Shader-Programmierung 2: Post-Processing

Description

Shader werden, vor allem bei 2D-Spielen, überwiegend für Post-Processing-Effekte eingesetzt. Im zweiten Teil gibt es mehrere Praxisbeispiele dafür und es wird gezeigt, wie man es in GameMaker Studio 2 implementiert. Doch vorher müssen wir ein paar Umwege gehen.

Was versteht man unter Post-Processing?

Post-Processing-Effekte werden normalerweise final über den ganzen Bildschirm gelegt. Das heißt, alles, was der Betrachter sehen kann, wird vor der Ausgabe auf irgendeine Weise manipuliert. Diese Manipulation wird normaler eingesetzt, um das gesamte Aussehen in eine bestimmte, optische Richtung zu verschieben. Die Art, wie man das machen kann, ist sehr vielfältig. Dabei kann es sein, dass solche Effekte permanent über das Spiel gelegt werden, oder nur für eine bestimmte Zeit sowie nur in bestimmten Räumen. Außerdem kann ein Post-Processing-Effekt sowohl statisch wie auch dynamisch sein.

Beispiele für Post-Processing-Effekte:

- Intensivere oder schwächere Farben
- Scharfzeichnung oder Weichzeichnung
- Bewegungsunschärfe
- Grundlegende Farbmanipulation, etwa eine andere Farbpalette
- Änderung von Helligkeitswerten
- Wettereffekte über dem Bildschirm
- CRT-Effekt
- chromatische Aberration
- Bildstörungen
- Bildschirm verwackeln
- Textur-Überlagerung
- Belichtung
- Einfärben
- verpixeln

- u. v. m.

Anmerkung: Für die Beispiele habe ich zwei Sets gebastelt, weil die Shader, je nach Grafiken, unterschiedlich gut wirken. Das eine Set erinnert an ganz alte Sierra On-Line Spiele wie **Police Quest I**, sind aber selbst erstellte Grafiken. Die Links zu den Grafiken des zweiten Sets befinden sich am Ende des Tutorials.

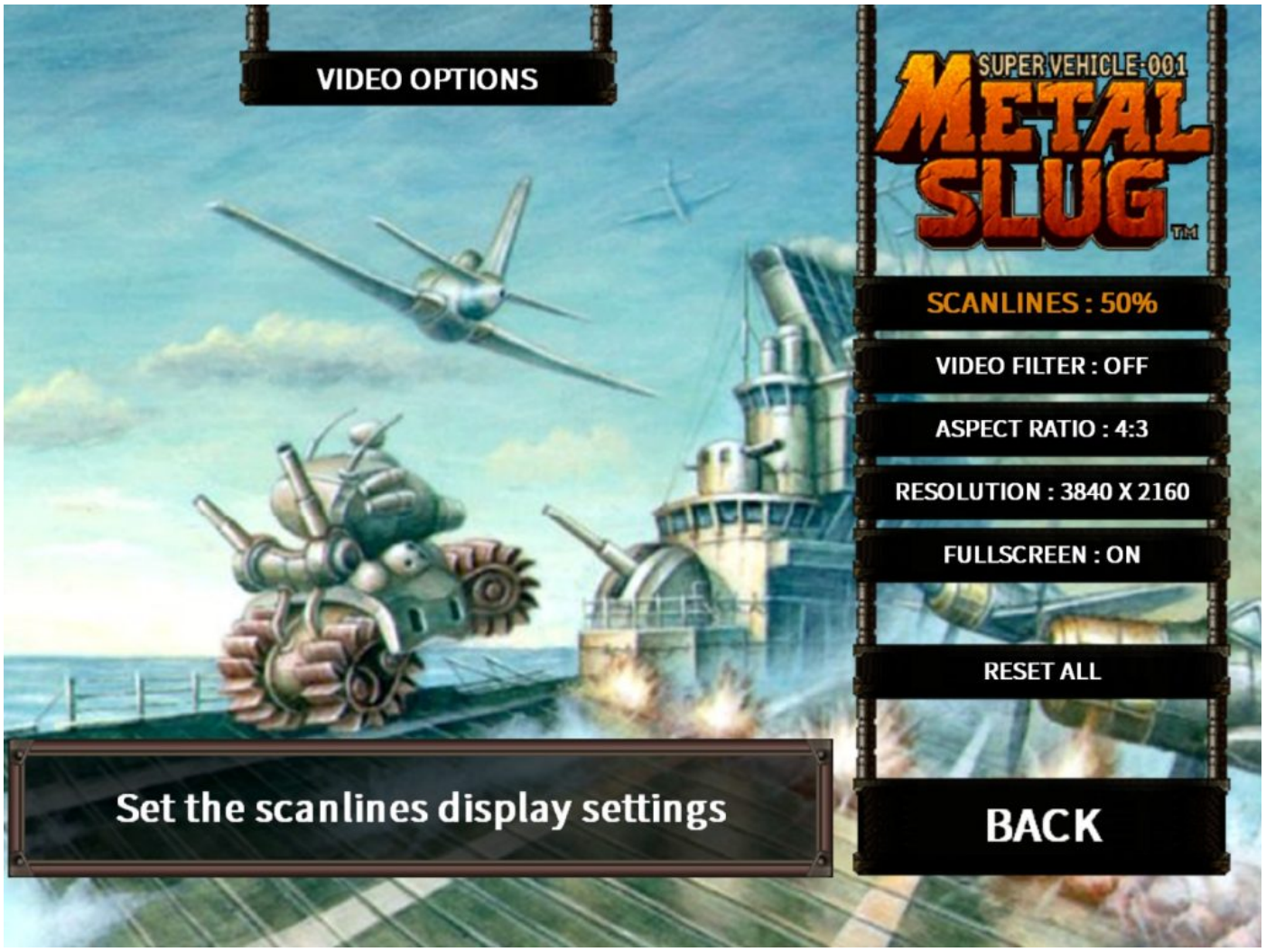
Praktische Beispiele für Post-Processing-Effekte

Wenn man nicht so im Thema ist, fragt man sich natürlich, ob man die meisten Effekte nicht einfach so einspart, indem man entsprechende Grafiken erstellt. Oder anders gefragt: Warum macht man es nicht gleich richtig?

So simpel ist das leider nicht. Einerseits lassen sich bestimmte Effekte auf der Sprite-Ebene nur schwer oder überhaupt nicht umsetzen, andererseits spielt die Dynamik eine wichtige Rolle. D. h. es wäre bspw. schön, wenn der Spieler selbst einige Entscheidungen treffen könnte.

In AAA-Spielen werden Post-Processing-Shader gerne kontinental eingesetzt, etwa was den Farbraum bzw. die Farbintensität betrifft. Spiele in den USA haben meist knalligere Farben, in Europa hingegen sind sie etwas dezenter. Der Spieler kann diese Einstellungen i. d. R. ändern.

Ähnliches gilt für CRT- und Verpixelungseffekte. Es wäre schön, wenn der Spieler selbst entscheiden könnte, wie „retro“ das Resultat aussehen soll. Ein gutes Beispiel hierfür ist die GOG-Version von **METAL SLUG**.



In METAL SLUG kann man in den Optionen bestimmen, wie viel Retro man haben möchte

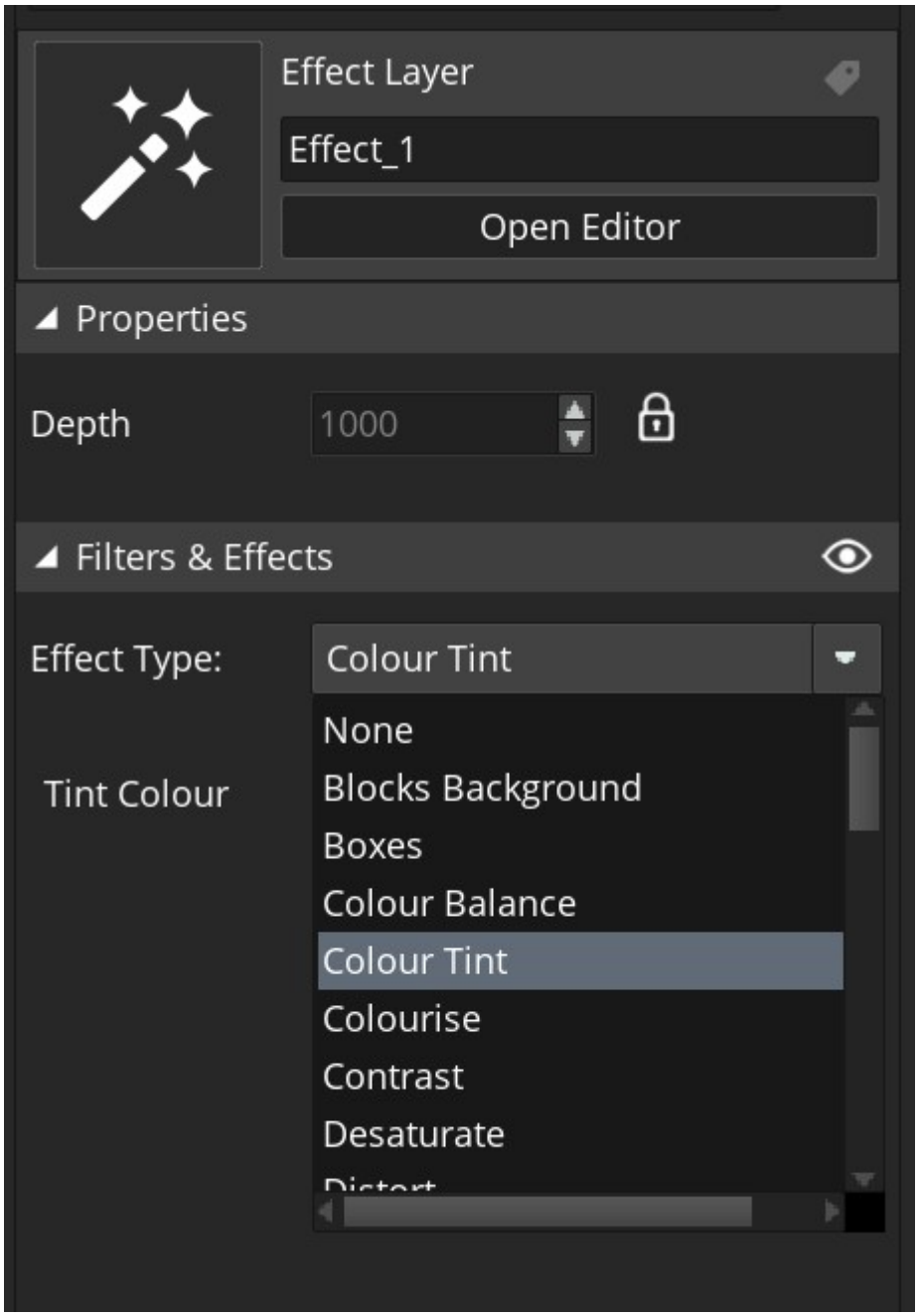
Ein weiteres Beispiel wären Spielelemente. Die Spielfigur geht in einen dunklen Raum. Sie schaltet das Licht an. Das Licht ist rot. Später kann die Spielfigur die Glühbirne gegen eine andere Farbe austauschen. Für solche Effekte sind Shader ideal.

GameMaker Filter und Effekte

„Ja aber… GMS hat doch mittlerweile Filter und Effekte, die genau das tun? Warum sollte mich das interessieren?“

Oh, stimmt eigentlich. Das Tutorial ist hiermit zu Ende.

Natürlich nicht. Es ist richtig, dass es in GameMaker Studio mittlerweile tolle Post-Processing-Effekte gibt. Diese sind mehr als nur eine nette Spielerei. Man kann einige Parameter verstellen und sogar auf verschiedenen Ebenen diverse Effekte anwenden. So kann man etwa auf dem Hintergrund einen Wassereffekt anzeigen, während im Vordergrund etwas ganz anderes passiert.



In GameMaker Studio gibt es mittlerweile umfangreiche Einstellungen für Filter und Effekte

Das Problem ist hier, dass man nur das nehmen kann, was angeboten wird und man es nur so verstellen darf, wie vorgegeben. Es fehlt also die Kontrolle. Und auch wenn die Liste dieser Effekte immer länger wird, kann man nicht erwarten, dass irgendwann genau der Shader erscheint, den man sich wünscht.

Bevor wir nun eigene Shader-Effekte umsetzen, muss ich noch zwei Themen ansprechen.

Die Kamera

Nur in den wenigsten Spielen basiert das ganze Level oder die ganze Spielwelt auf einen Raum, der

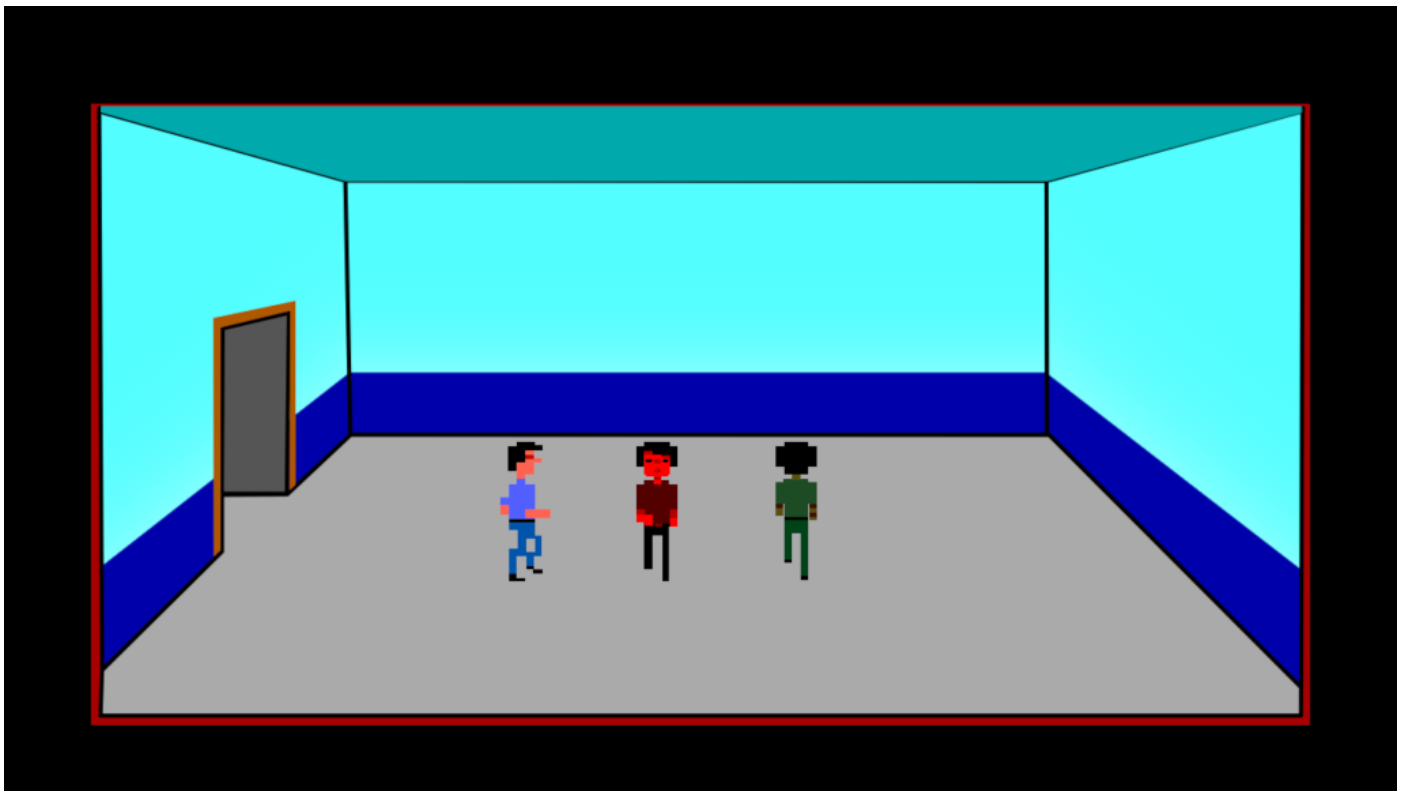
lediglich so groß ist wie der Bildschirm. Dies trifft auf einfache Platformer, viele Rätselspiele und alte Adventure zu. Meistens jedoch scrollt der Bildschirm, etwa bei Shootern, Autorennen, Platformern, Beat ‚em up u. s. w.

Hierfür braucht man eine Kamera, die dem Spieler folgt. Zu dem Thema gibt es sehr viele gute Tutorials, etwa dieses hier: CAMERAS AND VIEWS

Oberflächen

Noch wesentlich wichtiger als das Kamera-Thema sind – bezogen auf Shader und Post-Processing – Oberflächen. Das englische Wort dafür sind „Surfaces“. Auch das ist ein Thema für ein großes Tutorial, aber ich werde versuchen, die wesentlichen Fakten kompakt zu vermitteln.

Im Prinzip basiert alles im GameMaker auf Oberflächen. Normalerweise bekommen wir das aber nicht mit. Die globale Variable lautet `application_surface`. Auf dieser Fläche wird das ganze Spiel gezeichnet. Wir können die `application_surface` aber manipulieren oder eigene Oberflächen erschaffen, um darauf zu zeichnen.



Set 1 Ausgangsposition, bevor wir Post-Processing Shader einsetzen. Man erkennt, dass zwei der drei Figuren noch die Shader aus dem ersten Tutorial besitzen

Das ist sehr praktisch, gerade bei Post-Processing-Effekten. Wir erzeugen eine neue Oberfläche und manipulieren sie nach belieben. Allerdings gibt es ein paar Dinge zu beachten. So muss man wissen, dass Oberflächen flüchtig sind. Das bedeutet, dass sie aus dem Speicher verschwinden, wenn das Fenster geschlossen und anschließend wieder geöffnet wird.

Es gibt einige praktische Anwendungen für Oberflächen. Am häufigsten werden sie verwendet, um

bestimmte Effekte zu erzeugen, bei denen der Spieler oft nicht einmal merkt, dass es ein Effekt ist. Zum Beispiel kann man darauf Blut oder tote Feinde zeichnen, indem man nur das Sprite ohne Objekt nutzt. Das hat den Vorteil, dass man viel Speicher spart, da Objekte zahlreiche Variablen besitzen, die meistens nicht gebraucht werden.

Set 2 Ausgangsposition, ebenfalls mit drei Figuren und den entsprechenden Shadern

Der Nachteil an Oberflächen ist, dass sie Videospeicher verbrauchen. Wenn man sorglos damit umgeht oder schlecht programmiert, kann es zu einem Speicherüberlauf kommen, wodurch der Computer abstürzen kann.

Draw-GUI für Post-Processing

Wie eingangs beschrieben, wird bei Post-Processing-Effekten alles manipuliert. Aber was bedeutet „alles“? Nehmen wir an, wir hätten einen riesigen Raum von 300.000 x 300.000 Pixeln. Würden wir alles manipulieren, hätten wir sehr schnell ein Problem mit der Performance und/oder dem Videospeicher.

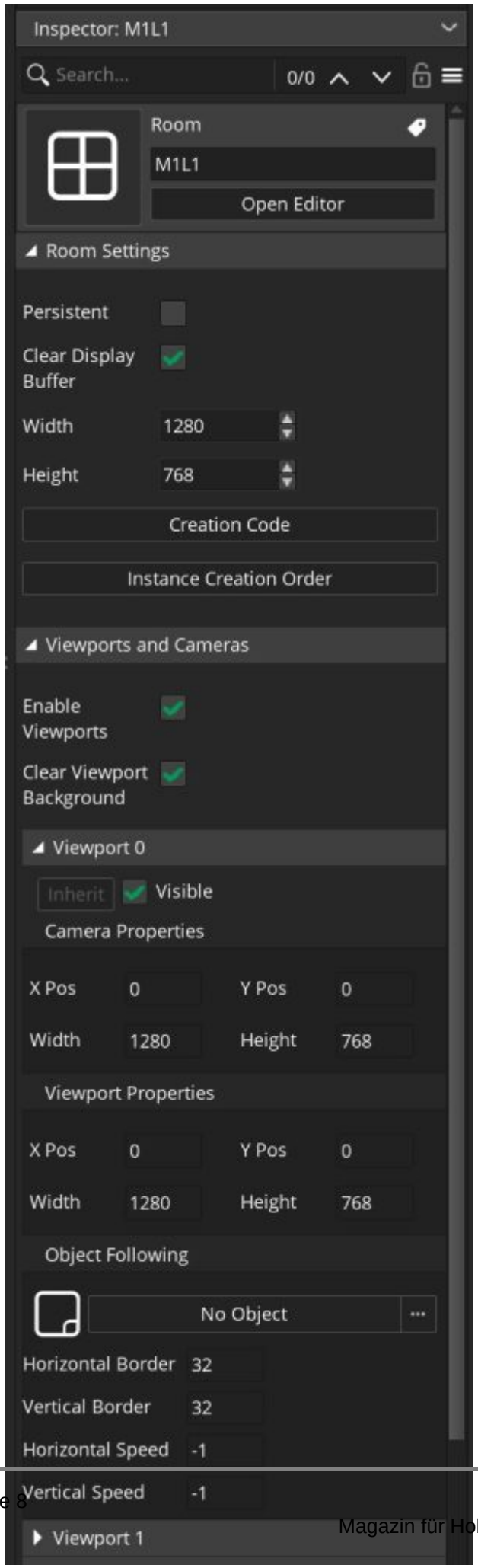
Zum Glück gibt es neben dem *Draw*- auch das *Draw-GUI-Event*. Das beschränkt sich auf die Bildschirmgröße. Wenn unser Spiel eine Auflösung von 1920×1080 Pixeln hat, dann wird im Draw-GUI-Event nur dieser Bereich berechnet, gezeichnet und angezeigt, unabhängig von der Raumgröße. Das ist sehr praktisch, schließlich behandelt Draw-GUI damit genau die Fläche, die wir für Post-Processing brauchen. Der Rest kann ignoriert werden.

Workflow

Das waren nun viele Informationen. Bevor wir nun endlich mit dem Code loslegen, möchte ich das Vorgehen zusammenfassen.

Zunächst brauchen wir natürlich einen Raum. Wenn wir eine Figur haben, dem der Betrachter folgen sollte, müssen wir erst eine Kamera mit entsprechender View einrichten.

Für die Post-Processing-Shader verwenden wir eigene Objekte. Um den Überblick zu bewahren, hat es sich bewährt, einen eigenen Layer mit der Bezeichnung *Shader* im Raum anzulegen. Außerdem stellen wir bei den Raumeinstellungen einen View ein.


Inspector: M1L1
Search...
0/0
Room
M1L1
Open Editor
Room Settings
Persistent
Clear Display Buffer
Width
1280
Height
768
Creation Code
Instance Creation Order
Viewports and Cameras
Enable Viewports
Clear Viewport Background
Viewport 0
Inherit
Visible
Camera Properties
X Pos
0
Y Pos
0
Width
1280
Height
768
Viewport Properties
X Pos
0
Y Pos
0
Width
1280
Height
768
Object Following
No Object
Horizontal Border
32
Vertical Border
32
Horizontal Speed
-1
Vertical Speed
-1
Viewport 1

GameMaker Studio 2

Bei Post-Processing-Effekten arbeiten wir mit dem Draw-GUI-Event. Außerdem erzeugen wir eine eigene Oberfläche, welche der Bildschirmgröße bzw. dem View-Port entspricht.

Es ist sowohl möglich, für jeden Shader ein Objekt zu erzeugen, oder ein Objekt, in dem sich alle Post-Processing-Shader befinden. In diesem Fall kann bspw. auf Tastendruck umgeschaltet werden. Beide Vorgehensweisen haben Vor- und Nachteile.

Praxistipp – Umgang mit Variablen

Wenn man einen Shader programmiert, möchte man häufig Variablen verwenden, um bspw. die perfekten Werte zu finden. In der Praxis hat es sich bewehrt, entsprechende Tasten einzurichten, mit denen man die Werte im Spiel verstellen kann. Das geht wesentlich schneller als Wert zu definieren, Spiel kompilieren und starten, Spiel abbrechen, Wert verändern u. s. w.

Wenn man nur einen Wert hat, finde ich das Mausrad sehr praktisch. Indem man hoch und runter dreht, verschiebt man den Wert. Natürlich ist es dabei sinnvoll, den entsprechenden Wert auf dem Bildschirm anzuzeigen.

Sobald man die perfekten Werte gefunden hat, kann man sie fest eingeben und die Steuerung des Shaders auskommentieren.

Vorsichtsmaßnahmen

Es gibt vor allem zwei Abfragen, die wichtig sind, bevor man einen entsprechenden Shader einsetzt:

1. Existiert die Oberfläche?
2. Wurde der Shader kompiliert?

Wie bereits erwähnt, sind Oberflächen flüchtig. Deshalb müssen wir im Draw-GUI-Event als erstes fragen, ob das Surface existiert. Wenn nicht, legen wir es neu an.

```
if (!surface_exists(surf))
{
 surf = surface_create(resX, resY);
}
```

Mit `surface_exists()` prüfen wir, ob die angegebene Oberfläche `surf` existiert. Mit dem Ausrufezeichen drehen wir die Frage um, was so viel bedeutet wie: *Wenn die Oberfläche `surf` nicht existiert, dann…*

Mit `surface_create()` legen wir in dem Fall die Oberfläche neu an. Das sollte sich ganz oben im Draw-GUI-Event befinden, damit wir immer sicherstellen, dass wir auf `surf` zeichnen können. Zu `resX` und `resY` komme ich später.

Ob der Shader kompiliert wurde, habe ich im ersten Tutorial bewusst nicht abgefragt, ist aber sinnvoll. Dies können wir mit `shader_is_compiled()` tun. Beispiele dafür sehen wir nachher noch zur

Genüge.

Shader-Objekt

Für alle nachfolgenden Shader ist der Aufbau des Objekts identisch. Unterschiede gibt es lediglich beim Namen des Shaders und die verwendeten Variablen. Das bedeutet, dass ich hier nur einmal den ganzen Aufbau erkläre. Bei den weiteren Beispielen gehe ich nur auf die Unterschiede ein.

Create-Event

```
resX = view_wport[0];
resY = view_hport[0];

surf = -1;
```

Wie oben im Screenshot gezeigt, nutzen wir *View-Port 0*. Entsprechend lesen wir in den ersten beiden Zeilen die horizontale und vertikale Auflösung aus.

In der letzten Zeile legen wir die Oberfläche mit *-1* an. Der Rest geschieht im Draw-GUI-Event.

Draw GUI

```
if (!surface_exists(surf))
{
 surf = surface_create(resX, resY);
}

view_set_surface_id(0, surf);

if (shader_is_compiled(shader_name))
{
 shader_set(shader_name);
  draw_surface(surf, 0, 0);
 shader_reset();
}
```

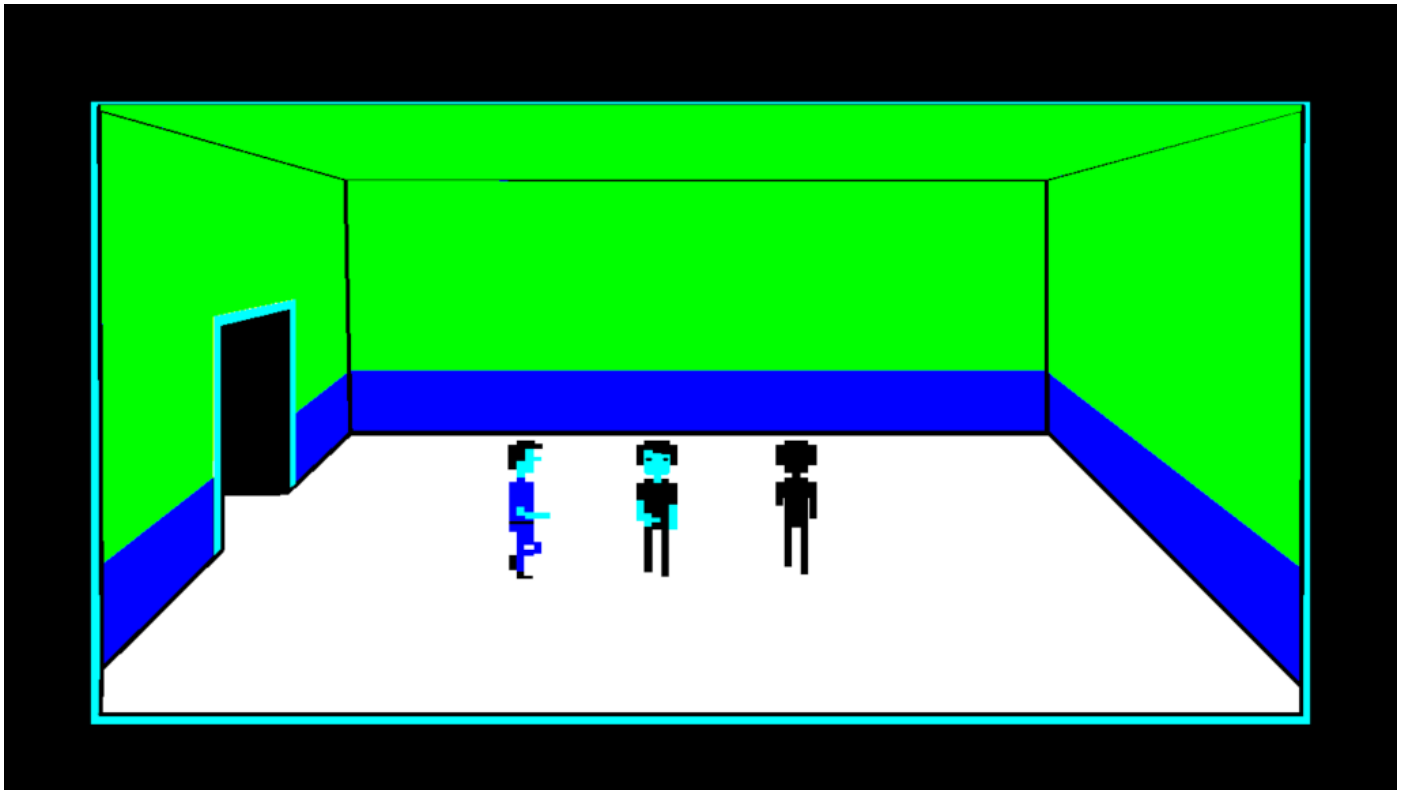
Die ersten Zeilen kennen wir bereits. Mit `view_set_surface_id()` legen wir den Inhalt fest, auf das eine Oberfläche gezeichnet werden soll. Dabei ist *0* unser View.

Anschließend fragen wir ab, ob der Shader kompiliert wurde. Wenn ja, setzen wir, wie im ersten Tutorial gezeigt, den Shader, zeichnen die Oberfläche mit `draw_surface(surface, x, y)` und beenden den Shader mit `shader_reset()`. Im Prinzip ist das also relativ einfach, weshalb wir nun endlich zu den eigentlichen Shadern kommen. Hierfür habe ich ein paar vorbereitet.

CGA-Shader

CGA (**C**olor **G**raphics **A**dapter) wurde 1981 eingeführt und war die erste Farbgrafikkarte von IBM für den IBM PC, die einen De-facto-Standard für Computerbildschirme etablierte. Je nach Auflösung können 4 bis 16 Farben angezeigt werden. Im folgenden Beispiel habe ich mich auf acht Farben

beschränkt, ihr könnt es aber gerne erweitern oder auf vier Farben reduzieren.



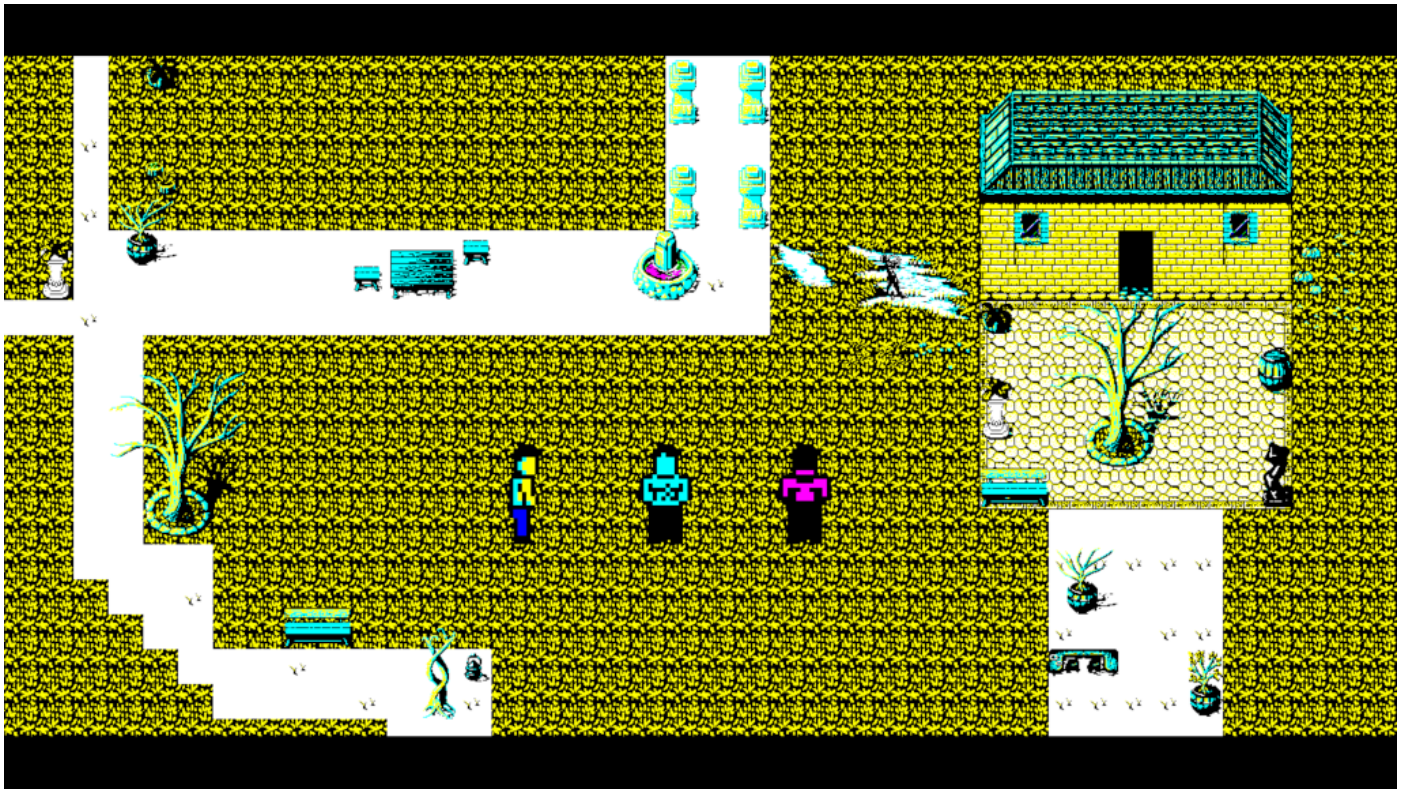
Set 1 mit dem CGA-Shader

Zur Erinnerung: Bei den Post-Processing-Effekten arbeiten wir nur mit dem Fragment-Shader. Hier der Code:

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

void main()
{
    vec4 col = texture2D(gm_BaseTexture, v_vTexcoord);

    // Konvertiere die Farbe in CGA-Farben
    vec4 cga_col;
    if (col.r < 0.5 && col.g < 0.5 && col.b < 0.5) {
        cga_col = vec4(0.0, 0.0, 0.0, 1.0); // Schwarz
    } else if (col.r >= 0.5 && col.g < 0.5 && col.b < 0.5) {
        cga_col = vec4(0.0, 1.0, 1.0, 1.0); // Cyan
    } else if (col.r < 0.5 && col.g >= 0.5 && col.b < 0.5) {
        cga_col = vec4(1.0, 0.0, 1.0, 1.0); // Magenta
    } else if (col.r >= 0.5 && col.g >= 0.5 && col.b < 0.5) {
        cga_col = vec4(1.0, 1.0, 0.0, 1.0); // Gelb
    } else if (col.r < 0.5 && col.g < 0.5 && col.b >= 0.5) {
        cga_col = vec4(0.0, 0.0, 1.0, 1.0); // Dunkelblau
    } else if (col.r >= 0.5 && col.g < 0.5 && col.b >= 0.5) {
        cga_col = vec4(1.0, 0.0, 1.0, 1.0); // Hellblau
    } else if (col.r < 0.5 && col.g >= 0.5 && col.b >= 0.5) {
        cga_col = vec4(0.0, 1.0, 0.0, 1.0); // Grün
    } else if (col.r >= 0.5 && col.g >= 0.5 && col.b >= 0.5) {
```

```
        cga_col = vec4(1.0, 1.0, 1.0, 1.0); // Weiß
    }

    gl_FragColor = cga_col;
}
```

Dieser Shader konvertiert jede Farbe in eine der acht Farben, basierend auf der Farbzusammensetzung von Rot, Grün und Blau. Wenn die Farbe nicht genau einer der *CGA*-Farben entspricht, wird sie auf die am nächsten liegende CGA-Farbe abgebildet.



Set 2 mit dem CGA-Shader. Man erkennt einen extremen Unterschied zu Set 1. Das ist eines der Gründe, warum solche Shader individuell angepasst werden sollten

Sowohl Create- wie auch Draw-GUI-Event entsprechen der o. g. Vorlage. Ihr müsst lediglich den Namen des Shaders austauschen.

Pixel-Shader

Wir wollen den Bildschirm verpixelt anzeigen? Kein Problem.

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;
uniform vec2 resolution;
uniform float pixelSize;

void main()
{
    vec2 uv = gl_FragCoord.xy / resolution.xy;  // Normierte Texturkoordinaten
    vec2 pixelSizeUV = vec2(pixelSize) / resolution.xy; // Normierte Pixelgrö
```

```
    vec2 roundedUV = floor(uv / pixelSizeUV) * pixelSizeUV * 0.5;  // Berechnu

    vec4 col = texture2D(gm_BaseTexture, roundedUV); // Textur-Farbwert für di
    gl_FragColor = col; // Ausgabe der Texturfarbe für das aktuelle Fragment-P
}
```

Hier werden in `main()` die normierten Texturkoordinaten `uv` für das aktuelle Fragment-Pixel sowie die normierte Pixelgröße `pixelSizeUV` berechnet. Dann werden die gerundeten Texturkoordinaten `roundedUV` für das Retro-Pixel-Raster ermittelt. Schließlich wird der Textur-Farbwert `col` für die gerundeten Texturkoordinaten berechnet und als Farbausgabe für das aktuelle Fragment-Pixel verwendet.

https://www.bytegame.de/wp-content/uploads/2023/06/Pixelshader-Set-1.mp4

Pixel-Shader von Set 1. Irgendwann erkennt man nichts mehr.

Da wir die Variablen `pixelSize` und `resolution` haben, müssen wir unsere Events noch anpassen.

Create-Event

```
uni_pixelate = shader_get_uniform(sh_pixelate, „pixelSize");
uni_resolution = shader_get_uniform(sh_pixelate, „resolution");
val_pixelate = 8;
```

Wir beginnen mit einem Wert von *8*. Wie bereits beschrieben, kann man den perfekten Wert bspw. per Mausrad ermitteln.

https://www.bytegame.de/wp-content/uploads/2023/06/Pixelshader-Set-2.mp4

Beim zweiten Set ist die Grafik ohnehin schon sehr feingranular

Draw GUI

```
if (shader_is_compiled(sh_pixelate))
{
 shader_set(sh_pixelate);
  shader_set_uniform_f(uni_pixelate, val_pixelate);
  shader_set_uniform_f(uni_resolution,resX, resY);
  draw_surface(surf, 0, 0);
 shader_reset();
}
```

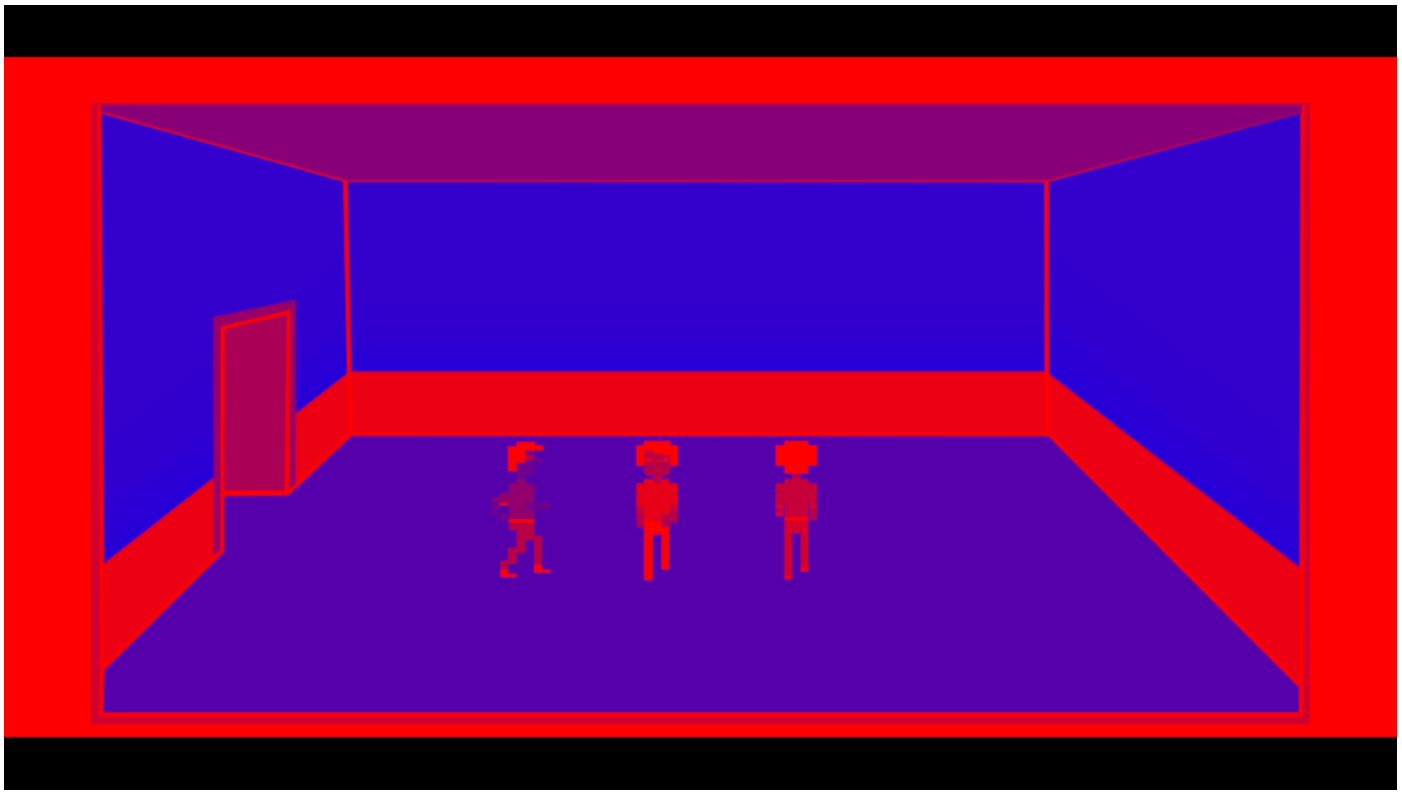
Infrarot-Shader

Wäre es nicht cool, wenn man alles im Infrarot-Wärmebild-Look sehen könnte? Hier ein simples Beispiel dafür:

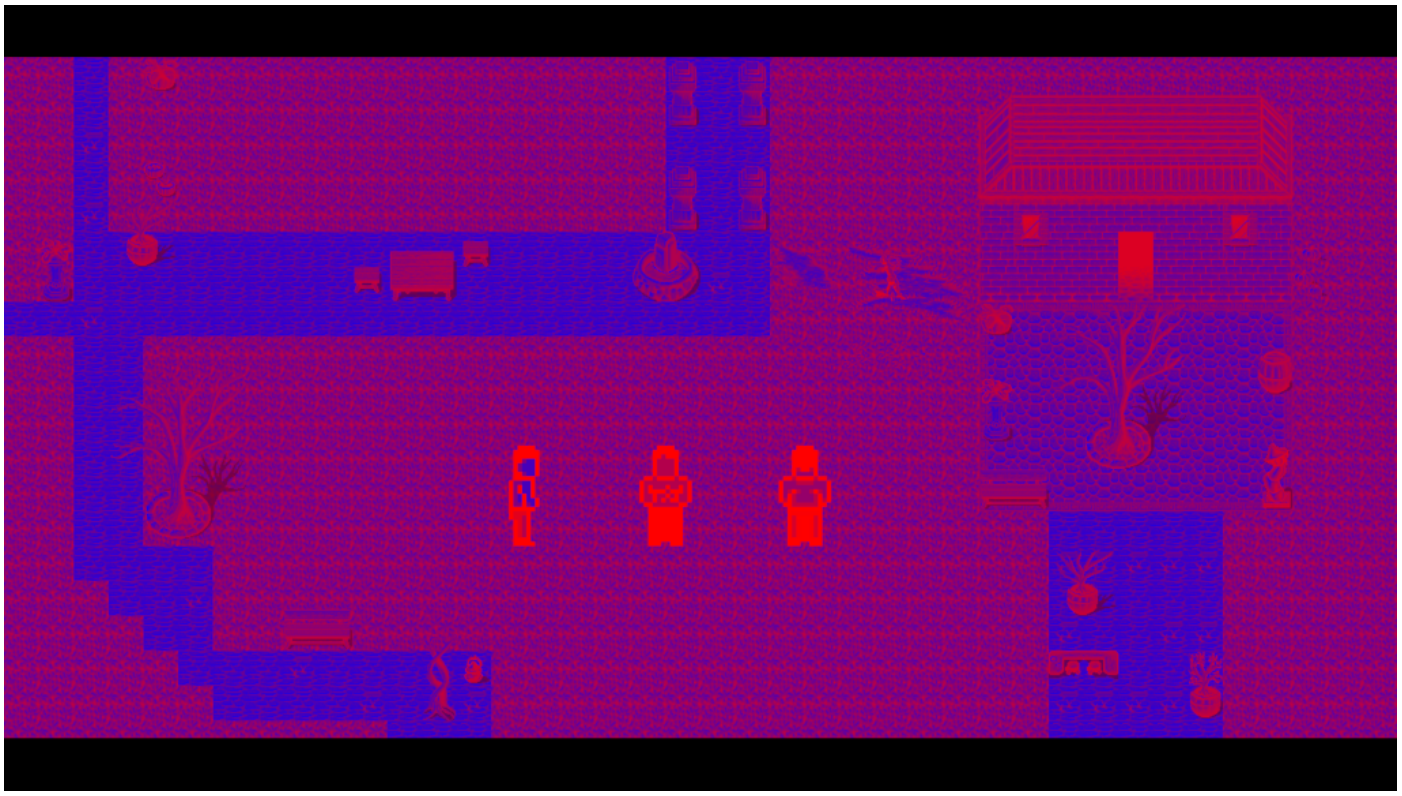
```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;
```

```
void main()
{
    // Samplen des Farbwertes
    vec4 color = texture2D(gm_BaseTexture, v_vTexcoord);

    // Extrahieren der Luminanz-Komponente
    float luminance = dot(color.rgb, vec3(0.299, 0.587, 0.114));

    // Infrarot-Transformation
    vec3 infraredColor = vec3(1.0 - luminance, 0.0, luminance);

    // Anzeigen des Ergebnisses
    gl_FragColor = vec4(infraredColor, color.a);
}
```

Da wir hier keine weiteren Variablen brauchen, ist der Rest wie im Ausgangsbeispiel.



Infrarot-Shader Set 1



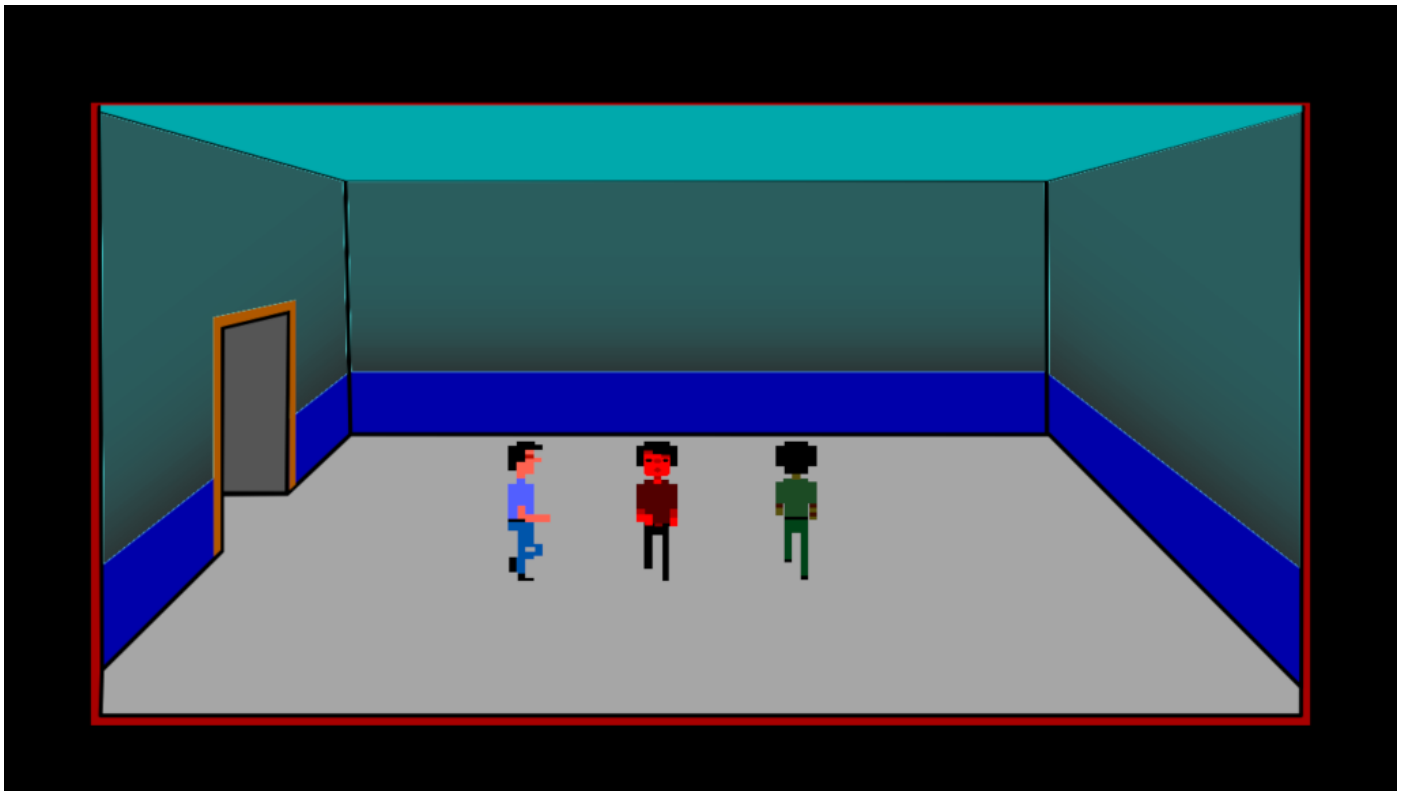
Infrarot-Shader Set 2

Neon-Shader

Eine ähnliche Veränderung können wir mit einem Neon-Effekt erzeugen. Wie gut und wie intensiv der Shader wirkt, hängt sehr stark vom Ausgangsbild ab. Um den Shader besser steuern zu können, haben wir gleich drei Variablen.

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

uniform float threshold;
uniform float outlineWidth;
uniform float outlineBrightness;

void main()
{
    vec4 col = texture2D(gm_BaseTexture, v_vTexcoord); // lädt die Farbe des a
    vec3 luminance = vec3(0.299, 0.587, 0.114); // definiert einen Vektor mit
    float intensity = dot(col.rgb, luminance); // berechnet die Helligkeit des

    float alpha = smoothstep(threshold, threshold + outlineWidth, intensity);
    vec4 outlineCol = vec4(outlineBrightness * vec3(1.0), alpha); // definiert

    gl_FragColor = mix(col, outlineCol, alpha);
}
```

Dieser Shader hat eine Funktion, welche die Konturen neu zeichnet. Er erhält Informationen über das Bild, indem er das Texturbild und die Farbe jedes Pixels analysiert.



Der Neon-Shader wirkt bei Set 1 relativ gut. Er war auch darauf abgestimmt

Da wir drei Variablen haben, müssen wir unsere Events entsprechend anpassen.

Create-Event

```
uni_threshold   = shader_get_uniform(sh_neon, „threshold");
uni_outlineWidth  = shader_get_uniform(sh_neon, „outlineWidth");
uni_outlineBrightness = shader_get_uniform(sh_neon, „outlineBrightness");
val_threshold   = 0.65;
val_outlineWidt   = 0.25;
val_outlineBrightness = 0.15;
```

Draw GUI

```
if (shader_is_compiled(sh_neon))
{
 shader_set(sh_neon);
  shader_set_uniform_f(uni_threshold, val_threshold);
  shader_set_uniform_f(uni_outlineWidth, val_outlineWidt);
  shader_set_uniform_f(uni_outlineBrightness, val_outlineBrightness);
  draw_surface(surf, 0, 0);
 shader_reset();
}
```

Beim zweiten Set kommt der Neon-Shader nur bei den Steinboden-Tiles und der linken Figur zur Geltung

Sättigungs-Shader

Um die Farbsättigung zu verringern, gibt es viele verschiedene Möglichkeiten. Die hier gezeigte ist etwas komplex, dafür aber sehr genau. Es ist ein schönes Beispiel dafür, dass man im Shader auch mehrere Funktionen anwenden kann. Um den Effekt zu steuern, brauchen wir nur eine Variable, mit der wir den Grad der Sättigung steuern.

https://www.bytegame.de/wp-content/uploads/2023/06/Saettigungsshader-Set-1.mp4

Wir verringern die Farbsättigung in Echtzeit

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;
uniform float saturation;

// Funktion zur Umwandlung einer RGB-Farbe in eine HSV-Farbe
vec3 rgb2hsv(vec3 c)
{
    // Vektor mit Konstanten, der in der Formel zur Umrechnung von RGB nach HS
    vec4 K = vec4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    // Berechnung von Hilfsvariablen
    vec4 p = mix(vec4(c.bg, K.wz), vec4(c.gb, K.xy), step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r), vec4(c.r, p.yzx), step(p.x, c.r));

    // Berechnung der Komponenten von HSV
    float d = q.x - min(q.w, q.y);
```

```
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}

// Funktion zur Umwandlung einer HSV-Farbe in eine RGB-Farbe
vec3 hsv2rgb(vec3 c)
{
    // Vektor mit Konstanten, der in der Formel zur Umrechnung von HSV nach RG
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    // Berechnung von Hilfsvariablen
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    // Rückgabe der RGB-Farbe
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main()
{
    // Lesen der Farbe des aktuellen Pixels aus der Textur
    vec4 texcolor = texture2D(gm_BaseTexture, v_vTexcoord);
    // Extrahieren der RGB-Farbwerte aus der gelesenen Farbe
    vec3 color = texcolor.rgb;

    // Konvertierung von RGB nach HSV
    vec3 hsv = rgb2hsv(color);
    // Ändern der Sättigung basierend auf der Uniform-Variable
    hsv.y *= saturation;
    // Konvertierung von HSV zurück nach RGB
    color = hsv2rgb(hsv);

    // Setzen der Farbe des aktuellen Pixels auf die neue RGB-Farbe mit der ur
    gl_FragColor = vec4(color, texcolor.a);
}
```

Create-Event

```
uni_saturation = shader_get_uniform(sh_saturation, „saturation");
val_saturation = 0.2;
```

Draw GUI

```
if (shader_is_compiled(sh_saturation))
{
 shader_set(sh_saturation);
  shader_set_uniform_f(uni_saturation, val_saturation);
  draw_surface(surf, 0, 0);
 shader_reset();
}
```

Radialer-Weichzeichner

Es gibt sehr viele verschiedene Methoden für Weichzeichner. Diese hier zeigt eine Art radialen Weichzeichner, für den es wiederum zahlreiche Algorithmen gibt.

https://www.bytegame.de/wp-content/uploads/2023/06/Radialer-Weichzeichner-Set-1.mp4

Der radiale Weichzeichner ließe sich gut für Übergänge einsetzen. Oder wenn die Spielfigur einen Schaden erleidet

```
varying vec2 v_vTexcoord;
uniform vec2 resolution;
uniform float blurStrength;

void main()
{
    vec2 uv = v_vTexcoord.xy;
    vec4 sum = vec4(0.0);
    vec2 texcoord = v_vTexcoord.xy * resolution.xy;

    int num_samples = 16;

    for (int i = 0; i < num_samples; i++)
    {
        float angle = 3.14159265359 * 2.0 * float(i) / float(num_samples);
        float dist = blurStrength * float(i) / float(num_samples);
        vec2 offset = vec2(dist * cos(angle), dist * sin(angle));

        sum += texture2D(gm_BaseTexture, (texcoord + offset) / resolution.xy);
    }

    gl_FragColor = sum / float(num_samples);
}
```

Create-Event

```
uni_blurStrength = shader_get_uniform(sh_radial_blur, „blurStrength");
uni_resolution = shader_get_uniform(sh_radial_blur, „resolution");
val_blurStrength = 10;
```

Die Variable `val_blurStrength` erscheint mit einem Wert von 10 als recht hoch. Tatsächlich ist es aber so, dass man es sehr fein einstellen kann. In meinem Beispiel ließ ich Werte zwischen *0* und *500* zu.

Draw GUI

```
if (shader_is_compiled(sh_radial_blur))
{
 shader_set(sh_radial_blur);
  shader_set_uniform_f(uni_blurStrength, val_blurStrength);
  shader_set_uniform_f(uni_resolution,resX, resY);
  draw_surface(surf, 0, 0);
 shader_reset();
```

```
}
```

Fazit und Ausblick

Die zahlreichen Beispiele sollen nicht nur das Gelernte verinnerlichen, sondern auch zeigen, welche Möglichkeiten es gibt. Natürlich kann man alle gezeigten Shader erweitern, verändern oder als Vorlage für ganz andere Effekte nutzen.

Während wir uns im ersten Teil um Grundlagen und einfache Dinge gekümmert haben, war dieser Teil deutlich anspruchsvoller. Doch mit Shadern lassen sich auch beeindruckende bildschirmfüllende Effekte zaubern. Wie das funktioniert, schauen wir uns im dritten Teil an.

Verwendete Sprites

Charakter
Hintergrund-Tile

Weiterführende Links

Shader-Programmierung 1: Grundlagen und Sprites
Shader-Programmierung 3: Effekte
Shader-Effekt: Warping
Textscroller: Wellen und einzelne Farben
Raster bar Effekt
Sonnenblumen und der goldene Schnitt

Date Created
9. Juni 2023
Author
sven