Spaß im Web – Freude beim programmieren

Die Open Source 2D-Spiele Engine Excalibur.js unter der Lupe.

In diesem Teil der Artikelserie zur browserbasierten Spieleprogrammierung tauchen wir tiefer in die Welt der Grafiken und Sprites ein, um das visuelle Erlebnis der Spiele zu generieren. Wir werden den Umgang mit Actors und Sprites untersuchen, um Charaktere und Objekte zu erstellen und zu verwalten.

von Dr. Veikko Krypczyk

Ausgangspunkt für unsere Betrachtungen ist eine „abgespeckte“ Variante unseres simplen Entwurfs zu einem Breakout-Spiels aus dem ersten Teil. Anhand des Beispiels können wir uns mit essenziellen Aspekten der Grafikprogrammierung in der 2D-Spiele Engine *Excalibur.js* auseinandersetzen. Sehen wir uns das genauer an. Starten wir mit einen Blick in den Quellcode:

import {

  Actor,

  Color,

  Engine,

} from "excalibur";

const game = new Engine({

  width: 800,

  height: 600,

});

const paddle = new Actor({

  x: 150,

  y: game.drawHeight - 40,

  width: 200,

  height: 20,

  color: Color.Chartreuse,

});

game.add(paddle);

const ball = new Actor({

  x: 100,

  y: 300,

  radius: 10,

  color: Color.Red,

});

game.add(ball);

game.start();

Nach den *Import*-Anweisungen und der Erzeugung der *Game-Engin*e werden zwei *Actors* generiert. In diesem Fall sind es ein Actor *ball* und ein Actor *paddle*. Damit haben wir unser erstes Thema für diesen Teil des Artikels. Es geht um *Actors* (nächster Abschnitt) Mit: *npn install* können wir alle Abhängigkeiten im Projekt installieren und das Spiel mittels *npm start* ausführen. Das Spiel ist dann im Browser unter der Adresse <http://localhost:1234/> zu erreichen (*Abbildung 1*).

Abbildung\_1.png

Abbildung 1: Ein Game (Szene) mit zwei Actors (Ball und Schläger).

Überblick über die Artikelserie

Teil 1: Motivation, Einführung in Excalibur.js und erste Schritte

**Teil 2: Fortgeschrittene Grafiken und Sprites**

Teil 3: Physik, Kollision und Interaktionen

Teil 4: Erweiterte Techniken.

./.

Grundlagen von Actors

*Actors* sind zentrale Bausteine der Spielwelt, die jedes interaktive oder sichtbare Objekt im Spiel repräsentieren. Sie können beweglich, unbeweglich, animiert oder statisch sein und nehmen an Kollisionen und physikalischen Berechnungen teil. Ein *Actor* kann alles sein, von einem Spielercharakter über Hindernisse bis hin zu Dekorationen oder Hintergrundelementen. Die Hauptmerkmale eines Actors sind:

* *Position und Größe*: Jeder Actor besitzt eine Position (definiert durch `x` und `y`), sowie Breiten- und Höhenwerte (definiert durch `*width*` und `*height*` die ihn auf der Spielbühne platzieren. Sie bestimmen, wo sich der Actor in der Welt befindet und wie groß er dargestellt wird.
* *Grafiken und Farben*: Ein Actor kann eine Farbe haben oder durch ein *Sprite* dargestellt werden. Standardmäßig werden Actor-Objekte als einfache farbige Rechtecke angezeigt, aber durch Hinzufügen von *Sprites* und *Animationen* kann der visuelle Stil eines Actors komplexer werden. Zum Thema Sprites kommen wir später.
* *Bewegung und Physik*: Actor-Objekte können sich bewegen und physikalisch mit der Welt interagieren. Sie haben Geschwindigkeiten und können durch Kräfte beeinflusst werden, beispielsweise durch Kollisionen oder Schwerkraft. Die Geschwindigkeit (*velocity*) eines Actors wird durch den *vel-Vektor* festgelegt, der aus dem *Vector-Objekt* stammt. Die Methode, um auf die Geschwindigkeit in X- und Y-Richtung zuzugreifen oder sie zu ändern, lautet:

actor.vel = new Vector(100, 0);

Mit dieser Zeile Quellcode setzt man die Geschwindigkeit in X-Richtung auf 100 und in Y-Richtung auf 0. *vel* ist ein *Vector-Objekt*, das sowohl die X- als auch die Y-Komponente der Geschwindigkeit eines Actors definiert. Betrachten wir das an unserem Beispiel, d.h. wir möchten für den Ball (Actor ‘ball‘) eine bestimmte Geschwindigkeit definieren. Der Quellcode stellt sich damit wie folgt dar:

const ball = new Actor({

x: 100,

y: 300,

radius: 10,

color: Color.Red,

});

ball.vel = new Vector(100, 150);

game.add(ball);

In diesem Beispiel bewegt sich der Actor (ein Ball) mit einer Geschwindigkeit von 100 in X-Richtung und 150 in Y-Richtung. Ändern Sie die Werte im Quellcode und beobachten Sie die Auswirkungen nach dem Speichern im Browser ohne Neustart des Spiels (Hot Reload). Die *vel-Eigenschaft* ist somit ein Vektor, der die Geschwindigkeit und Bewegungsrichtung des Actors beschreibt. Darüber hinaus können *Actor*-Objekte verschiedene Kollisionstypen wie *Passive*, *Active* oder *Fixed* haben, um das Kollisionsverhalten zu bestimmen.

* *Aktionssystem*: Actors haben ein integriertes Aktionssystem, mit dem sie Bewegungen und Aktionen in einer Kette durchführen können, beispielsweise *bewegen*, *drehen*, *verkleinern* oder *vergrößern*. Dies wird mit Methoden wie `*moveTo()*` oder `*rotateBy()*` umgesetzt. Auch dazu ein Beispiel. Wir bewegen das Paddle automatisch, so dass es zu einer bestimmten Position animiert wird. Die *moveTo()*-Methode bewegt dieses zu einer angegebenen Position innerhalb eines festgelegten Zeitraums. Sie verwendet Vektoren und der Vorgang dauert so lange, wie die Geschwindigkeit des Actors es vorgibt:

const paddle = new Actor({

x: 150,

y: game.drawHeight - 40,

width: 200,

height: 20,

color: Color.Chartreuse,

});

game.add(paddle);

paddle.actions.moveTo(600, game.drawHeight - 40, 200);

Mit *paddle.actions.moveTo(600, game.drawHeight - 40, 200);* erfolgt die Bewegung des Paddle von seiner Startposition *(150, game.drawHeight - 40)* zu einer neuen Position *(600, game.drawHeight - 40)*. Der dritte Parameter *200* gibt die Geschwindigkeit in Pixeln pro Sekunde an. Hier bewegt sich das Paddle mit einer Geschwindigkeit von 200 Pixeln pro Sekunde. Mit der Methode *moveTo()* kann man dynamische Bewegungen im Spiel definieren und Objekte auf festgelegten Pfaden bewegen.

* *Kollisionserkennung und -reaktion*: Actors können auf Kollisionen mit anderen Actors reagieren. Je nach Kollisionstyp können sie vollständig kollisionsfähig sein oder nur Kollisionen melden, ohne darauf zu reagieren. Man kann Kollisionen und Interaktionen mit der Methode *`on("collisionstart", ...)`* überwachen und steuern.
* *Szenenmanagement*: Jeder Actor wird einer Szene hinzugefügt und interagiert mit der Umgebung in dieser Szene. Durch die Methode *`game.add(actor)`* wird der Actor zur aktuellen Szene hinzugefügt und sichtbar gemacht. Zum Thema *Scenes* kommen wir auch noch später.

Ein Actor hat einen eigenen *Lebenszyklus (Life Cycle)*. Wir sehen uns diesen jetzt an.

Actor Life Cycle

Der *Actor Life Cycle* *in Excalibur.js* beschreibt die verschiedenen Phasen, die ein Actor durchläuft, sobald er erstellt und in die Spielszene eingefügt wird. Das Verstehen des Actor Life Cycles (*Abbildung 2, Tabelle 1*) ist wichtig, um die Interaktionen und die Performance von Actors effizient zu steuern.

Abbildung\_2.png

Abbildung 2: Actor Life Cycle [3].

Wir haben es mit den folgenden Phasen zu tun:

1. *Erstellung*: Der Actor wird erzeugt, aber er ist noch nicht aktiv in der Szene.
2. *Hinzufügen zur Szene:* Der Actor wird zur Spielszene hinzugefügt und beginnt zu interagieren.
3. *Update*: Der Actor wird während jeder Spielschleife aktualisiert (Bewegung, Kollisionen, Physik).
4. *Rendering*: Der Actor wird in jeder Spielschleife auf den Bildschirm gezeichnet.
5. *Kollisionen*: Der Actor nimmt an Kollisionen teil und kann auf Kollisionen reagieren.
6. *Zerstörung*: Der Actor wird entfernt und nicht mehr aktualisiert oder gerendert.
7. *Recycling*: Optional kann ein Actor recycelt und wiederverwendet werden, um Ressourcen zu sparen.

|  |  |  |
| --- | --- | --- |
| Phase | Beschreibung | Beispiel |
| Erstellung | Der Actor wird erzeugt, jedoch noch nicht in die Szene eingefügt. Eigenschaften wie Position, Größe und Farbe werden definiert. | const player = new Actor({ x: 100, y: 100, width: 50, height: 50, color: Color.Blue }); |
| Hinzufügen zur Szene | Der Actor wird der Spielszene hinzugefügt und beginnt, in der Spielwelt sichtbar und aktiv zu werden. | game.add(player); |
| Aktualisieren | In jedem Frame wird die update()-Methode des Actors aufgerufen, um Bewegung, Kollisionen und andere Logik zu berechnen. | player.update = (engine, delta) => { player.pos.x += player.vel.x \* delta; }; |
| Rendering | Der Actor wird in jedem Frame gerendert, wodurch seine visuelle Darstellung basierend auf Position, Farbe oder Sprites aktualisiert wird. | player.graphics.use(playerSprite); |
| Kollisionserkennung und -behandlung | Der Actor erkennt Kollisionen mit anderen Objekten und reagiert entsprechend. Dies wird während der Update-Phase überprüft und kann angepasst werden. | player.on("collisionstart", (event) => { console.log("Collision with: ", event.other); }); |
| Zerstörung | Der Actor wird aus der Szene entfernt und nicht mehr aktualisiert oder gerendert. Er bleibt für eine Weile im Speicher, bevor er vollständig entfernt wird. | player.kill(); |
| Recycling | Anstatt den Actor zu löschen, kann er wiederverwendet werden, indem er aus der Szene entfernt und später wieder hinzugefügt wird. | Nach dem Entfernen wird der Actor wieder in der Szene hinzugefügt, anstatt einen neuen zu erstellen. |

Tabelle 1: Überblick über die Lebensphasen eines Actors in Excalibur.js.

Scenes

In Excalibur.js sind *Scenes* zentrale Bausteine, die das Konzept der Organisation und Trennung von verschiedenen Spielabschnitten oder Leveln ermöglichen. Eine Szene ist ein in sich geschlossener Bereich der Spielwelt, in dem Akteure, Grafiken, Kollisionen und Spiellogik spezifisch für diesen Abschnitt oder Level verwaltet werden. Szenen helfen Entwickler:innen dabei, Spielinhalte logisch zu trennen und zu strukturieren, beispielsweise verschiedene Levels, Menüs oder Zwischensequenzen. Kommen wir zu den Details:

* *Was ist eine Szene (Scene)?* Eine Szene ist eine Instanz der Klasse *`Scene*` *in Excalibur.js*. Sie besteht aus einer eigenen Sammlung von *Actors*, *UI-Elementen* und *Spiel-Logik*, die in einem bestimmten Teil des Spiels aktiv ist. Jede Szene enthält eine eigene Spielwelt und wird isoliert von anderen Szenen verwaltet. Typische Anwendungsfälle sind: verschiedene Level eines Spiels, ein Hauptmenü, ein Pausenmenü, Optionenbildschirm oder Zwischensequenzen, beispielsweise ein Intro oder Endsequenz.
* *Lebenszyklus einer Szene*: Szenen können in verschiedenen Phasen aktiv sein. Das sind die Phasen *Initialisierung*, *Aktivierung*, *Aktualisierung*, *Rendering*, *Deaktivierung* und *Zerstörung* (*Tabelle 2, Abbildung 3*).

|  |  |  |
| --- | --- | --- |
| Phase | Beschreibung | Beispiel |
| Initialisierung | Die Szene wird erstellt und vorbereitet, aber noch nicht aktiviert. Hier werden die grundlegenden Objekte und Actor-Elemente definiert. | const levelOne = new Scene(); |
| Hinzufügen zur Engine | Die Szene wird zur Engine hinzugefügt, damit sie später aktiviert und verwendet werden kann. | game.add("levelOne", levelOne); |
| Aktivierung | Die Szene wird durch *goToScene()* aktiviert. Dies macht die Szene zur aktiven Spielszene, die gerendert und aktualisiert wird. | game.goToScene("levelOne"); |
| Aktualisierung | Die *update()-*Methode der Szene wird in jedem Frame aufgerufen, um die Spiel-Logik, die Bewegung der Actors und die Kollisionen zu verwalten. | scene.update(engine, delta); |
| Rendering | Die Szene rendert ihre Actors und Grafikelemente auf den Bildschirm. Nur die aktuelle Szene wird in jedem Frame gerendert. | scene.draw(ctx, delta); |
| Deaktivierung | Wenn eine andere Szene aktiviert wird, wird die aktuelle Szene deaktiviert und pausiert. Sie bleibt jedoch im Speicher erhalten. | game.goToScene("mainMenu"); // Deaktiviert levelOne |
| Zerstörung oder Entfernung | Eine Szene kann aus der Engine entfernt werden, wenn sie nicht mehr benötigt wird. | game.remove("levelOne"); |

Tabelle 2: Phasen, die eine Szene im Laufe ihrer Lebensdauer in Excalibur.js durchläuft.

Abbildung\_3.png

Abbildung 3: Scene Life Cycle [2].

Sehen wir uns das an einem Beispiel an. Szenen sind ideal, um ein *Hauptmenü* oder *Level*-*Übergänge* zu erstellen. Sie können eine Szene für das Hauptmenü definieren und bei der Auswahl eines Levels zur entsprechenden Levelszene wechseln. Der Quellcode könnte dafür wie folgt aussehen:

import { Actor, Color, Engine, Scene, Vector } from "excalibur";

// Erstelle das Spiel (Engine)

const game = new Engine({

width: 800,

height: 600

});

// Menüszene erstellen

const menu = new Scene();

// Level 1 Szene erstellen

const level1 = new Scene();

// Menüszene initialisieren

menu.onInitialize = () => {

// Erstelle einen Button in der Mitte des Bildschirms

const startButton = new Actor({

x: game.drawWidth / 2,

y: game.drawHeight / 2,

width: 100,

height: 50,

color: Color.Green

});

// Wenn der Button angeklickt wird, zur Level 1 Szene wechseln

startButton.on("pointerup", () => {

game.goToScene("level1");

});

// Füge den Button zur Menüszene hinzu

menu.add(startButton);

};

// Level 1 Szene initialisieren

level1.onInitialize = () => {

// Erstelle einen Actor (z.B. Spielercharakter) im Level 1

const player = new Actor({

x: 100,

y: 300,

width: 50,

height: 50,

color: Color.Blue

});

// Füge den Spieler zur Level 1 Szene hinzu

level1.add(player);

};

// Füge die Szenen zur Engine hinzu

game.add("menu", menu);

game.add("level1", level1);

// Starte mit der Menüszene

game.goToScene("menu");

// Starte das Spiel

game.start();

Erklärung des Quellcodes:

* *Engine*: Die Spiel Engine wird mit einer Auflösung von 800x600 erstellt.
* *Szenen*: Zwei Szenen werden erstellt – eine für das *Hauptmenü* und eine für *Level 1*.
* *Hauptmenüszene*: Ein *Button* wird in der Mitte des Bildschirms erstellt, der die Farbe *Grün* hat. Wenn der Benutzer auf den Button klickt (mit *pointerup*), wechselt das Spiel zu *Level 1*.
* *Level 1-Szene*: Ein blauer Actor wird als Spielercharakter im Level hinzugefügt.
* *Szene wechseln*: Das Spiel beginnt mit der Menüszene (*menu*), und beim Drücken des *Start-Buttons* wechselt das Spiel zu Level 1.
* *Spielstart*: Die Engine wird mit *game.start()* gestartet.

Mit diesem Code erhält man ein einfaches Beispiel, wie man in *Excalibur.js* Szenen erstellen und zwischen einem Hauptmenü und einem Level wechseln kann. Die Menüdarstellungen wurden hier lediglich angedeutet. Szenen können auch verwendet werden, um Pausenmenüs zu implementieren. Dies kann erreicht werden, indem eine Pausenszene erstellt wird, die durch einen Tastenanschlag aufgerufen und angezeigt wird. *Excalibur.js* verwaltet Szenen effizient, indem es sicherstellt, dass nur die aktiven Szenen aktualisiert und gerendert werden. Nicht aktive Szenen bleiben im Speicher, sind jedoch inaktiv, bis sie wieder aktiviert werden. Dies ermöglicht eine einfache Verwaltung von komplexeren Spielen, bei denen verschiedene Spielzustände (z.B. Menüs, Level und Zwischensequenzen) voneinander getrennt und unabhängig verwaltet werden können. Scenes in *Excalibur.js* bieten eine klare Möglichkeit, verschiedene Abschnitte eines Spiels zu strukturieren und zu trennen. Von der Verwaltung von Levels, Menüs, Pausenfunktionen bis hin zu Zwischensequenzen helfen Szenen, die Spielstruktur und den Fluss des Spiels zu steuern.

Sprites

*Sprites* sind grafische Darstellungen von Objekten in 2D-Spielen. Sie werden oft verwendet, um Akteure (Actors) visuell darzustellen. In *Excalibur.js* sind Sprites ein grundlegendes Mittel zur Anzeige von grafischen Objekten wie Spielern, Gegnern und Objekten in der Spielwelt. Ein *Sprite* enthält eine Bilddatei, welche die grafische Darstellung bildet. Ein einzelnes Sprite kann von einer Bildquelle geladen und einem Actor zugewiesen werden. Der *Actor* verwendet dann das *Sprite* als visuelle Darstellung. Dazu ein Beispiel mit den folgenden Schritten:

1. *Laden der Bildressource für das Sprite:* Nehmen wir an wir haben eine Bilddatei mit der Bezeichnung ‘*Monster.png*‘, welche wir für das Sprite verwenden möchten. Dann können wir eine Ressource wie folgt laden:

import { ImageSource, Loader } from "excalibur";

import smonster from "./images/monster.png";

export const Resources = {

  Mmonster: new ImageSource(smonster)

} as const;

export const loader = new Loader();

for (const res of Object.values(Resources)) {

  loader.addResource(res);

}

Ebenso wird ein sogenannter *Loader* für *Excalibur.js* generiert, damit dieser beim Erzeugen des Spiels alle Ressourcen laden kann. Damit man in der Programmiersprache TypeScript ‘*\*.png-Dateien‘* laden kann benötigt man folgende Anweisung (in einer eigenen Datei):

declare module "\*.png";

Die Anweisung *`declare module "\*.png";`* wird in TypeScript verwendet, um dem Compiler mitzuteilen, wie er mit Dateien eines bestimmten Typs (in diesem Fall *`\*.png`-Dateien*) umgehen soll. TypeScript ist von Haus aus stark typisiert und kennt nicht automatisch den Typ von Nicht-JavaScript-Dateien wie Bildern, CSS-Dateien oder anderen Dateitypen. Diese Deklaration erlaubt es TypeScript, *`.png`*-Dateien wie Module zu behandeln. Das bedeutet, dass wir Bilddateien wie `*import*`-Module in den TypeScript-Code einbinden können, ohne dass der Compiler Fehler meldet.

1. *Erstellen eines Actors und Zuweisen des Sprites mit der Bild-Ressource:*

import { Actor, Color, vec } from "excalibur";

import { Resources } from "./resources";

export class Player extends Actor {

  constructor() {

    super({

      pos: vec(150, 150),

      width: 100,

      height: 100

    });

  }

  onInitialize() {

    this.graphics.add(Resources.Mmonster.toSprite());

  }

}

Dieser TypeScript-Code definiert eine Klasse `*Player*`, die von der Basisklasse `*Actor*` erbt. Der Konstruktor erstellt den `*Player*`-Actor an der Position *`(150, 150)`* mit einer Breite und Höhe von *`100x100`* Pixeln. *`super({...})`* ruft den Konstruktor der *`Actor`*-Klasse auf und übergibt die Positions- und Größeninformationen. Die *onInitiali*z*e()-*Methode wird aufgerufen, sobald der Actor zum Spiel hinzugefügt wird. In dieser Methode wird das Sprite mittels der Methode *`Resources.Mmonster.toSprite()* zum Actor hinzugefügt. Damit wird die Grafik dem Actor zugewiesen, sodass der `*Player*` das `*monster.png*`-Sprite anzeigt.

1. *Die Game-Klasse:* Diese hat jetzt folgenden Aufbau:

import { Engine } from "excalibur";

import { Player } from "./player";

import { loader } from "./resources";

class Game extends Engine {

    constructor() {

      super({width: 800, height: 600});

    }

    initialize() {

      const player = new Player();

      this.add(player);

      this.start(loader);

    }

  }

  export const game = new Game();

  game.initialize();

Dieser TypeScript-Code erstellt eine einfache Spielstruktur, indem er die Spiel-Engine, einen *Player*-Actor und einen *Ressourcen-Loader* verwendet. Der Konstruktor ruft den `*Engine*`-Konstruktor auf (`*super*`) und legt die Größe des Spielfensters auf 800x600 Pixel fest. Die Methode *`initialize()`* erstellt eine neue Instanz der `*Player*`-Klasse, also eine Spielerfigur. Dann wird der `*Player*` zum Spiel hinzugefügt *(`this.add(player)`*), wodurch er als Spielobjekt in die Spielszene integriert wird. Schließlich wird *`this.start(loader)`* aufgerufen, um das Spiel zu starten und alle notwendigen Ressourcen (definiert im *`loader`)* zu laden. Eine Instanz des Spiels wird als *`game`* exportiert*.`game.initialize()`* wird aufgerufen, um das Spiel zu initialisieren und den Player hinzuzufügen sowie das Spiel zu starten.

Im Beispiel wurde ein *Sprite* aus einem Bild erstellt und einem *Actor* zugewiesen. Dieser *Actor* wird dann in die Szene eingefügt und im Spiel gerendert. Durch die Arbeit mit dem *Loader* wird ein entsprechendes Startbild angezeigt, bis der Startvorgang fertig ist (*Abbildung 4*).

Abbildung\_4.png

Abbildung 4: Darstellung eines Sprites, z.B. in einem Actor als Spielfigur.

Spritesheets

Ein *Spritesheet* ist eine Bilddatei, die mehrere Einzelgrafiken (Sprites) in einem Raster oder einer Matrix zusammenfasst. Diese Technik wird häufig in Spielen verwendet, um Speicher und Ladezeiten zu optimieren, da mehrere Animationen oder Grafiken in einer einzigen Datei zusammengefasst werden, anstatt viele einzelne Bilddateien zu laden. Ein Spritesheet besteht aus:

* *Mehreren Sprites:* Jedes Sprite stellt eine einzelne Grafik oder eine Animation eines Charakters, Objekts oder Effekts dar.
* *Rasterstruktur:* Die Sprites sind in gleichmäßigen Abständen in Zeilen und Spalten angeordnet.
* *Einheitliche Größe:* Jedes Sprite hat oft die gleiche Breite und Höhe, damit es leicht zu extrahieren und verwenden ist.

*Spritesheets* werden oft für Animationen verwendet. Zum Beispiel könnte jedes Sprite eine Phase einer Animation darstellen, wie z.B. die verschiedenen Frames eines laufenden Charakters. Anstatt jedes Frame einzeln zu speichern und zu laden, wird eine Sequenz von Frames als Teil eines Spritesheets geladen und dann in einer bestimmten Reihenfolge gerendert, um eine flüssige Animation zu erzeugen. Kommen wir zu einem Beispiel: Ein Spritesheet für ein Kartenspiel könnte die einzelnen Karten in Zeilen und Spalten anordnen (*Abbildung 5*).

Abbildung\_5.png

Abbildung 5: Ein Spritesheets enthält mehrere Einzelgrafiken in Form eines Rasters.

Wird ein Spritesheet für eine Animation genutzt, so ist jedes Frame in der Animation ein Teilbild. Vorteile des Einsatzes von *Spritesheets* sind:

* *Performance-Optimierung*: Statt viele einzelne Bilddateien zu laden, wird ein Spritesheet verwendet, um mehrere Sprites in einer Datei zusammenzufassen. Dies reduziert die Anzahl der notwendigen Lade- und Renderoperationen, was zu einer verbesserten Leistung führt.
* *Einfachere Animationen*: Ein Spritesheet ermöglicht es, Animationen effizient zu verwalten, da die verschiedenen Animationsphasen (z.B. Laufen, Springen) in einer Datei gespeichert sind.
* *Speicherplatz sparen*: Durch die Bündelung von Sprites in einer Datei kann der Speicherplatzbedarf im Vergleich zu mehreren einzelnen Bilddateien reduziert werden.

Struktur eines Spritesheets

Die Struktur eines Spritesheets hängt von der Art und Weise ab, wie das Spritesheet erstellt wurde. Wie kann man die Struktur eines Spritesheets bestimmen? Es gibt folgende Optionen:

* *Spritesheets von Dritten:* Spritesheets werden oft von Künstlern oder Tools erstellt. Normalerweise gibt es Vorgaben oder Dokumentationen, die die Struktur des Spritesheets beschreiben.
* *Anhand der Grafik selbst*: Man kann das Spritesheet öffnen (z B. in einem Bildbearbeitungsprogramm wie Photoshop, GIMP). Dort kann man die Abmessungen des gesamten Bildes und jedes einzelnen Sprites ermitteln. Ein Beispiel: Wenn das Spritesheet eine Größe von 256x256 Pixel hat und jedes Sprite 64x64 Pixel groß ist, kann man daraus schließen, dass es sich um ein 4x4-Raster handelt (16 Sprites gesamt).
* *Tools zur Erstellung von Spritesheets*: Es gibt verschiedene Tools, die Spritesheets automatisch erstellen. Diese Tools erzeugen oft zusätzlich eine JSON oder XML\*-Datei, die die Struktur des Spritesheets beschreibt, z.B. die Position, Größe und Reihenfolge der Sprites.

./.

Text

In Excalibur.js wird die Textdarstellung primär über die Klasse `*Label*` realisiert, die speziell dafür ausgelegt ist, Text in die Spielszene einzufügen, zu formatieren und an beliebiger Stelle darzustellen. Labels bieten eine Vielzahl von Optionen, um Text in Spielen zu gestalten, von einfachen Statusanzeigen bis hin zu dynamischen *Heads-Up Display-(HUD)-*Elementen. Ein `*Label*` ist im Grunde ein *Actor*-ähnliches Objekt, das Text darstellt und in der Spielszene bewegt und transformiert werden kann. Sie können ein Label auf einfache Weise erstellen und ihm Eigenschaften wie Text, Position, Farbe und Schriftart zuweisen. Ein Beispiel in Form der Einblendung des Spielstandes (Score) in Form eines HUD (*Abbildung 6*) könnte wie folgt aussehen:

import { Engine, Label, Color, Font, FontUnit } from 'excalibur';

// Erstelle das Spiel (Engine)

const game = new Engine({

width: 800,

height: 600

});

// Erstelle ein Text-Label für den Punktestand (Score)

const scoreLabel = new Label({

text: 'Score: 0',

x: 20,

y: 20, // Position oben links

color: Color.White,

font: new Font({

family: 'impact',

size: 24,

unit: FontUnit.Px}),

});

// Füge das Label dem Spiel hinzu

game.add(scoreLabel);

// Führe das Spiel aus

game.start().then(() => {

console.log('Spiel gestartet');

});

// Einfache Logik, um den Punktestand zu aktualisieren (fiktiv)

let score = 0;

game.on('preupdate', () => {

score += 1;

scoreLabel.text = `Score: ${score}`;

});

Abbildung\_6.png

Abbildung 6: Ein Label als HUD für die Spielstandanzeige.

Beachten Sie dazu die folgenden Erläuterungen:

* *Textinitialisierung*: Das Label wird mit dem anfänglichen Text "*Score: 0*" erstellt. Dieser Text wird im HUD (oben links) angezeigt.
* *Positionierung*: Das Label wird an den Koordinaten (*x: 20, y: 20*) platziert, also in der oberen linken Ecke des Bildschirms.
* *Farbe*: Die Textfarbe wird auf Weiß (*Color.White*) gesetzt, um einen klaren Kontrast zum Hintergrund zu bieten.
* *Schriftart*: Die Schriftart wird auf *Impact* gesetzt, mit einer Größe von *24 Pixel*. Dies geschieht durch die Definition eines neuen *Font*-Objekts, das die Schriftfamilie, -größe und -einheit (Pixel) festlegt.

Beispielhaft aktualisieren wir das Label bei jedem Update-Vorgang der Spiele-Engine. Hier wird lediglich der angezeigte Wert um eins erhöht (inkrementiert). In einer produktiven Anwendung könnte an dieser Stelle der Punktstand, verbliebende Spielzeit usw. ausgegeben werden:

* *Logik*: Die Aktualisierung erfolgt im *preupdate*-Event der Spiel-Engine, das vor jedem neuen Frame ausgelöst wird.
* *Punktstandserhöhung*: Der Punktestand (score) wird bei jedem Frame um 1 erhöht.
* *Label-Aktualisierung*: Der neue Wert des Punktestands wird dem Label als Text zugewiesen, wodurch der sichtbare Text auf dem Bildschirm kontinuierlich aktualisiert wird (z.B. "Score: 1", "Score: 2", usw.).

In *Excalibur.js* gibt es weitere Möglichkeiten, *Text* innerhalb der Spielszene darzustellen. Eine Methode ist die Verwendung des `Text`-Objekts ‘Text` können Sie sowohl statische als auch dynamische Texte in Ihrem Spiel anzeigen, und es bietet viele Anpassungsoptionen, wie Schriftgröße, Schriftart, Farbe und Ausrichtung. Dieses Text-Objekt wird jedoch nicht eigenständig gerendert, sondern muss innerhalb eines anderen Objektes, beispielsweise einem Actor verwendet werden. Auch dazu haben wir ein Beispiel vorbereitet (*Abbildung 7*):

import {Text, Engine, Actor, vec, Font, Color, FontUnit, FontStyle, TextAlign, BaseAlign, Direction } from 'excalibur';

const game = new Engine({

  width: 800,

  height: 600

});

const text = new Text({

    text: 'Some Text Drawn Here\n\nNext line',

    font: new Font({

      size: 30,

      unit: FontUnit.Px,

      family: 'sans-serif',

      style: FontStyle.Normal,

      bold: false,

      textAlign: TextAlign.Left,

      baseAlign: BaseAlign.Alphabetic,

      direction: Direction.LeftToRight,

      shadow: {

        blur: 2,

        offset: vec(2, 2),

        color: Color.Black,

      },

    })

});

const actor = new Actor({

    pos: vec(300, 100)

});

actor.graphics.use(text);

game.add(actor);

game.start();

Abbildung\_7.png

Abbildung 7: Ein Text-Objekt (mehrzeilig) innerhalb eines Actor-Objektes.

Was passiert im Einzelnen:

* *Erstellung des Text-Objekts*: Ein *`Text`-*Objekt wird erstellt, das den angezeigten Text und dessen Formatierung definiert. Der Text selbst lautet *"Some Text Drawn Here\n\nNext line"* und enthält manuelle Zeilenumbruch-Sequenzen (*`\n*`), die dafür sorgen, dass der Text in Zeilen angezeigt wird.
* *Schriftkonfiguration*: Ein `*Font*`-Objekt wird verwendet, um die Schriftgröße (30 Pixel), die Schriftfamilie (*sans-serif*), den Schriftstil (*normal*) und die Ausrichtung des Textes festzulegen. Es wird außerdem kein Fettdruck verwendet.
* *Ausrichtung*: Der Text wird linksbündig (*TextAlign*); alphabetisch (*BaseAlign*), was sich auf die vertikale Positionierung der Textbasis bezieht und von links nach rechts (*Direction*) ausgerichtet.
* *Schattierung*: Ein Schatteneffekt wird hinzugefügt, der eine leichte Unschärfe (Blur von 2) und eine Versetzung des Schattens um 2 Pixel in jede Richtung erzeugt. Die Farbe des Schattens ist Schwarz.

Dann erfolgt die Integration in ein *Actor*-Element:

* *Actor-Erstellung*: Der `*Text*` wird in einem `*Actor*`-Objekt platziert. Der `*Actor*` dient als Container für den Text und ermöglicht es, den Text auf dem Bildschirm zu positionieren und zu bewegen.
* *Text in Actor verwenden*: Der Text wird dem `*Actor*` über die `*graphics.use()`-*Methode zugewiesen, wodurch der ‘*Text‘* auf dem ‘*Actor*‘ sichtbar gemacht wird.
* *Positionierung*: Der `*Actor*` wird an einer bestimmten Position auf dem Bildschirm platziert *(`vec(300, 100)`),* wodurch das Text‘-Objekt an dieser Stelle im Spielfenster angezeigt wird.

Zusammengefasst: Das ‘Text‘-Objekt wird als grafisches Element erstellt, formatiert und in einem `*Actor*` integriert, der ihn auf dem Bildschirm positioniert und verwaltet. Fassen wir den Unterschied zwischen ‘*Text‘*- und ‘*Label‘*-Objekten für die Darstellung von Text nochmals zusammen (*Tabelle 3*).

|  |  |  |
| --- | --- | --- |
| Merkmal | Text | Label |
| Definition | Ein Text-Objekt, das speziell für das Zeichnen von Text mit erweiterter Formatierung und Styling verwendet wird. | Ein spezielles UI-Element für einfache Textdarstellungen im Spiel. |
| Verwendung | Dient der Darstellung von komplexem Text, inklusive Schriftformatierung, Schatten, Ausrichtung usw. | Wird meist verwendet, um statischen oder sich ändernden Text (z.B. Punktestand) anzuzeigen. |
| Erweiterte Formatierung | Bietet umfassende Steuerung über Schriftgröße, -familie, -stil, Schatten, Ausrichtung und Textfluss (links nach rechts usw.). | Einfachere Steuerung über grundlegende Schriftattribute wie Größe und Farbe. |
| Ausrichtung | Unterstützt detaillierte Ausrichtungsoptionen wie Textausrichtung (z.B. linksbündig, zentriert) und Basislinienausrichtung. | Bietet grundlegende Ausrichtungsmöglichkeiten, aber weniger detailliert als das Text-Objekt. |
| Schattierung | Unterstützt Schatteneffekte (z.B. Versatz, Unschärfe) für den Text. | Keine Unterstützung für Schatteneffekte. |
| Integration in Actor | Muss in einem Actor verwendet werden, um auf dem Bildschirm angezeigt zu werden. | Kann als eigenständiges Element im Spiel hinzugefügt und positioniert werden. |
| Dynamische Updates | Kann dynamisch aktualisiert werden, z.B. durch Veränderung der Zeichenkette oder Schriftattribute. | Wird häufig für dynamische Textanzeigen wie Punktestand, Timer, etc. verwendet. |
| Komplexität | Eignet sich für detailreiche Textdarstellungen mit erweiterten Formatierungsoptionen. | Eignet sich für einfache, schnelle Textanzeigen, die leicht zu implementieren sind. |
| Performance | Kann etwas mehr Rechenleistung erfordern, wenn viele erweiterte Optionen genutzt werden. | Leichter und performanter für einfache Textdarstellungen. |

Tabelle 3: Gegenüberstellung von Text- und Label-Element für die Textdarstellung.

Kurz zusammengefasst: Ein ‘*Text‘-Objekt* wird für komplexere Textdarstellungen mit erweiterten Formatierungsmöglichkeiten verwendet. Ein ‘*Label*‘-Objekt eignet sich für einfachere Textanzeigen und ist oft ausreichend für grundlegende HUD-Elemente wie Punktestände oder Timer.

Fazit und Ausblick

Zu den grafischen Elementen gibt es noch viele weitere Objekte, welche uns die Bibliothek *Excalibur.js* zur Verfügung stellt, beispielsweise Graphics, Raster Graphics, Linien usw. Ein wichtiges Feature, um bei der Spieleprogrammierung voranzukommen, ist das Thema *Tilemaps*. Eine *Tilemap* ist eine zweidimensionale Rasterstruktur, die aus kleineren Kacheln (*Tiles*) besteht, die wiederholt werden, um größere Spielflächen wie Karten, Levels oder Hintergründe darzustellen. Jede Kachel repräsentiert ein einzelnes Element (z.B. Boden, Wand, Wasser). Tilemaps ermöglichen es, große Spielwelten effizient zu erstellen und zu verwalten, indem Kacheln wiederverwendet werden. Sie werden häufig in 2D-Spielen für die Darstellung von Levels oder Umgebungen verwendet. Tilemaps sind besonders nützlich für prozedurale Generierung und flexible Levelgestaltung. Wir verwenden diese auch in *Excalibur.js. Tilemaps*, *Physik*, *Kollision* und *Interaktionen* sind das Thema unseres nächsten Teils der Serie.

Quellcode

Die Quellcode-Snippets zu diesem Teil der Artikelserie kann man von [4] herunterladen.

./.

*Dr. Veikko Krypczyk ist begeisterter Entwickler und Fachautor. Er bietet Seminare und Workshops zu unterschiedlichen Themen der Softwareentwicklung. Weitere Informationen zu diesen und anderen Themen der IT und Angebote zu Seminaren und Workshops finden Sie unter https://larinet.com und https://www.tech-punkt.com. Hier finden Sie Seminare zu aktuellen Themen, welche den Inhalt kompakt und auf den Punkt gebracht vermitteln.*

Links & Literatur

[1] <https://excaliburjs.com/>

[2] <https://excaliburjs.com/docs/actors>

[3] <https://excaliburjs.com/docs/scenes>

[4] <https://larinet.com>