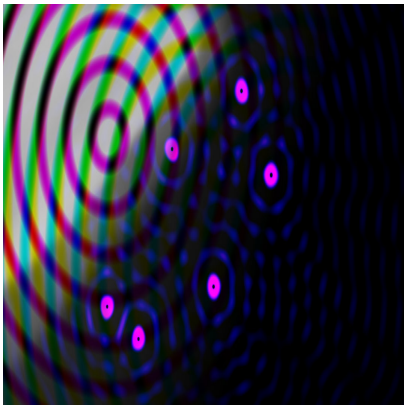

Shader-Effekt: Interferenz-Effekt

Description

Es gibt einen ziemlich alten, aber immer noch sehr hübschen Effekt aus der Demoszene (vor allem PC und Amiga): Den Interferenz-Effekt. In diesem Tutorial möchte ich zwei Wege zeigen, wie man das über Shader im GameMaker realisieren kann.

Was ist ein Interferenz-Effekt?

Die Interferenz ist ein physikalisches Phänomen, das auftritt, wenn zwei oder mehr Wellen aufeinandertreffen und sich überlagern. Dieses Phänomen ist in vielen Bereichen der Physik und Optik von großer Bedeutung.

Die Grundidee der Interferenz besteht darin, dass, wenn zwei Wellen aufeinandertreffen, die Amplituden (Höhen) der Wellen sich addieren oder subtrahieren können, abhängig von der Phasenbeziehung zwischen den Wellen. Wenn die Wellen in Phase sind (ihre Spitzen und Täler treffen zur gleichen Zeit ein), addieren sich die Amplituden, und es entsteht ein konstruktiver Interferenzeffekt, der zu einer Verstärkung der Welle führt. Dies führt zu hellen Bereichen in einem Interferenzmuster.

Wenn die Wellen gegenläufig (in entgegengesetzten Phasen) sind, subtrahieren sich die Amplituden, und es entsteht ein destruktiver Interferenzeffekt, der zu einer Abschwächung der Welle führt. Dies führt zu dunklen Bereichen in einem Interferenzmuster.

Das Objekt

Wie immer bei meinen GML-Shader-Tutorials beginnen wir mit einem entsprechenden Objekt. Der Code ist für beide Effekte identisch, wir müssen nur im Draw-Event den Namen des Shaders anpassen. Alternativ könnt ihr gleich eine entsprechende Variable im Create-Event definieren und im Draw-Event verwenden.

Create-Event

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

Draw-Event

```
time += time_add;

shader_set(sh_interference01);
shader_set_uniform_f(shader_get_uniform(sh_interference01,"resolution"), displ
shader_set_uniform_f(shader_get_uniform(sh_interference01,"time"),time);
draw_surface_ext(application_surface, 0, 0, 1, 1, 0, c_white, 0);
shader_reset();
```

Der Shader heißt dann später *sh_interference01* bzw. *sh_interference02*.

Der einfache Effekt sh_interference01

Hierfür brauchen wir nur den Fragment-Shader, den Code habe ich recht ausführlich kommentiert.

```
// Definiere Konstanten für die Welleneigenschaften
#define WAVELENGTH 0.4    // Wellenlänge
#define WAVESPEED 8.0    // Wellengeschwindigkeit

// Uniform-Variablen, die von GameMaker bereitgestellt werden
uniform float time;        // Zeit
uniform vec2 resolution;  // Bildschirmauflösung

// Funktion zur Berechnung der Wellenhöhe
float wave_height(vec2 p, vec2 c)
{
    // Berechne den Abstand zwischen dem Punkt 'p' und dem Zentrum 'c'
    float d = distance(p, c) / 96.0;

    // Berechne die Wellenhöhe basierend auf der Sinuswelle
    float waveValue = (1.0 - sin((d - WAVESPEED * time) / WAVELENGTH));

    return waveValue;
}

// Haupt-Fragmentshader-Funktion
void main(void)
{
    vec2 uv = gl_FragCoord.xy;

    // Verschiebe die Position von uv um die Bildschirmmitte
    vec2 centeredUV = uv - vec2(resolution.x / 2.0, resolution.y / 2.0);

    // Berechne die Wellenhöhe an verschiedenen Positionen
    float upperWave = wave_height(centeredUV, vec2(-resolution.x / 4.0, 0.0));
    float middleWave = wave_height(centeredUV, vec2(0.0, 0.0));
    float lowerWave = wave_height(centeredUV, vec2(resolution.x / 4.0, 0.0));

    // Verwende die Wellenhöhe, um die Farben zu modulieren
```

```
    vec3 upperColor = vec3(1.0, 0.0, 0.0);   // Rotes Fragment für die linke L
    vec3 middleColor = vec3(0.0, 0.0, 1.0);   // Blaues Fragment für die mittl
    vec3 lowerColor = vec3(0.0, 1.0, 0.0);   // Grünes Fragment für die rechte

    // Kombiniere die Farben basierend auf der Wellenhöhe
    vec3 finalColor = upperColor * upperWave + middleColor * middleWave + lowe

    // Setze die Farbe des Fragments mit der berechneten Farbe
    gl_FragColor = vec4(finalColor, 1.0);  // Alpha-Kanal (vollständig sichtba
}
```

So sieht das Resultat aus:

https://www.bytegame.de/wp-content/uploads/2023/11/sh_interference01.mp4

Erklärung

1. Zuerst werden die Konstanten für die Welleneigenschaften definiert:
    - `WAVELENGTH` repräsentiert die Wellenlänge der Sinuswelle.
    - `WAVESPEED` gibt die Geschwindigkeit an, mit der die Welle sich bewegt.
2. Dann kommen die Uniform-Variablen, die von GameMaker bereitgestellt werden:
    - `time` repräsentiert die Zeit, die im Shader verwendet wird.
    - `resolution` gibt die Bildschirmauflösung an.
3. Die Funktion `wave_height(vec2 p, vec2 c)` berechnet die Wellenhöhe an einem Punkt `p` basierend auf einem Zentrum `c`:
    - Zuerst wird der Abstand `d` zwischen `p` und `c` berechnet und durch 96.0 geteilt.
    - Dann wird die Wellenhöhe `waveValue` basierend auf einer Sinuswelle berechnet. Hierbei wird `d` mit der Zeit `time`, der Wellenlänge `WAVELENGTH` und der Geschwindigkeit `WAVESPEED` verwendet.
    - Das Ergebnis wird zurückgegeben.
4. Die Haupt-Fragmentshader-Funktion `main(void)` wird aufgerufen:
    - `vec2 uv` repräsentiert die Bildschirmkoordinaten des aktuellen Fragments.
    - `vec2 centeredUV` verschiebt die Koordinaten `uv` um die Bildschirmmitte, um die Position relativ zur Mitte zu berechnen.
5. Nun wird die Wellenhöhe an verschiedenen Positionen berechnet:
    - `upperWave` berechnet die Wellenhöhe für die linke Lichtquelle, die bei 25% der Bildschirmbreite (negative x-Koordinate) platziert ist.
    - `middleWave` berechnet die Wellenhöhe für die mittlere Lichtquelle, die in der Mitte (50% der Bildschirmbreite) platziert ist.
    - `lowerWave` berechnet die Wellenhöhe für die rechte Lichtquelle, die bei 75% der

Bildschirmbreite (positive x-Koordinate) platziert ist.

6. Wir weisen Farben den Lichtquellen zu:
   - `upperColor` ist rot und repräsentiert die linke Lichtquelle.
   - `middleColor` ist blau und repräsentiert die mittlere Lichtquelle.
   - `lowerColor` ist grün und repräsentiert die rechte Lichtquelle.
7. Jetzt kombinieren wir die Farben basierend auf der Wellenhöhe:
   - `finalColor` wird berechnet, indem die Farben der Lichtquellen (`upperColor`, `middleColor`, `lowerColor`) mit den jeweiligen Wellenhöhen (`upperWave`, `middleWave`, `lowerWave`) multipliziert und addiert werden.
8. Die endgültige Farbe `finalColor` wird als die Farbe des aktuellen Fragments gesetzt, wobei der Alpha-Kanal auf 1.0 (vollständig sichtbar) gesetzt wird.

Modifikation

Man kann den Effekt noch etwas modifizieren und die beiden äußeren Kreise um die Mitte kreisen lassen.

https://www.bytegame.de/wp-content/uploads/2023/11/sh_interference01b.mp4

```
#define WAVELENGTH 0.4
#define WAVESPEED 8.0
#define ROTATIONSPEED 2.0 // Drehgeschwindigkeit der äußeren Kreise

uniform float time;
uniform vec2 resolution;

float wave_height(vec2 p, vec2 c)
{
    float d = distance(p, c) / 96.0;
    float waveValue = (1.0 - sin((d - WAVESPEED * time) / WAVELENGTH));
    return waveValue;
}

void main(void)
{
    vec2 uv = gl_FragCoord.xy;
    vec2 centeredUV = uv - vec2(resolution.x / 2.0, resolution.y / 2.0);

    // Berechne die Wellenhöhe an verschiedenen Positionen
    float upperWave = wave_height(centeredUV, vec2(-resolution.x / 4.0, 0.0));
    float middleWave = wave_height(centeredUV, vec2(0.0, 0.0));
    float lowerWave = wave_height(centeredUV, vec2(resolution.x / 4.0, 0.0));

    // Verwende die Wellenhöhe, um die Farben zu modulieren
    vec3 upperColor = vec3(1.0, 0.0, 0.0);   // Rotes Fragment für die linke L
    vec3 middleColor = vec3(0.0, 0.0, 1.0);   // Blaues Fragment für die mittl
```

```
    vec3 lowerColor = vec3(0.0, 1.0, 0.0);   // Grünes Fragment für die rechte

    // Drehung der äußeren Lichtquellen um die Mitte
    float angle = time * ROTATIONSPEED;  // Berechne den Rotationswinkel basie
    mat2 rotationMatrix = mat2(cos(angle), -sin(angle), sin(angle), cos(angle)

    // Verschiebe die Position der äußeren Lichtquellen um die Mitte
    vec2 leftPosition = vec2(-resolution.x / 4.0, 0.0);
    vec2 rightPosition = vec2(resolution.x / 4.0, 0.0);

    // Berechne die Wellenhöhe an den gedrehten Positionen
    vec2 rotatedLeftPosition = rotationMatrix * leftPosition;
    vec2 rotatedRightPosition = rotationMatrix * rightPosition;

    float rotatedUpperWave = wave_height(centeredUV, rotatedLeftPosition);
    float rotatedLowerWave = wave_height(centeredUV, rotatedRightPosition);

    // Kombiniere die Farben basierend auf der Wellenhöhe der gedrehten Lichtq
    vec3 finalColor = upperColor * rotatedUpperWave + middleColor * middleWave

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

Wer sich jetzt darüber beschwert, dass hier die versprochenen Wellen fehlen, hat natürlich Recht. Dafür gibt es den zweiten Effekt.

Komplexerer Effekt mit Wellen

```
#define PI 3.1415926535      // Konstante für Pi
#define X uv.x * 96.0        // Skalierte x-Koordinate
#define Y -uv.y * 96.0       // Skalierte y-Koordinate

uniform float time;          // Zeitvariable
uniform vec2 resolution;     // Bildschirmauflösung

vec3 Zfunc(vec2 uv) {
    float d = sqrt(X * X + Y * Y);           // Entfernung vom Zentrum
    float f = sin(d - time * 9.0) / d * 10.0;  // Wellenfunktion
    float z = f + f;                         // Zweifache Anwendung der Welle
    return vec3(z - 0.5, 0.2 - z, 1.0 - z);   // Farbvektor basierend auf der
}

void main(void) {
    vec2 uv = (gl_FragCoord.xy - 0.5 * resolution.xy) / resolution.y;  // Bere

    vec3 c = vec3(0);  // Initialisiere den Farbvektor

    for (float i = 0.0; i < 6.0; i++) {
        // Berechne die Offset-Position für jede Welle basierend auf der Zeit
        vec2 offset = vec2(sin(time + i) / PI + sin(i) / (7.0 - i), cos(time /
        c += Zfunc(uv - offset);  // Berechne die Farbe basierend auf der Offs
    }
```

```
    // Definiere Farben direkt im Shader-Code
 vec3 baseColor = vec3(1.0, 0.0, 0.0); // Rote Basisfarbe
 vec3 borderColor = vec3(0.0, 0.0, 1.0); // Blaue Randfarbe
 vec3 bgColor = vec3(0.0, 0.0, 0.0);    // Schwarzer Hintergrund

    // Kombiniere die Farben basierend auf der Wellenfunktion und den definier
    vec3 finalColor = baseColor * c / 6.0 + borderColor * (1.0 - c / 6.0) + bg

    gl_FragColor = vec4(finalColor, 1.0);  // Setze die Farbe des Fragments
}
```

So sieht das Resultat aus:

https://www.bytegame.de/wp-content/uploads/2023/11/sh_interference02.mp4

Natürlich kann man die Farben fast beliebig variieren. Und das sieht wirklich super aus, falls man bspw. einen Hintergrund für ein Menü oder Credits sucht.

Erklärung

1. Zfunc-Funktion:
    - Die `Zfunc`-Funktion berechnet die Farbe an einer bestimmten Position (`uv`) auf dem Bildschirm.
    - Die Funktion verwendet die Entfernung `d` von der Zentrumskoordinate und wendet eine Sinuswelle auf diese Entfernung an.
    - Der Sinuswert wird durch die Entfernung `d` geteilt und mit 10 multipliziert, um die Wellenintensität zu steuern.
    - Das Ergebnis wird zweimal auf die Farbkomponenten angewendet, um einen Farbverlauf zu erzeugen.
    - Das Ergebnis ist ein Vektor `vec3`, der die Farbe an der Position `uv` repräsentiert.
2. Haupt-Fragmentshader:
    - Die `main`-Funktion ist der Einstiegspunkt des Fragmentshaders.
    - Zuerst wird die Bildschirmkoordinate `uv` berechnet, wobei der Ursprung in der Mitte des Bildschirms liegt und die Koordinaten auf die Bildschirmauflösung skaliert werden.
    - Ein Vektor `c` wird initialisiert, um die summierte Farbe über verschiedene Positionen zu speichern.
3. Schleife:
    - Eine Schleife von `i = 0` bis `i < 6` durchläuft sechs Iterationen.
    - In jeder Iteration wird ein Offset `offset` für die Positionsberechnung erstellt. Dieser Offset basiert auf trigonometrischen Funktionen (Sinus und Kosinus) von `time` und `i`. Dadurch entstehen Bewegungen in den Wellen.
    - Der `offset` wird von der aktuellen `uv`-Position subtrahiert, und die `Zfunc`-Funktion wird mit dieser neuen Position aufgerufen.

- Das Ergebnis wird zu `c` addiert, um die Farbe von allen sechs Bereichen auf dem Bildschirm zu akkumulieren.

4. Farbdefinitionen:
   - Farben werden als `vec3`-Vektoren definiert, wobei `baseColor` die Grundfarbe, `borderColor` die Randfarbe und `bgColor` die Hintergrundfarbe repräsentiert.
5. Farbverarbeitung:
   - Die Farbverarbeitung erfolgt, indem die Farben basierend auf der Wellenfunktion und den definierten Farben kombiniert werden. `baseColor` wird mit dem Ergebnis von `c` multipliziert, `borderColor` wird mit `(1.0 - c)` multipliziert, und `bgColor` wird ebenfalls mit `(1.0 - c)` multipliziert.
   - Dies führt zu einer Mischung der Farben, wobei die Intensität der Wellen die Anteile der Farben beeinflusst.
6. Setzen der Farbe:
   - Schließlich wird `gl_FragColor` mit dem resultierenden `finalColor` als `vec4` gesetzt, wobei der Alpha-Kanal auf 1.0 gesetzt ist, was bedeutet, dass die Farbe vollständig sichtbar ist.

Das war es auch schon wieder. Viel Spaß bei der Anwendung!

Hier noch ein Beispiel, wo diese Effekte – und viele weitere – verwendet wurden:

Weiterführende Links

Shader-Effekt: Warping
Raster bar Effekt
Projekt Tic-Tac-Toe – Teil 1
Projekt Snake

Date Created
17. November 2023
Author
sven