Spaß im Web – Freude beim programmieren

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

Business-Apps bestimmen die Aufträge der meisten Entwickler:innen. Auch der Autor kam bisher wenig mit der Spiele-Programmierung in Berührung. Über Zufall ist er auf die Bibliothek Excalibur.js gestoßen und dort länger hängen geblieben als geplant. Aus einem flüchtigen Blick wurde eine Stunde, ein Abend, ein erstes Projekt und vielleicht demnächst die Realisierung einer ersten Spielidee – auf jeden Fall diese Artikelserie.

von Dr. Veikko Krypczyk

Statt komplizierter Datenbankabfragen und langwieriger Diskussionen um das beste User Interface geht es in dieser Artikelserie um die Programmierung von Spielen. Keine großen 3D-Games, sondern kleine interaktive Spielwelten, welche direkt im Browser ausgeführt werden. Leser:innen, welche noch nicht genau wissen, ob sie weiterlesen möchten, die mögen auf die folgenden motivieren Argumente schauen:

* *Spaß am Experimentieren*: Die Spieleentwicklung bietet viel Raum für kreatives Ausprobieren. Man kann Ideen testen und schnell Ergebnisse sehen, was den gesamten Prozess dynamisch und unterhaltsam macht. Die sofortige Rückmeldung, die durch das Testen von Prototypen möglich ist, motiviert dazu, weiter an Ideen zu feilen und sie zu perfektionieren.
* *Langfristige Freude am Erfolg:* Zu sehen, wie aus einer kleinen Idee ein vollständig spielbares Produkt wird, ist eine äußerst erfüllende Erfahrung. Für viele ist es nicht nur der Prozess des Schaffens, der Freude bereitet, sondern auch das Endprodukt: ein fertiges Spiel, das anderen Spaß bereitet.
* *Lernen durch Praxis:* Die webbasierte Spieleentwicklung bietet eine praktische und unterhaltsame Möglichkeit, grundlegende Programmierkenntnisse zu vertiefen. Technologien wie HTML5, CSS3 und JavaScript, oft zusammen mit Frameworks wie *Excalibur.js*, werden verwendet, um Spiele zu erstellen. Dies ermöglicht es Fähigkeiten in der Webentwicklung spielerisch zu erweitern und sich dabei auch mit Themen wie Grafik, Physik und Sound zu beschäftigen.
* *Breite Zugänglichkeit:* Webbasierte Spiele können direkt im Browser gespielt werden, ohne dass zusätzliche Software installiert werden muss. Dadurch ist der potenzielle Spielerkreis enorm groß, da fast jeder ein Gerät mit einem Browser besitzt, sei es ein Computer, Smartphone oder Tablet. Diese Barrierefreiheit fördert die Motivation, Spiele für ein weltweites Publikum zu entwickeln.

Ich habe bewusst die Punkte Spaß und Freude ganz nach oben gerückt. Es geht darum sich spielerisch mit neuen Technologien der Programmierung auseinanderzusetzen. Die gewonnen Erkenntnisse sind auch gewinnbringend nutzbar, wenn wir an unseren Business-Applikationen „werkeln“. Als technische Basis nutzen wir die 2D-Spiele-Engine *Excalibur.js* [1]. Starten wir mit einem Überblick über *Excalibur.js*.

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.

./.

Excalibur.js – Ein Überblick

*Excalibur.js* ist eine Open Source 2D Spiele-Engine, die in *TypeScript* geschrieben wurde und sich auf die Erstellung von webbasierten 2D-Spielen konzentriert. Die Engine wurde entwickelt, um sowohl Anfängern als auch erfahrenen Entwickler:innen eine benutzerfreundliche und leistungsstarke Plattform zu bieten. Sie ermöglicht es, Spiele direkt im Browser auszuführen, wodurch sie für Webentwickler besonders attraktiv ist. *Excalibur.js* vereinfacht den Prozess der Spieleentwicklung, indem es viele grundlegende Aspekte wie Szenenmanagement, Physik, Eingabeverarbeitung und Rendering handhabt. Durch den Einsatz von TypeScript, als Alternative zu JavaScript profitiert man von einer hohen Typsicherheit und besserer Wartbarkeit des Codes. Excalibur.js ist eine „*batteries included*“-Engine, was bedeutet, dass sie viele Funktionen von Haus aus bereitstellt, wie beispielsweise:

* *Szenen und Akteure (Actors):* Excalibur.js arbeitet mit einem Theater-Metapher-Ansatz, bei dem Spiele in „Szenen“ aufgeteilt werden, die jeweils verschiedene „Akteure“ enthalten. Akteure können bewegliche Charaktere, Gegner oder Objekte im Spiel sein. Entwickler:innen können diese leicht erstellen und verwalten, um komplexe Spiele zu bauen.
* *Physik und Kollisionen*: Die Engine bietet eine eingebaute Physik-Engine, die sowohl einfache als auch realistischere Physikmodelle unterstützt. Dadurch können Kollisionen zwischen Objekten automatisch erkannt und bearbeitet werden, ohne dass man viel Code schreiben muss.
* *Rendering von Sprites und Animationen*: Excalibur.js ermöglicht das einfache Laden und Anzeigen von Sprites, die Grundbausteine von 2D-Spielen. Es unterstützt zudem die Erstellung von Animationssequenzen, um flüssige Bewegungen von Charakteren oder Objekten zu ermöglichen.
* *Eingabe und Benutzerinteraktionen*: Die Engine bietet umfassende Unterstützung für Benutzerinteraktionen über Maus, Tastatur, Touch-Eingaben und Gamecontroller, was eine flexible Steuerung ermöglicht.

Es gibt mehrere Gründe, warum Excalibur.js eine attraktive Wahl für die Entwicklung von 2D-Webspielen ist:

* *TypeScript-basiert*: Excalibur.js ist vollständig in TypeScript geschrieben, was es Entwickler:innen ermöglicht, von den Vorteilen einer typsicheren Sprache zu profitieren. Dies führt zu weniger Fehlern, besserer Code-Qualität und einer leichteren Wartung, insbesondere bei größeren Projekten. Entwickler, die bereits mit JavaScript vertraut sind, werden sich in TypeScript schnell zurechtfinden und von den zusätzlichen Funktionen profitieren.
* *Einfache Handhabung und geringe Einstiegshürde*: Die API von Excalibur.js ist gut dokumentiert und einfach zu verwenden. Dadurch eignet sich die Engine hervorragend für Anfänger:innen, die erste Schritte in der Spieleentwicklung machen, ohne sich zu sehr mit der technischen Komplexität einer Spiele-Engine auseinandersetzen zu müssen. Gleichzeitig ist sie flexibel genug, um auch fortgeschrittene Spieleentwickler:innen zufriedenzustellen.
* „*Batteries included*“-Ansatz: Excalibur.js liefert viele nützliche Funktionen von Haus aus, so dass Entwickler sich auf das Design und die Logik ihres Spiels konzentrieren können, anstatt sich um die Implementierung grundlegender Systemfunktionen zu kümmern. Dazu gehören das Management von Szenen, die Steuerung von Akteuren, Physik, Animationen und die Eingabeverarbeitung.
* *Cross-Platform und Browser-Unterstützung*: Da Excalibur.js in TypeScript/JavaScript entwickelt wurde, funktioniert es in allen modernen Browsern und auf verschiedenen Geräten, einschließlich mobiler Endgeräte. Das macht es einfach, ein Spiel zu entwickeln, das plattformübergreifend zugänglich ist, ohne Anpassungen an die Spiel-Engine vornehmen zu müssen. Zudem können Spiele mit nativen Wrappern wie Apache Cordova oder Electron in native Apps für Mobilgeräte oder Desktops umgewandelt werden.
* *Aktive Community und Open Source*: Excalibur.js ist vollständig Open Source, was bedeutet, dass die Community aktiv zur Weiterentwicklung der Engine beitragen kann. Entwickler:innen haben Zugang zu einer großen Menge an Ressourcen, Dokumentation und Beispielen. Außerdem gibt es viele Tutorials und Unterstützung, was es einfacher macht, schnell in die Spieleentwicklung einzusteigen.

Insgesamt hoffe ich, damit genügend Interesse zum Ausprobieren (und Weiterlesen) geweckt zu haben. Schauen wir einmal, wie weit wir in diesem Teil noch kommen.

Excalibur.js installieren und erste Schritte

Um mit Excalibur.js zu starten, müssen Sie einige grundlegende Installationsschritte durchlaufen und ein Projekt initialisieren. Im Folgenden wird dieser Prozess beschrieben. Bevor Sie mit der Installation beginnen, stellen Sie sicher, dass Ihr System folgende Voraussetzungen erfüllt:

* *Node.js* und *npm* (Node Package Manager): *Node.js* ist notwendig, um JavaScript außerhalb eines Browsers auszuführen und wird gemeinsam mit dem *npm*-Paketmanager installiert, der für die Verwaltung der Projektabhängigkeiten erforderlich ist. Laden Sie Node.js von der offiziellen Website Node.js, <https://nodejs.org> herunter und installieren Sie es. Sobald Node.js installiert ist, können Sie die Installation überprüfen, indem Sie die folgenden Befehle in der Kommandozeile ausführen:

node -v

npm -v

Diese Befehle geben die installierten Versionen von *Node.js* und *npm* zurück, falls die Installation erfolgreich war.

* *Ein Texteditor oder eine IDE*: Sie benötigen einen Editor oder eine Entwicklungsumgebung, um den Code zu schreiben. Empfohlene Editoren sind Visual Studio Code, Sublime Text oder Atom. Visual Studio Code ist besonders beliebt, da es TypeScript nativ unterstützt.

Starten wir mit einem ersten Beispiel, welches wir von der Homepage der Spiele-Engine laden. Die Beschreibung finden Sie unter [3]; das Repository finden Sie unter [4]. Das Ziel ist es ein Zusammenspiel der folgenden Komponenten zu erreichen:

* TypeScript-Compiler für die Übersetzung des TypeScript-Codes in JavaScript.
* Lokaler Webserver, um das Spiel auf den Entwicklungsrechner ausführen und testen zu können.
* Überwachung von Änderungen am Quellcode (TypeScript), neu kompilieren des Quellcodes (TypeScript-Compiler) und Weiterleiten der Änderungen an den lokalen Webserver, damit die Applikation (Game) neu ausgeführt werden kann (Hot Reload).

Als Tool kommt dazu der *Parcel-Bundler* zum Einsatz (siehe Textkasten).

Parcel-Bundler mit lokalem Server

*Parcel* [2] ist ein moderner *Web-Bundler*, der dazu entwickelt wurde, Entwicklungsprozesse zu vereinfachen und die Erstellung von Webprojekten zu beschleunigen. Er zeichnet sich durch eine *Zero-Configuration*-Philosophie aus, was bedeutet, dass Entwickler:innen ohne aufwendige Konfigurationsdateien wie bei anderen Bundlern (z.B. Webpack) sofort loslegen können. Parcel bietet automatische Optimierungen und unterstützt eine Vielzahl von Dateitypen, was ihn besonders flexibel macht. Er verfügt über die folgenden Funktionen:

* *Dateibündelung (Bundling):* Der Hauptzweck von Parcel ist es, verschiedene Abhängigkeiten (JavaScript-Dateien, CSS, HTML, Bilder, etc.) in ein einziges, optimiertes Paket zu bündeln. Das Ergebnis ist eine kompakte Datei oder eine Gruppe von Dateien, die im Browser effizienter geladen werden können. Parcel analysiert automatisch alle Abhängigkeiten im Projekt, erstellt eine Abhängigkeitsgrafik und verknüpft die Dateien so, dass sie in der richtigen Reihenfolge geladen werden.
* *Zero Configuration*: Im Gegensatz zu vielen anderen Web-Bundlern benötigt Parcel keine manuelle Konfiguration. Sobald Sie eine Datei wie *index.html* oder *index.js* angeben, durchsucht Parcel automatisch den Quellcode nach Abhängigkeiten, analysiert sie und erstellt ein Bundle. Dies spart enorm viel Zeit und Aufwand für Entwickler, besonders bei kleinen bis mittelgroßen Projekten.
* *Hot Module Replacement (HMR):* Parcel bietet eine eingebaute *Hot Module Replacement*-Funktionalität. Dies bedeutet, dass bei einer Änderung des Quellcodes nur die geänderten Module neu geladen werden, ohne dass die gesamte Seite neu geladen werden muss. HMR verbessert die Entwicklungsproduktivität erheblich, da Änderungen sofort im Browser sichtbar sind.
* *Unterstützung für verschiedene Dateitypen*: Parcel unterstützt eine Vielzahl von Dateitypen, darunter JavaScript, TypeScript, CSS, SASS, SCSS, HTML, Bilder, SVG und mehr. Es erkennt und verarbeitet diese automatisch, ohne dass zusätzliche Plugins erforderlich sind.
* *Optimierung für die Produktion*: Neben der Entwicklungsunterstützung bietet Parcel auch umfangreiche Optimierungen für Produktions-Builds. Dazu gehören die *Minifizierung* von JavaScript und CSS, das *Code-Splitting* zur Verringerung der Dateigröße und die Optimierung von Bildern, um die Ladezeiten zu reduzieren.
* *Parallelverarbeitung und Caching*: Parcel nutzt alle verfügbaren CPU-Kerne zur Parallelverarbeitung, was den Erstellungsprozess erheblich beschleunigt. Zudem implementiert Parcel ein intelligentes Caching-System, wodurch bereits gebaute Module bei späteren Kompilierungen nicht erneut verarbeitet werden müssen, was die Build-Zeiten weiter verkürzt.

Integrierter Lokaler Server

Parcel bietet einen integrierten lokalen Entwicklungsserver, der automatisch startet, wenn Sie ein Projekt mit Parcel ausführen. Dieser Server ermöglicht es, die Webanwendung lokal in einem Browser zu testen, ohne dass Sie dafür manuell einen separaten Server einrichten müssen. Wenn Sie Parcel mit dem folgenden Befehl ausführen, startet es automatisch den lokalen Server:

npx parcel index.html

Dies startet den Entwicklungsserver und öffnet die HTML-Datei im Browser. Parcel überwacht die Dateien in Ihrem Projekt und lädt sie bei Änderungen automatisch neu. Mittels der Funktion • Hot Module Replacement werden Änderungen am Quellcode sofort im Browser sichtbar, ohne die gesamte Seite neu laden zu müssen. Der Parcel-Server ist vor allem für die Entwicklung gedacht. Standardmäßig läuft der Server auf *Port 1234*, aber Sie können den Port ändern, wenn Sie möchten. Parcel bietet auch Optionen, um den Server zu konfigurieren, etwa durch Setzen eines anderen Ports oder Aktivieren von HTTPS. Wenn Sie den lokalen Server auf einem anderen Port starten möchten, können Sie dies folgendermaßen tun:

npx parcel index.html --port 8080

Parcel macht es sehr einfach, einen lokalen Entwicklungsserver zu nutzen, was besonders nützlich ist, um Ihre Webprojekte schnell und effektiv zu testen. Das Ziel ist es, den Prozess der Webentwicklung effizienter und benutzerfreundlicher zu gestalten. Durch die Automatisierung von Konfigurationsaufgaben und die Bereitstellung von Funktionen wie HMR hilft Parcel dabei, den Fokus auf den eigentlichen Entwicklungsprozess zu legen, anstatt sich mit komplizierten Build-Systemen auseinanderzusetzen.

./.

Kommen wir jetzt zum genannten Beispiel:

* Klonen Sie das Repository von [4] auf Ihren lokalen Dateisystem.
* Öffnen Sie das Projekt in Ihrem Editor, beispielsweise Visual Studio Code (*Abbildung 1*).
* Über die Kommandozeile können Sie mittels *npm install* die Abhängigkeiten im Projekt aktualisieren.
* Mittels *npm start* kann das Spiel gestartet werden (*Abbildung 2*). Der lokale Serverlauscht standardmäßig auf der Adresse <http://loacalhost:1234>.

Abbildung\_1.png

Abbildung 1: Das initiale „Hello Word“-Game in Visual Studio Code.

Abbildung\_2.png

Abbildung 2: Ausführung des Hello Word“-Games (Breakout) im lokalen Browser.

Kommen wir jetzt dazu uns mit den wesentlichen Zeilen des Quellcodes vertraut zu machen. Verantwortlich ist die Datei main.ts im Ordner src. Der Quellcode wird in TypeScript geschrieben. Starten wir mit den Imports:

import {

Actor,

CollisionStartEvent,

CollisionType,

Color,

Engine,

vec,

} from "excalibur";

Die Zeile importiert mehrere Klassen und Funktionen aus der *Excalibur.js*-Bibliothek, die für die Entwicklung von 2D-Spielen verwendet werden:

* *Actor*: *Actor* ist eine der zentralen Klassen und repräsentiert ein Objekt im Spiel, z.B. den Spielercharakter, Gegner oder statische Objekte. Ein *Actor* kann Positionen, Bewegungen, Animationen und Kollisionsverhalten haben. Beispiel: Ein Spieler könnte als Actor modelliert werden.
* *CollisionStartEvent*: Dieses Event wird ausgelöst, wenn eine Kollision zwischen zwei *Actor*-Objekten beginnt. Es ist nützlich, um zu definieren, was passieren soll, wenn Objekte miteinander kollidieren, wie z.B. das Abziehen von Leben oder das Triggern eines Sounds. Beispiel: Wenn der Spieler mit einem Gegner kollidiert, könnte dieses Event verwendet werden, um den Schaden zu berechnen.
* *CollisionType:* Diese Klasse definiert verschiedene Typen von Kollisionen für *Actor*-Objekte. *Excalibur.js* bietet verschiedene *CollisionTypes*, wie *Passive* (keine physische Kollision) oder *Active* (Kollision wird berechnet). Dadurch kann das Verhalten eines Objekts bei Kollisionen festgelegt werden. Beispiel: Ein Objekt mit *CollisionType.Active* interagiert physikalisch mit anderen Objekten, während ein *Passive*-Objekt nur Kollisionen meldet, aber nicht darauf reagiert.
* *Color*: Diese Klasse wird verwendet, um Farben in *Excalibur.js* zu definieren. Sie ermöglicht das Festlegen von Farben für verschiedene Objekte und Hintergründe. Excalibur unterstützt sowohl Standardfarben als auch benutzerdefinierte RGB-Farben. Beispiel: *Color.Red* definiert eine rote Farbe, die für Objekte wie Spieler oder Gegner verwendet werden kann.
* Engine: Die Engine ist das Kernstück der Spielumgebung. Sie verwaltet den Spielzustand, die Spielschleife (*Update/Render-Zyklen*) und die Interaktion zwischen Objekten. Die Engine initialisiert das Spiel und steuert die gesamte Simulation. Beispiel: Durch die Initialisierung einer *Engine*-Instanz starten Sie das Spiel und fügen Akteure und Szenen hinzu.
* *vec*: Dieser Eintrag steht für Vektoroperationen und wird verwendet, um Positionen, Bewegungen und physikalische Eigenschaften von Objekten im Raum zu berechnen. Mit Vektoren können Sie beispielsweise die Geschwindigkeit oder Richtung eines Akteurs definieren. Beispiel: *vec(10, 20)* erzeugt einen Vektor, der die Position oder Bewegung eines Objekts im 2D-Raum beschreibt. Weiter geht es mit den folgenden Quellcodeabschnitt:

const game = new Engine({

  width: 800,

  height: 500,

});

Erstellt wird eine neue Instanz der *Engine*-Klasse, die das gesamte Spiel verwaltet und steuert. Diese Instanz wird in der Variable *game* gespeichert und mit zwei Parametern *width* und *height* konfiguriert, die die Abmessungen des Spielbildschirms festlegen. Kommen wir zur folgenden Definition eines Spielers:

const paddle = new Actor({

x: 150,

y: game.drawHeight - 40,

width: 200,

height: 20,

color: Color.LightGray,

});

Dieser Befehl erstellt einen neuen *Actor* der als *Schläger (Paddle)* in einem Breakout-ähnlichen Spiel fungieren könnte. Der Schläger wird mit einer Position, Größe und Farbe versehen und später zu der Spielszene hinzugefügt.

Mit der folgenden Quellcodezeile wird das Kollisionsverhalten definiert:

paddle.body.collisionType = CollisionType.Fixed;

Dies legt den Kollisionstyp des Schlägers fest. *CollisionType.Fixed* bedeutet, dass der Schläger unbeweglich ist und nicht durch andere Objekte verschoben werden kann, aber dennoch an Kollisionen teilnimmt. Mit diesem Kollisionstyp wird der Schläger wie ein Objekt mit unendlicher Masse behandelt, das andere bewegliche Objekte beeinflusst, aber selbst nicht bewegt wird. Dies ist ideal für den Schläger in einem Spiel wie Breakout, da der Spieler den Schläger steuert und dieser beim Aufprall mit dem Ball nicht verschoben werden soll. Das Hinzufügen des Schlägers zur Spielszene erfolgt mittels:

game.add(paddle);

Dies ist notwendig, damit der Schläger auf dem Bildschirm angezeigt und Teil der Spielwelt wird. Ohne diese Zeile würde der Schläger im Code existieren, aber nicht im Spiel sichtbar sein oder interagieren. Kommen wir zu den Paddle-Bewegung:

game.input.pointers.primary.on("move", (evt) => {

paddle.pos.x = evt.worldPos.x;

});

Dieser Abschnitt verbindet die Bewegung der Maus (oder eines Touch-Ereignisses) mit der Position des Paddles. Immer wenn der Spieler die Maus bewegt, wird die X-Position des Paddles auf die X-Koordinate der Maus gesetzt.

Wir benötigen einen Ball:

const ball = new Actor({

x: 100,

y: 300,

radius: 10,

color: Color.Red,

});

Es wird ein Ball erstellt, der als ein *Actor* mit einem kreisförmigen Gebilde und einem *Radius* von 10 dargestellt wird. Der Ball startet an den Koordinaten *(100, 300)* mit einer *roten* Farbe:

setTimeout(() => {

ball.vel = ballSpeed;

}, 1000);

Mit *setTimeout* wird nach *1* Sekunde die Geschwindigkeit des Balls gesetzt. Der Vektor *ballSpeed* gibt die Richtung und Geschwindigkeit an, in der sich der Ball bewegt. In diesem Fall *vec(100, 100).* Das Kollisionsverhalten des Balls definieren wir wie folgt:

ball.body.collisionType = CollisionType.Passive;

Der Kollisionstyp des Balls wird auf *CollisionType.Passive* gesetzt, was bedeutet, dass der Ball Kollisionen erkennt, aber Excalibur wird nicht automatisch die physikalische Reaktion berechnen. Dies ermöglicht es, benutzerdefinierte Kollisionseffekte zu definieren, wie das Umkehren der Bewegungsrichtung.

Wir benötigen eine Erkennung der Bildschirmkollision des Balls:

ball.on("postupdate", () => {

if (ball.pos.x < ball.width / 2) {

ball.vel.x = ballSpeed.x;

}

if (ball.pos.x + ball.width / 2 > game.drawWidth) {

ball.vel.x = ballSpeed.x \* -1;

}

if (ball.pos.y < ball.height / 2) {

ball.vel.y = ballSpeed.y;

}

});

Diese Funktion überprüft, ob der Ball mit den Rändern des Bildschirms kollidiert. Wenn dies der Fall ist, wird die Geschwindigkeit in der entsprechenden Achse (*x* oder *y*) umgekehrt, so dass der Ball zurückprallt.

Ebenso benötigen wir Ziegelsteine:

for (let j = 0; j < rows; j++) {

for (let i = 0; i < columns; i++) {

bricks.push(

new Actor({

x: xoffset + i \* (brickWidth + padding) + padding,

y: yoffset + j \* (brickHeight + padding) + padding,

width: brickWidth,

height: brickHeight,

color: brickColor[j % brickColor.length],

})

);

}

}

Hier werden die Ziegelsteine durch Schleifen erstellt. Es gibt 5 Spalten und 3 Reihen. Jeder Ziegel wird mit einem *Offset* und einer spezifischen Farbe platziert. Wir prüfen, ob der Ball mit den Ziegelsteinen kollidiert:

ball.on("collisionstart", function (ev: CollisionStartEvent) {

if (bricks.indexOf(ev.other) > -1) {

ev.other.kill();

}

var intersection = ev.contact.mtv.normalize();

if (!colliding) {

colliding = true;

if (Math.abs(intersection.x) > Math.abs(intersection.y)) {

ball.vel.x \*= -1;

} else {

ball.vel.y \*= -1;

}

}

});

Diese Funktion überprüft, ob der Ball mit einem der Ziegelsteine kollidiert. Wenn dies der Fall ist, wird der Ziegelstein aus der Szene entfernt (*ev.other.kill()*). Die Bewegungsrichtung des Balls wird anhand der Kollision umgekehrt, je nachdem, welche Achse des Kollisionsvektors (x oder y) größer ist.

Wenn der Ball den Bildschirm verlässt (am unteren Rand), wird eine Nachricht angezeigt, die den Spieler darüber informiert, dass er verloren hat:

ball.on("exitviewport", () => {

alert("You lose!");

});

Das war die Kommentierung des „Hello World“-Games (Breakout), welches als Einstieg in die Programmierung mit Excalibur.js auf der Webseite der Spiele-Engine zum Download bereitsteht. Wir werden nun nachfolgend (und in den anderen Teilen) die einzelnen Elemente eines Spiels näher beleuchten und diese an eigenen kleinen Beispielen einsetzen. Starten wir mit einer genaueren Betrachtung der Game Loop.

Game Loop

In Excalibur.js (und vielen anderen Spiele-Engines) bildet die Spielschleife (*Game Loop*) das Herzstück der Spielmechanik. Die Spielschleife ist ein sich ständig wiederholender Zyklus, der zwei Hauptaufgaben hat, den Spielzustand aktualisieren (*Update*) und das Spielrendering (*Render*) durchzuführen. Dabei wird der Prozess so lange wiederholt, bis das Spiel endet oder pausiert wird. Jeder Zyklus wird als *Frame* bezeichnet, und die Anzahl der *Frames pro Sekunde (FPS)* ist ein wichtiges Merkmal für die Dynamik des Spiels.

* *Update-Phase*: In dieser Phase wird der Status des Spiels aktualisiert. Alle Akteure (wie Spielercharaktere, Gegner, Objekte) führen in dieser Phase ihre Aktionen aus. Hier wird auch die Physik berechnet, Eingaben vom Spieler verarbeitet und Kollisionen überprüft. Typische Aufgaben in der Update-Phase: Bewegung von Akteuren, Physikberechnungen (Schwerkraft, Kollisionen etc.), Aktualisierung von Animationen und das Verarbeiten von Benutzereingaben (Tastatur, Maus etc.) *Excalibur.js* bietet eine Methode namens *update(),* die für jedes Objekt, wie z.B. Akteure, verwendet wird, um ihren Status auf jedem Frame zu aktualisieren.
* *Render-Phase*: Nach dem Update wird in der Render-Phase der aktuelle Spielzustand gezeichnet. Das bedeutet, dass alle grafischen Elemente wie Akteure, Hintergründe, Sprites und UI-Elemente basierend auf ihrem aktuellen Status auf den Bildschirm gezeichnet werden. In Excalibur.js geschieht dies mithilfe einer Canvas-Rendering-API. Hier wird der Bildschirm neu „gezeichnet“, um die Position, Farbe und Form aller Objekte entsprechend dem aktualisierten Status darzustellen. Die Engine selbst optimiert dabei das Zeichnen, indem sie z.B. nur sichtbare Objekte rendert.

Rendering in Excalibur.js

Das Rendering bezieht sich auf das Zeichnen von grafischen Elementen auf den Bildschirm. Excalibur.js verwendet für das Rendering das *HTML5-Canvas*-Element. Die Engine verwaltet dabei die Zeichenoperationen über ein eigenes Rendering-System, das auf Leistung optimiert ist.

* *Zeichenoperationen*: Excalibur.js verwendet verschiedene Zeichenoperationen, um Akteure und andere Spielobjekte auf der Leinwand (Canvas) darzustellen. Jede Zeichenoperation entspricht einem Frame des Spiels, in dem alle sichtbaren Elemente auf den Bildschirm gezeichnet werden.
* *Sprites und Animationen*: Excalibur.js ermöglicht die Verwendung von Sprites, um Animationen zu erzeugen. Sprites sind kleine Bilder, die in einer Sequenz abgespielt werden können, um Bewegungen oder Aktionen darzustellen. Die Engine kümmert sich um das Rendern dieser Animationen basierend auf den Aktualisierungen, die während der Update-Phase vorgenommen wurden.

Zusammenspiel von Spielschleife und Rendering

Die beiden Phasen – *Update* und *Render* – laufen innerhalb der Spielschleife kontinuierlich ab. Pro Zyklus wird zuerst der Status aller Akteure und Objekte aktualisiert, und anschließend wird der neue Zustand gerendert, so dass der Spieler die Veränderungen auf dem Bildschirm sieht (*Abbildung 3*). Dies geschieht so schnell, dass es für das Auge wie eine flüssige Bewegung aussieht. Excalibur.js handhabt diesen Prozess automatisch, indem die *Engine.start()*-Methode aufgerufen wird, die die Spielschleife startet. Die Spielschleife läuft so lange, bis das Spiel beendet wird. Der Code für ein einfaches Beispiel kann wie in *Listing 1* dargestellt aussehen. In diesem Beispiel übernimmt Excalibur.js das Update und Rendern des Akteurs automatisch in jeder Schleifeniteration.

Abbildung\_3.png

Abbildung 3: Die Game Loop von Excalibur.js [5].

Nachdem einige technische Grundlagen behandelt wurden, ist es nun wichtig, ein Blick auf die konzeptionellen Elemente der Spieleentwicklung zu werfen. Denn während eine saubere und effiziente Programmierung das Rückgrat eines Spiels bildet, sind es die kreativen Entscheidungen im Game-Design, die das Spielerlebnis prägen. Es gilt: Auch „Mini-Spiele“ brauchen eine Spielidee.

Grundpfeiler der Game-Design-Konzeption

Die Entwicklung von Computerspielen beginnt mit einem soliden konzeptionellen Fundament, das den kreativen und strukturellen Rahmen für das gesamte Spiel liefert. Die ersten Schritte in diesem Prozess beinhalten die Entwicklung einer Spielidee, die Ausarbeitung des Gameplays, die Gestaltung der Geschichte, das Level-Design und die Auswahl der Zielgruppe und Plattform. Diese konzeptionellen Grundlagen beeinflussen alle weiteren Entwicklungsaspekte, von der Programmierung bis zur Veröffentlichung. Im Folgenden werden die fünf zentralen Elemente dieser frühen Phase detailliert erläutert.

Spielidee und Konzept

Die *Spielidee* bildet den Ausgangspunkt der Entwicklung. Sie legt fest, was das Spiel einzigartig macht und warum Spieler es spielen wollen. Die Idee kann aus einer Vielzahl von Quellen stammen: bestehenden Spielen, Literatur, Filmen oder der Vorstellungskraft des Entwicklers. Wichtig ist, dass die Spielidee innovativ genug ist, um Interesse zu wecken, und gleichzeitig realistisch in der Umsetzung bleibt. Das *Spielkonzept* geht über die bloße Idee hinaus und beschreibt das Ziel des Spiels sowie die Mittel, mit denen der Spieler dieses Ziel erreicht. Hier wird auch die Art der Interaktion definiert: Handelt es sich um ein Actionspiel, ein Puzzlespiel oder ein Rollenspiel? Die Spielwelt und die Atmosphäre werden ebenfalls in dieser Phase festgelegt, da sie maßgeblich zur Immersion beitragen. Das Setting (z.B. Fantasy, Science Fiction oder historische Umgebungen) beeinflusst die Ästhetik, die erzählte Geschichte und das Spielerlebnis. Ein wichtiger Bestandteil der Spielidee ist das *Spielziel*. Es bestimmt, worauf der Spieler hinarbeitet. Ob es darum geht, Feinde zu besiegen, Rätsel zu lösen oder einfach nur Punkte zu sammeln – das Ziel definiert die Motivation des Spielers und die Struktur des Spiels.

Gameplay und Spielmechanik

Das *Gameplay* beschreibt, wie der Spieler mit dem Spiel interagiert und welche Aktionen er ausführen kann. Es umfasst die Kernmechaniken des Spiels, die wiederkehrenden Handlungen, die der Spieler ausführt. In einem Actionspiel könnten dies z.B. Laufen, Springen und Schießen sein, während es in einem Puzzlespiel um das Verschieben von Objekten geht. Die Spielmechaniken müssen sorgfältig gestaltet werden, um ein ausgewogenes, fesselndes und unterhaltsames Spielerlebnis zu bieten. Das *Regelsystem* legt fest, wie das Spiel funktioniert. Es definiert, was der Spieler tun kann und welche Einschränkungen es gibt. Diese Regeln sind oft mit den Spielmechaniken verknüpft und sorgen dafür, dass das Spiel konsistent und fair bleibt. Ein Beispiel für eine Regel wäre, dass der Spieler nur eine bestimmte Anzahl von Leben hat oder dass er in bestimmten Bereichen des Spiels nicht speichern kann. Ein weiteres wichtiges Element ist die *Schwierigkeitskurve*. Diese beschreibt, wie die Herausforderung des Spiels im Laufe der Zeit ansteigt. Ein gut gestaltetes Spiel beginnt oft mit einfachen Aufgaben und steigert die Komplexität allmählich, um den Spieler zu fordern, ohne ihn zu überfordern. Eine ausgewogene Schwierigkeitskurve trägt dazu bei, die Motivation des Spielers aufrechtzuerhalten und ihm das Gefühl zu geben, Fortschritte zu machen.

Narrative und Storytelling

Die Geschichte eines Spiels ist ein zentrales Element, das die Spielerbindung stärkt. Sie gibt dem Spieler einen Kontext und eine Motivation für seine Handlungen. Während einige Spiele wie Puzzles oder Plattformspiele auf eine komplexe Handlung verzichten können, sind narrative Spiele wie Rollenspiele oder Abenteuerspiele stark auf eine fesselnde Story angewiesen. Die Geschichte kann linear oder nicht-linear erzählt werden. In einem linearen Spiel folgt der Spieler einer festen Handlung, während in einem nicht-linearen Spiel Entscheidungen getroffen werden können, die den Verlauf der Geschichte beeinflussen. Diese Art von Erzählung erhöht die Wiederspielbarkeit eines Spiels und schafft ein dynamischeres Erlebnis. Ein wichtiger Aspekt des Storytellings ist die *Immersion*. Sie beschreibt den Grad, in dem der Spieler in die Spielwelt eintauchen kann. Um eine immersive Erfahrung zu schaffen, müssen die narrative Struktur, die visuelle Gestaltung und das Gameplay eng miteinander verwoben sein. Der Spieler sollte das Gefühl haben, dass seine Handlungen Auswirkungen auf die Welt haben und dass er ein aktiver Teil der Geschichte ist. Auch die *Charakterentwicklung* spielt in vielen Games eine wichtige Rolle. Spieler können sich mit dem Protagonisten identifizieren, wenn dieser eine nachvollziehbare Entwicklung durchläuft. Dies kann durch die Handlung selbst oder durch das Fortschrittssystem im Spiel geschehen, bei dem der Charakter neue Fähigkeiten erlernt und sich an die Herausforderungen der Spielwelt anpasst.

Level-Design und Progression

Das *Level-Design* bezieht sich auf die Gestaltung der Spielumgebung und die Art und Weise, wie der Spieler durch das Spiel geführt wird. In einem gut gestalteten Level wird der Spieler durch visuelle Hinweise und Herausforderungen motiviert, die Spielwelt zu erkunden und Aufgaben zu bewältigen. Das Level-Design trägt maßgeblich dazu bei, wie spannend und interessant das Spiel für den Spieler bleibt. Eine wichtige Komponente des Level-Designs ist der *Flow*. Der Begriff beschreibt, wie reibungslos der Spieler von einer Aufgabe zur nächsten gelangt, ohne sich frustriert oder verloren zu fühlen. Ein gut durchdachter Level sorgt dafür, dass der Spieler immer weiß, was zu tun ist, ohne dabei in Routine zu verfallen. Auch die Platzierung von Rätseln, Gegnern und Belohnungen spielt hier eine Rolle. Die *Progression* beschreibt den Fortschritt des Spielers im Spiel. Diese kann durch das Absolvieren von Levels, das Sammeln von Gegenständen oder das Erreichen bestimmter Ziele dargestellt werden. Die Art und Weise, wie sich das Spiel entwickelt und wie der Spieler Fortschritte macht, trägt wesentlich zur Langzeitmotivation bei. Ein Beispiel für eine gelungene Progression ist ein Rollenspiel, bei dem der Spieler nach und nach neue Fähigkeiten freischaltet und seine Spielfigur verbessert.

Zielgruppe und Plattform

Die Definition der *Zielgruppe* ist ein entscheidender Schritt in der Konzeption eines Spiels. Die Frage, für wen das Spiel entwickelt wird, beeinflusst nahezu alle Aspekte des Designs, von der Spielmechanik bis zur Grafik. Ein Spiel für eine jüngere Zielgruppe wird andere Anforderungen haben als ein Strategiespiel für erfahrene Spieler. Aspekte wie die Altersgruppe, Vorlieben und das Spielverhalten der Zielgruppe bestimmen, wie zugänglich oder herausfordernd ein Spiel gestaltet wird. Auch die *Plattformwahl* ist von zentraler Bedeutung. Jedes Spiel muss an die technischen und interaktiven Möglichkeiten der Plattform angepasst werden, für die es entwickelt wird. Ob für PC, Konsole oder Mobile – die Plattform beeinflusst die Steuerungsmöglichkeiten, die Grafikleistung und sogar die Monetarisierungsstrategien. Spiele für mobile Plattformen haben oft ein einfacheres Interface und kürzere Spielsessions im Vergleich zu Konsolen- oder PC-Spielen, die auf längere, intensivere Spielerlebnisse ausgelegt sind.

Zusammenfassend lässt sich sagen, dass die Konzeption eines Computerspiels eine durchdachte Planung und Abstimmung vieler kreativer und technischer Elemente erfordert. Die Spielidee und das Gameplay legen den Grundstein, während die narrative Tiefe, das Level-Design und die Zielgruppenorientierung sicherstellen, dass das Spiel sowohl fesselnd als auch ansprechend ist. Diese konzeptionellen Grundlagen sind entscheidend für den Erfolg eines Spiels und bestimmen maßgeblich die Spielerfahrung. Auch wenn wir mit den Experimenten mit der Spiele-Engine (zunächst) keine ganz großen Games entwickeln wollen, können uns die Basiselemente des Game-Design eine gewisse Orientierung geben.

Templates

Nochmals zurück zur Programmierung: Unter [5] finden wir ein Template, um ein eigenes Spiel zu entwickeln. Es enthält die grundlegende Ordnerstruktur und die wesentlichen Abhängigkeiten, u.a. den o.g. *Parcel-Bundler*. Wenn Sie dieses Repository klonen (oder herunterladen) und mittels:

npm install

die Abhängigkeiten aktualisieren, bekommen Sie ein initiales Projekt für Ihre ersten Experimente (*Abbildung 4*). Mit:

npm start

können Sie das Spiel auf der Basis des Template starten (*Startseite* und ein *Actor*). Nach und nach können Sie auf der Basis dieses Templates die eigene Spielidee umsetzen.

Abbildung\_4.png

Abbildung 4: Ein Template auf der Basis von Excalibur.js und Parcel.

Fazit

Der Anfang in die Welt der Spieleprogrammierung ist gemacht im zweiten Teil unserer Artikelserie arbeiten wir weiter an den Grundlagen, insbesondere der Darstellung von Grafiken und Sprites. Vielleicht haben Sie bis dahin schon eine erste Idee für ein kleines 2D-Game, welches Sie für die Browser dieser Welt realisieren möchten.

*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://parceljs.org/>

[3] <https://excaliburjs.com/docs/getting-started>

[4] https://github.com/excaliburjs/sample-breakout

[5] <https://github.com/excaliburjs/template-ts-parcel-v2>

[6] <https://medium.com/@razorshark21/understanding-the-fundamental-of-game-design-8ced9daaaa5b>

Listing 1: Ein einfaches Beispiel für eine Game Loop [1].

import { Engine } from 'excalibur';

// Create an instance of the engine.

const game = new Engine({

width: 800,

height: 600,

});

// Start the engine to begin the game.

game.start();

// Create an actor with x position of 150px,

const paddle = new Actor({

x: 150,

y: game.drawHeight - 40,

width: 200,

height: 20,

color: Color.Chartreuse,

});

paddle.body.collisionType = CollisionType.Fixed;

game.add(paddle);

// Add a mouse move listener

game.input.pointers.primary.on("move", (evt) => {

paddle.pos.x = evt.worldPos.x;

});

./.