 f = fract(pos);
    float a = random(i + vec2(0.0, 0.0));
    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));
    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;
}

float fbm(vec2 pos) {
    float v = 0.01;
    float a = 0.6;
    vec2 shift = vec2(50.0);
    mat2 rot = mat2(cos(0.5), sin(0.5), -sin(0.5), cos(0.5));
    for (int i=0; i<NUM_OCTAVES; i++) {
        v += a * noise(pos);
        pos = rot * pos * 2.0 + shift;
        a *= 0.5;
    }
    return v;
}

void main(void) {
    vec2 p = (gl_FragCoord.xy * 1.0 - resolution.xy) / min(resolution.x, resol

    float t = 0.0, d;

    float time2 = 0.6 * time / 2.0;

    vec2 q = vec2(0.0);
    q.x = fbm(p + 0.30 * time2);
    q.y = fbm(p + vec2(1.0));
    vec2 r = vec2(0.0);
    r.x = fbm(p + 1.0 * q + vec2(1.2, 3.2) + 0.135 * time2);
    r.y = fbm(p + 1.0 * q + vec2(8.8, 2.8) + 0.126 * time2);
    float f = fbm(p + r);
```

```
/// Feuer
vec3 color = mix(
    vec3(1, 0, 0), // Rot
    vec3(1, 0.5, 0), // Orange
    clamp((f * f) * 2.0, 0.0, 1.0)
);

color = mix(
    color,
    vec3(1, 1, 0), // Gelb
    clamp(length(q), 0.0, 1.0)
);

color = mix(
    color,
    vec3(0, 0, 0), // Schwarz
    clamp(length(r.x), 0.0, 1.0)
);
```

Der Shader verwendet eine fbm-Funktion, um den komplexen, sich verändernden Mustern zu erzeugen. Die Funktion `fbm` nimmt eine Position (`pos`) und verwendet mehrere Oktaven von Noise, um einen Wert zu berechnen. Dieser Wert wird dann aufaddiert und skaliert, um den Gesamteffekt zu erzeugen.

Die Funktion `noise` wird verwendet, um zufällige Werte basierend auf einer Position zu generieren. Diese Werte werden für die Berechnung des `fbm` und für die Variation der Flammenformen genutzt. Es werden die Funktionen `rotX` und `rotY` verwendet, um Matrizen für die Rotationen zu erstellen.

Die Farben der Flammen werden basierend auf den erzeugten Werten berechnet. Es werden Farbmischungen (Mixing) verwendet, um verschiedene Farben basierend auf den Werten zu mischen. Zum Beispiel werden Rottöne, Orange und Gelbtöne verwendet, um die Farben der Flammen darzustellen.

Außerdem erzeugt der Shader einen Flickereffekt, indem er den Wert `f` quadriert und dann als Faktor für die Farbmischungen verwendet. Dadurch entsteht ein flackernder Effekt, der die Flammen darstellt.

Spiral Tunnel

Tunneleffekte sind immer beeindruckend, weil dem Betrachter eine dreidimensionalität vorgegaukelt wird, die eigentlich nicht existiert.

https://www.bytegame.de/wp-content/uploads/2023/07/Tunnel-Effekt.mp4

Create-Event

```
time = 0;
time_add = 0.2;
```

Draw-Event

```
time += time_add;

shader_set(sh_tunnel01);
shader_set_uniform_f(shader_get_uniform(sh_tunnel01,"resolution"), display_get
shader_set_uniform_f(shader_get_uniform(sh_tunnel01,"time"),time);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

precision highp float;

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

void main( void )
{
    vec2 U = gl_FragCoord.xy;
    vec4 f = resolution.xyxy;

 f = length(U+=U-f.xy)/f;
 f = sin(f.w-.1) * vec4(sin(6./f + atan(U.x,U.y)*10. - time).w < sin(f.w-.1));

 float wave = sin(f.w-.1) * sin(f.w-.1) * sin(f.w-.1);
 U.y += wave * 50.0;

 gl_FragColor = vec4(0, f.x * 0.8, 0, 1.0);
}
```

Der Shader verwendet die Fragmentkoordinaten `gl_FragCoord.xy`, um die Position jedes Pixels im Bildschirmraum zu erhalten. Die Koordinaten `U` werden durch die Bildschirmauflösung `resolution.xy` skaliert, um sicherzustellen, dass das Muster unabhängig von der Bildschirmgröße richtig dargestellt wird.

Durch die Verwendung der Länge (`length`) der verschobenen Koordinaten `U+=U-f.xy` wird die Distanz jedes Pixels zum Mittelpunkt des Bildschirms berechnet. Dadurch entsteht eine Art Verzerrung, die das Tunnelmuster erzeugt.

Es werden Sinusfunktionen verwendet, um die Spiralbewegung des Tunnels darzustellen. Der Ausdruck `sin(6./f + atan(U.x,U.y)*10. - time)` erzeugt eine sich ändernde Spirale basierend auf den Koordinaten und der Zeit.

Die Farbe des Pixels wird basierend auf den berechneten Werten festgelegt. Die grüne Komponente (`f.x * 0.8`) wird anhand des Distanzwerts `f.x` und einer Konstante multipliziert, um den Grünton

des Tunnels zu steuern. Die finale Farbe wird dem Pixel (`gl_FragColor`) zugewiesen, um es zu rendern.

Wenn man die Farbe Orange erhalten möchte, muss man die letzte Zeile nur etwas abändern:

```
gl_FragColor = vec4(f.x * 0.8, f.x * 0.4, 0, 1.0);
```

Variation

Der Effekt lässt sich auch leicht anpassen. Durch ein paar kleine Änderungen…

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

precision highp float;

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

void main( void )
{
    vec2 U = gl_FragCoord.xy;
    vec4 f = resolution.xyxy;

 f = length(U+=U-f.xy)/f;
 f = sin(f.w-.1) * vec4(sin(24./f + atan(U.x,U.y)*40. - time*4.).w < sin(f.w-.

 float twist = sin(f.w-.1) * sin(f.w-.1) * sin(f.w-.1);
 U.x += twist * 10.0;
 U.y += twist * 10.0;

 gl_FragColor = vec4(f.x * 0.8, f.x * 0.4, 0, 1.0); // Orange
}
```

…sieht der Effekt so aus:

https://www.bytegame.de/wp-content/uploads/2023/07/Tunnel-Effekt2.mp4

Spiraleffekt

Dieser Effekt erzeugt eine Darstellung einer sich drehenden Spirale mit verschiedenen Farben. Ich finde, er wäre ganz gut für Credits geeignet.

https://www.bytegame.de/wp-content/uploads/2023/07/Spiraleffekt.mp4

Create-Event

```
time = 0;
time_add = 0.02;
```

Draw-Event

```
time += time_add;
shader_set(sh_spiral01);
shader_set_uniform_f(shader_get_uniform(sh_spiral01,"resolution"), display_get
shader_set_uniform_f(shader_get_uniform(sh_spiral01,"time"),time);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

precision highp float;

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

void main( void ) {
 vec2 p =  (-resolution.xy + 2.0 * gl_FragCoord.xy) / resolution.y;
 p = vec2( length(p), atan(p.x, p.y) );
 vec2 q = vec2( atan(p.x,p.y), 3.*length(dot(.2*sin(p),p)/1.0) );

 float f = smoothstep( -0.2, 0.5, cos(q.x + q.y * 30.0 - time * 5.0) );
 float g = smoothstep( -01.0, 0.5, sin(q.y + q.x * 320.0 + time * 30.0) * q.y

 vec3 col = mix( vec3(0.0, 0.16, 0.32), vec3(0.53, 0.81, 0.92), f );
 col = mix( col, vec3(0.0, 0.27, 0.56), sign(f) );

 gl_FragColor = vec4( col, 12.0 );
}
```

Die Koordinaten p werden durch die Bildschirmauflösung `resolution` und die Position jedes Pixels `gl_FragCoord.xy` skaliert und verschoben. Außerdem werden die Koordinaten p in Polarkoordinaten umgewandelt. Die Länge des Vektors p entspricht dem Abstand des Pixels vom Mittelpunkt, und der Winkel `atan(p.x, p.y)` gibt die Richtung an.

Die Werte `q.x` und `q.y` basieren auf den Polarkoordinaten p und steuern die Spiralenbewegung. Es wird die Funktion `dot(.2*sin(p),p)` verwendet, um die Muster innerhalb der Spirale zu erzeugen.

Die Werte `f` und `g` werden durch die Funktion `smoothstep` geglättet und dienen als Faktoren für die Farbmischungen. Die Farben werden mit Hilfe von `mix` und `sign` gemischt, um den Übergang zwischen verschiedenen Farbtönen zu steuern. Die Farbe `col` wird basierend auf den berechneten Werten festgelegt. Es werden verschiedene Blautöne verwendet, um die Spirale darzustellen. Diefinale Farbe wird dem Pixel (`gl_FragColor`) zugewiesen, um es zu rendern. Der Alpha-Wert (`12.0`)ist in diesem Fall fest vorgegeben.

Variation

Hier kann man ebenfalls mit intensiveren Farben arbeiten. Eine Variation sähe so aus:

https://www.bytegame.de/wp-content/uploads/2023/07/Spiraleffekt2.mp4

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

precision highp float;

uniform float time;
uniform vec2 mouse;
uniform vec2 resolution;

void main( void ) {
    vec2 p =  (-resolution.xy + 2.0 * gl_FragCoord.xy) / resolution.y;
    p = vec2( length(p), atan(p.x, p.y) );
    vec2 q = vec2( atan(p.x,p.y), 3.*length(dot(.2*sin(p),p)/1.0) );

    float f = smoothstep( -0.2, 0.5, cos(q.x + q.y * 30.0 - time * 5.0) );
    float g = smoothstep( -1.0, 0.5, sin(q.y + q.x * 320.0 + time * 30.0) * q.

    vec3 col = mix( vec3(1.0, 0.0, 0.0), vec3(1.0, 1.0, 0.0), f );
    col = mix( col, vec3(0.0, 1.0, 0.0), sign(f) );

    gl_FragColor = vec4( col, 1.0 );
}
```

Fraktale

Nun kommen wir endlich zu den Fraktalen. Ich zeige zwei Varianten. In beiden Fällen zoomen wir hinein, wir haben also nicht nur eine einzelnes Bild. Der zweite Effekt ist zudem noch etwas speziell, aber schauen wir uns zunächst die einfachere Variante an.

https://www.bytegame.de/wp-content/uploads/2023/07/Fraktal-Effekt1.mp4

Create-Event

```
time = 20.11;

offset_x = 0;
```

```
offset_y = 0;
width = room_width;
height = room_height;

res_x = 1;
res_y = 1;
rot = 0;
```

Step-Event

```
if (mouse_wheel_up())
{
   time += 0.02;
}
else if (mouse_wheel_down())
{
   time -= 0.02;
}

if (res_x < 23)
{
 res_x *= 1.00315;
 res_y *= 1.0032;
} else if (res_x < 60) {
 res_x *= 1.0032;
 res_y *= 1.0032;
}
```

Draw-Event

```
shader_set(sh_fractal02);
shader_set_uniform_f(shader_get_uniform(sh_fractal02,"time"),time);
shader_set_uniform_f(shader_get_uniform(sh_fractal02,"screen_size"),res_x,res_
draw_surface_ext(application_surface, 0, 0, 1, 1, rot, c_white, 0);

shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

uniform float time;
uniform vec2 screen_size;

float distanceToMandelbrot(const vec2 c)
{
    if(256.0 * c.x * c.x - 96.0 * c.x + 32.0 * c.y - 3.0 < 0.0) return 0.0;
    if(16.0 * (c.x * c.x + 2.0 * c.x + 1.0) - 1.0 < 0.0) return 0.0;

    float di = 1.0;
    vec2 z = vec2(0.0, 0.0);
    float m2 = 0.0;
```

```
    vec2 dz = vec2(0.0, 0.0);

    for(int i = 0; i < 300; i++)
    {
        if(m2 > 1024.0)
        {
            di = 0.0;
            break;
        }

        dz = vec2(z.x * dz.x - z.y * dz.y, z.x * dz.y + z.y * dz.x) + vec2(1.0
        z = vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y) + c;
        m2 = dot(z, z);
    }

    float d = 0.5 * sqrt(dot(z, z) / dot(dz, dz)) * log(dot(z, z));
 if(di > 0.5) d = 0.0;

    return d;
}

vec4 colorFromPalette(float t)
{
    vec4 color1 = vec4(1.0, 0.0, 0.0, 1.0);   // Rot
    vec4 color2 = vec4(1.0, 0.5, 0.0, 1.0);   // Orange
    vec4 color3 = vec4(1.0, 1.0, 0.0, 1.0);   // Gelb
    vec4 color4 = vec4(0.0, 1.0, 0.0, 1.0);   // Grün
    vec4 color5 = vec4(0.0, 0.0, 1.0, 1.0);   // Blau
    vec4 color6 = vec4(0.5, 0.0, 0.5, 1.0);   // Violett

    if (t < 1.0)
    {
        return mix(color4, color5, (t - 60.0) * 0.05);
    }
    else
    {
        return mix(color5, color6, (t - 80.0) * 0.05);
    }
}

void main()
{
    vec2 p = (2.0 * v_vTexcoord - screen_size) / screen_size.y;

    float tz = 0.5 - 0.5 * cos(0.225 * time);
    float zoo = pow(0.5, 13.0 * tz);
    vec2 c = vec2(-0.05, 0.6805) + p * zoo;

    float d = distanceToMandelbrot(c);
    d = clamp(pow(4.0 * d / zoo, 0.2), 0.0, 1.0);

 vec4 col = colorFromPalette(d);

    gl_FragColor = col;
```

```
}
```

Im Prinzip besteht alles nur aus zwei Farben, dem blauen Hintergrund und das grüne Fraktal. Dabei zoomen wir mehr oder weniger wahllos rein. Per Mausrad können wir das sogar, über das Step-Event, ein wenig steuern.

Die Funktion `distanceToMandelbrot` berechnet die Distanz eines Punktes im Komplexen Raum zur Mandelbrot-Menge. Sie verwendet eine iterative Berechnung mit einem Schwellenwert, um zu bestimmen, ob ein Punkt innerhalb der Mandelbrot-Menge liegt oder nicht.

Die Funktion `colorFromPalette` weist einem Wert `t` aus dem Bereich [0, 1] eine Farbe aus einer vordefinierten Palette zu. Abhängig von `t` wird die Farbe aus einer Mischung von Rot, Grün und Blau bestimmt.

Die `main`-Funktion verwendet den Mandelbrot-Algorithmus, um für jeden Pixel den entsprechenden Punkt im Komplexen Raum zu berechnen und die Distanz zur Mandelbrot-Menge zu ermitteln. Basierend auf dieser Distanz wird ein Farbwert aus der Palette zugewiesen und als Ausgabefarbe verwendet.

Komplexes Fraktal mit Zoom

Die zweite Variante ist um einiges Komplexer. Der Zoom ist nicht wahllos. Ich habe hierfür ein kleines System geschrieben, mit dem ich zunächst die Bewegungen per Hand (Maus und Richtungstasten) erfasst und gespeichert habe. Anschließend habe ich die Werte genommen und die Bewegungen interpoliert.

https://www.bytegame.de/wp-content/uploads/2023/07/Fraktal-Effekt2.mp4

Im Code zeige ich nur das Resultat, weil es sonst den Rahmen dieses Tutorials sprengen würde. Der Shader-Code erzeugt ein Mandelbrot-Fraktal, bei dem die Farben basierend auf dem HSV-Farbraum zyklisch angepasst werden. Du kannst die Parameter wie `xx`, `yy`, `zoom` und die Geschwindigkeitsrate anpassen, um das Fraktal und die Farben zu verändern.

Create-Event

```
time = 0;
time_add = 0.02;

// tool
sspeed = 5;
zoom = 2*sspeed;
xx = -1;
yy = 0;
time_start = current_time;
index = 1;
record = false;

values[0][0] = xx;
values[0][1] = yy;
```

```
values[0][2] = time_start;

values[1][0] = -1.14;
values[1][1] = -0.16;
values[1][2] = 9278;

values[2][0] = -1.19;
values[2][1] = -0.28;
values[2][2] = 17412;

values[3][0] = -1.17;
values[3][1] = -0.27;
values[3][2] = 30646;

values[4][0] = -1.16;
values[4][1] = -0.26;
values[4][2] = 34713;

values[5][0] = -1.16;
values[5][1] = -0.27;
values[5][2] = 54015;

values[6][0] = -1.16;
values[6][1] = -0.27;
values[6][2] = 95167;
```

Step-Event

```
var index = 0;

// In jedem Frame überprüfen wir, ob die aktuelle Zeit größer als der Wert in
while (index < array_length(values) - 1 && current_time > values[index+1][2]/s
{
 index++;
}

if (index < array_length(values)-1)
{
 var t = clamp((current_time - values[index][2]/sspeed) / (values[index + 1][2

 // Wir interpolieren zwischen den Werten in den ersten und zweiten Spalten
 xx = lerp(values[index][0], values[index + 1][0], t);
 yy = lerp(values[index][1], values[index + 1][1], t);
}
```

Draw-Event

```
time += time_add;

shader_set(sh_fractal04);
shader_set_uniform_f(shader_get_uniform(sh_fractal04,"resolution"), display_ge
shader_set_uniform_f(shader_get_uniform(sh_fractal04,"time"),time);
shader_set_uniform_f(shader_get_uniform(sh_fractal04,"xx"),xx);
```

```
shader_set_uniform_f(shader_get_uniform(sh_fractal04,"yy"),yy);
shader_set_uniform_f(shader_get_uniform(sh_fractal04,"zoom"),zoom);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

uniform vec2 resolution;
uniform float time;
uniform vec2 mouse;
uniform float xx;
uniform float yy;
uniform float zoom;

vec4 hsv2rgba(in vec3 hsv) {
 // Dunkelblau hat den HSV-Wert (240, 1, 0.6)
 vec3 baseColor = vec3(240.0/360.0, 1.0, 0.6);

 // Berechne die RGB-Werte für das dunkelblaue Fraktal
 float h = baseColor.x;
 float s = baseColor.y;
 float v = baseColor.z;
 vec3 k = vec3(1.0, 2.0/3.0, 1.0/3.0);
 vec3 p = clamp(abs(6.0*fract(h - k) - 3.0) - 1.0, 0.0, 1.0);
 vec3 baseRGB = v * mix(k.xxx, p, s);

 // Füge dem dunkelblauen Fraktal den Wert aus dem HSV-Farbraum hinzu
 hsv.x = hsv.x + baseColor.x;
 hsv.y = hsv.y * baseColor.y;
 hsv.z = hsv.z * baseColor.z;

 // Berechne die RGB-Werte für die angepasste Farbe
 h = hsv.x;
 s = hsv.y;
 v = hsv.z;
 k = vec3(1.0, 2.0/3.0, 1.0/3.0);
 p = clamp(abs(6.0*fract(h - k) - 3.0) - 1.0, 0.0, 1.0);
 vec3 hsvRGB = v * mix(k.xxx, p, s);

 // Misch die RGB-Werte des dunkelblauen Fraktals und der angepassten Farbe
 float mixAmount = 0.5;
 vec3 mixedRGB = mix(baseRGB, hsvRGB, mixAmount);

 return vec4(mixedRGB, 1.0);
}

vec4 hsvCycled2rgba(in vec3 hsv, in float spread, in float speed) {
 hsv.x = fract(hsv.x*spread + time*speed);
 return hsv2rgba(hsv);
}
```

```
int mandelbrot(in vec2 p) {
 vec2 t = vec2(0.0, 0.0);
 const int maxIterations = 1000;

 for (int i = 0; i < maxIterations - 1; i++)
 {
  if (t.x*t.x + t.y*t.y > 4.0)
  {
      return i;
  }
  t = vec2(t.x*t.x - t.y*t.y, 2.0*t.x*t.y) + p;
 }

 return -1;
}

void setColor(out vec4 fragColor, in vec4 fragCoord) {
 vec2 c = resolution*0.5;
 vec2 uv = fragCoord.xy;

 vec2 offset = vec2(xx, yy);
 float zoomAmount = zoom;

 float t = time * 0.1;
 float scale = resolution.y * 0.35 * pow(zoom, t);
 vec2 p = (uv - c + vec2(xx, yy))/(scale) + offset;

 int iterations = mandelbrot(p);
 if (iterations < 0)
 {
  fragColor = vec4(0.0, 0.0, 0.05, 1.0);
  return;
 }

 float value = float(iterations)/-36.0;
 vec3 hsv = vec3(value, 1.0, 1.0);
 fragColor = hsvCycled2rgba(hsv, 1.0, -1.0/10.0);
}

void main(void) {
 setColor(gl_FragColor, gl_FragCoord);
}
```

Das ist viel Holz. Die Funktion `hsv2rgba` wandelt einen HSV-Farbvektor in einen RGBA-Farbvektor um. Sie verwendet eine Formel, um die RGB-Werte für das dunkelblaue Fraktal zu berechnen und fügt dann die Werte aus dem HSV-Farbraum hinzu, um die angepasste Farbe zu erhalten. Die Funktion `hsvCycled2rgba` nimmt einen HSV-Farbvektor, einen Spread-Wert und eine Geschwindigkeitsrate entgegen. Sie zyklisiert den HSV-Wert basierend auf der Zeit und ruft dann `hsv2rgba` auf, um den entsprechenden RGBA-Farbvektor zu erhalten.

Mit der Funktion Funktion `mandelbrot` implementiert man den Mandelbrot-Algorithmus. Sie iteriert über eine komplexe Zahlenreihe und prüft, ob die Iteration gegen Unendlich divergiert. Wenn dies der Fall ist, wird die Anzahl der Iterationen zurückgegeben.

Die Funktion `setColor` setzt die Farbe für einen Fragment-Shader. Sie berechnet die Koordinaten und den Zoom-Faktor für das Mandelbrot-Fraktal basierend auf den Eingabewerten. Anschließend ruft sie `mandelbrot` auf, um die Anzahl der Iterationen für den Punkt im Fraktal zu erhalten. Basierend auf den Iterationen wird ein HSV-Farbvektor erstellt und dann mit `hsvCycled2rgba` in einen RGBA-Farbvektor umgewandelt. Dieser Farbvektor wird dann als Ausgabefarbe gesetzt. Die `main`-Funktion ruft `setColor` auf und verwendet `gl_FragColor` als Ausgabevariable für die Farbe des Fragments.

Das Beispiel zeigt, dass einerseits vieles möglich ist und man durchaus zwischen Game Maker und Shader-Effekt kommunizieren kann, was dazu führt, dass sich für Spiele auch dynamische Effekte erzeugen lassen. So lassen sich beispielsweise Zoom, Geschwindigkeit und Farben dem Spielgeschehen anpassen.

Crossing Bars

Zum Ende noch einen Effekt, den ich in alten Demos sehr geliebt habe. Er besteht aus mehreren rotierenden Balken.

https://www.bytegame.de/wp-content/uploads/2023/07/Crossing-Bars.mp4

Create-Event

```
time = 0;
time_add = 0.01;
```

Draw-Event

```
time += time_add;
shader_set(sh_crossing_bars);
shader_set_uniform_f(shader_get_uniform(sh_crossing_bars,"resolution"), displa
shader_set_uniform_f(shader_get_uniform(sh_crossing_bars,"time"),time);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Fragment-Shader

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

uniform float time;
uniform vec2 resolution;

#define PI 3.14159265
float map(vec2 uv, float angle, float dist) {
    float newX = uv.x * cos(angle) - uv.y * sin(angle);
```

```
    return sin((newX - dist/2.0) * dist) > 0.0 ? 1.0 : 0.0;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = (fragCoord - 0.5 * resolution.xy) / resolution.y;

    float angle = time * 8.0;
    float aLag = (PI / 4.0) * sin(time * 0.7);
    float c = 0.0;

    // Anzahl der Schattierungen
    int numShades = 4;
    float shadeIntensity = 1.0 / float(numShades);

    for (int i = 0; i < numShades; i++) {
        angle += aLag;
        c += map(uv, angle, 25.0) * shadeIntensity;
    }

    // Farbeinstellungen für Hintergrund und Balken
    vec3 bgColor = vec3(0.0, 0.0, 0.0); // Hintergrundfarbe (schwarz)
    vec3 elementColor = vec3(0.0, 0.8, 0.8); // Farbe der Balken

    // Mischung der Farben basierend auf dem Schattierungswert
    vec3 finalColor = mix(bgColor, elementColor, c);

    fragColor = vec4(finalColor, 0.8);
}

void main(void)
{
    mainImage(gl_FragColor, gl_FragCoord.xy);
}
```

Mit der Funktion `map` wird bestimmt, ob ein Punkt auf einem Balken liegt oder nicht. Sie nimmt die UV-Koordinaten des Punktes, einen Winkel und eine Distanz als Parameter. Sie berechnet eine neue X-Koordinate unter Berücksichtigung des Winkels und überprüft dann, ob diese X-Koordinate innerhalb des Balkens liegt, indem sie die Sinusfunktion verwendet. Wenn der Punkt innerhalb des Balkens liegt, gibt die Funktion 1.0 zurück, andernfalls 0.0.

In der Funktion `mainImage` wird zuerst die Normalisierung der Bildkoordinaten durchgeführt, um die UV-Koordinaten im Bereich von -0.5 bis 0.5 zu erhalten. Der Winkel `angle` wird basierend auf der Zeit aktualisiert, und `aLag` beeinflusst die Variation des Winkels.

Die Schattierungen der Balken werden durch die Schleife erzeugt. Für jeden Schritt in der Schleife wird der Winkel angepasst und die Funktion `map` verwendet, um zu überprüfen, ob der Punkt innerhalb des Balkens liegt. Die Schattierungsintensität wird berechnet und zur Variablen `c` addiert. Die Farbeinstellungen für den Hintergrund (`bgColor`) und die Balken (`elementColor`) können angepasst werden. In diesem Fall ist der Hintergrund schwarz und die Balkenfarbe ein Türkis.

Schließlich wird die endgültige Farbe für den Punkt basierend auf der Schattierung (c) und den

Farbeinstellungen gemischt. Der Alpha-Wert (Transparenz) ist auf 0,8 eingestellt. Die `main`-Funktion ruft `mainImage` auf und übergibt die Ausgabefarbe und die Koordinaten des Fragments.

Fazit und Ausblick

Das war nun sehr viel Code, aber ich denke, dass die zahlreichen Beispiele das Verständnis vertiefen werden. Natürlich wären noch sehr viele andere Effekte möglich gewesen. Mit der Zeit werde ich weitere solcher Effekte als einzelne Tutorials veröffentlichen.

Im nächsten und letzten Schritt dieser kleinen Serie geht es dann richtig in die dritte Dimension.

Date Created
21. Juli 2023
Author
sven