Spaß im Web – Freude beim programmieren

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

Im letzten Teil haben wir uns mit Actors, Scenes, Sprites und der grafikbasierten Textdarstellung beschäftigt. Weitere wichtige Elemente, um ein browserbasiertes Game zu erstellen, sind Tilemaps (Karten), Physiksimulationen, Kollisionen und Interaktionen. Damit haben wir hoffentlich ein spannendes Programm für diesen dritten Teil ausgewählt.

von Dr. Veikko Krypczyk

Um Sie liebe Leser wieder ins Boot zu holen, starten wir unsere Reise für diesen Artikel wieder mit einem Beispiel aus dem letzten Teil. Wir laden den Quellcode des Beispiels *Sprite* in Visual Studio Code und starten es über die Kommandozeile, d.h. 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: Darstellung eines Sprite.

Das ist eine typische Szene in einem 2D-Browser Game: Ein Actor möchte sich durch eine virtuelle Spielewelt bewegen. Diese Spielewelt wird meist in Form einer 2D-Grafik erstellt. Wir bezeichnen diese als *Tilemap*.

Überblick über die Artikelserie

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

Teil 2: Fortgeschrittene Grafiken und Sprites

**Teil 3: Tilemaps, Physik, Kollision und Interaktionen**

Teil 4: Erweiterte Techniken.

./.

Tilemaps – Konzept

*Tilemaps* sind ein zentraler Bestandteil vieler 2D-Spiele und werden verwendet, um große, detaillierte Spielwelten zu erstellen, ohne dass viel Speicherplatz benötigt wird. Sie bestehen aus vielen kleinen, wiederverwendbaren grafischen Elementen, den sogenannten *Kacheln* oder *Tiles*. Diese Tiles repräsentieren einzelne Elemente einer Karte, wie beispielsweise Bodenflächen, Wände, Hindernisse oder andere Umgebungsdetails. Durch das systematische Platzieren und Kombinieren dieser Kacheln in einem Rastermuster entsteht eine visuell zusammenhängende Spielwelt. Eine Tilemap ist in der Regel als ein *Raster* strukturiert, bei dem jede Zelle ein *Tile* darstellt. Die wichtigsten Bestandteile einer Tilemap sind:

* *Tiles*: Kleine Bilder oder Grafiken (beispielsweise 32x32 Pixel), die spezifische Umgebungsmerkmale darstellen.
* *Tile-Ebenen (Layers):* Häufig sind Tilemaps in verschiedene Ebenen unterteilt, um beispielsweise Hintergründe, Kollisionsebenen, interaktive Bereiche oder Vordergründe separat zu verwalten. Das erlaubt eine detaillierte Strukturierung der Karte.
* *Tile-Index:* Jedes Tile hat in einer Tilemap eine Position (beispielsweise `(x, y)`) und wird oft mit einem Index versehen, der auf die entsprechende Grafik verweist.

Tilemaps eignen sich besonders für Spiele, die wiederkehrende grafische Muster benötigen, wie Plattformspiele, Rollenspiele und Puzzle-Spiele. Durch das Wiederverwenden von Kacheln können große Karten erstellt werden, ohne dass viel Speicherplatz verbraucht wird. Häufig wird auch eine *Kollisionsschicht* hinzugefügt, die definiert, welche Tiles begehbar sind und welche nicht. In Rollenspielen beispielsweise können bestimmte Tiles interaktiv sein (beispielsweise Türen oder Truhen), die das Gameplay erweitern.

Erstellung einer Tilemap (Tilemap-Editor)

Um eine Tilemap zu erstellen, wird oft ein *Tilemap-Editor* verwendet. Diese Editoren bieten eine visuelle Oberfläche, auf der Entwickler Tiles auswählen und auf ein Raster platzieren können, um eine Karte zu erstellen. Der Editor speichert die Karte dann als Datenstruktur (oft in XML, JSON oder anderen Formaten), die das Spiel interpretieren kann. Einige bekannte *Tilemap-Editoren* sind:

* *Tiled [2]:* Einer der beliebtesten Open-Source-Editoren für 2D-Maps. Tiled unterstützt mehrere Ebenen, benutzerdefinierte Kollisionen, Objekte und das Exportieren in gängige Formate wie JSON, XML und mehr. Tiled ist ideal für eine Vielzahl von 2D-Spielen und bietet Flexibilität und eine breite Community.
* *Aseprite [3]:* Vor allem für Pixel Art bekannt, kann Aseprite ebenfalls Tilemaps erstellen, eignet sich jedoch mehr für einfache Tilemaps und weniger komplexe Karten.
* *Unity Tilemap Editor [4]:* Für Spieleentwickler, die mit Unity arbeiten, bietet Unity einen eingebauten Tilemap-Editor. Mit ihm können Karten direkt in der Unity-Umgebung erstellt werden, und er unterstützt komplexe Kollisionen, interaktive Tiles und viele weitere Optionen.

Die Dokumentation unserer Spiele-Bibliothek empfiehlt die Nutzung von *Tiled*. Wir haben im Textkasten einige Informationen zu diesem Tool zusammengestellt.

Tiled – Ein kompakter Überblick

Der Tiled Editor ist ein Open-Source-Tool zur Erstellung von 2D-Tilemaps und wird häufig von Spieleentwickler:innen verwendet, um Karten für Rollenspiele und andere 2D-Spiele zu erstellen. *Tiled* bietet eine benutzerfreundliche Oberfläche und leistungsfähige Funktionen, mit denen komplexe, detailreiche Spielkarten erstellt und konfiguriert werden können. Die Hauptfunktionen des Tiled Editors sind:

* *Mehrere Ebenen (Layers):* Tiled unterstützt verschiedene Ebenen, wie beispielsweise Kachelebenen, Objektebenen und Bild-Layer. Damit können Kartenstrukturen in mehrere Schichten unterteilt werden, wie beispielsweise Hintergründe, Vordergrundobjekte und Kollisionsobjekte. Die Kachelebenen (*Tile Layers*) sind ideal für den Aufbau der eigentlichen Karte, während Objektebenen (*Object Layers*) für interaktive oder kollidierbare Elemente verwendet werden. *Tilesets* können in Tiled importiert und für verschiedene Karten verwendet werden. Ein *Tileset* ist ein Bild mit mehreren kleinen Grafiken (Kacheln), die als einzelne Bestandteile der Karte platziert werden können. Tiled unterstützt auch animierte Tiles und bietet eine flexible Anpassung der Tilegrößen und Abstände.
* *Kollisions- und Objekt-Definitionen:* In Tiled können Kollisionen direkt auf die Kacheln angewendet werden, so dass eine Kachel als „kollidierbar“ markiert wird. Dies ermöglicht es, eine Karte mit Hindernissen zu versehen, die dann im Spiel automatisch erkannt werden können. Außerdem können in den Objektebenen spezielle Eigenschaften definiert werden, wie beispielsweise Trigger oder Ereignisse, die im Spiel aufgerufen werden können.
* *Unterstützung von Tile-Objekten und benutzerdefinierten Eigenschaften:* Tiled erlaubt das Hinzufügen von Objekten, die individuell auf der Karte platziert werden können und beispielsweise als Startpositionen, Sammelobjekte oder Trigger-Punkte fungieren können. Benutzerdefinierte Eigenschaften können den Tiles hinzugefügt werden, wie etwa „interaktiv: ja“, „Schaden: 10“, um spezifische Aktionen zu definieren.
* *Export in verschiedene Formate:* Tiled kann die erstellten Karten in verschiedenen Formaten exportieren, darunter JSON, TMX (das native XML-Format von Tiled), CSV und andere. JSON und TMX sind häufig genutzte Formate, da sie leicht von Spiel-Engines und Bibliotheken verarbeitet werden können.

./.

Um eine Tilemap zu erstellen, werden üblicherweise die folgenden Schritte durchlaufen:

1. *Tileset vorbereiten*: Eine Sammlung von wiederverwendbaren Tiles (beispielsweise Gras, Wasser, Wand) wird erstellt. Diese Tiles sind häufig in einer einzigen Bilddatei gespeichert, dem sogenannten *Spritesheet*.
2. *Tiles platzieren:* Der Entwickler wählt Tiles aus und platziert sie auf einem Raster, um die Weltkarte des Spiels zu gestalten.
3. *Layer hinzufügen*: Je nach Bedarf werden mehrere Ebenen erstellt – beispielsweise eine Hintergrundebene, eine Kollisionsebene und eine Ebene für interaktive Objekte.
4. *Exportieren*: Die Tilemap wird in einem Format exportiert, das das Spiel verarbeiten kann, beispielsweise JSON oder XML.

Viele bekannte und auch weniger bekannte Spiele nutzen Tilemaps um die Welt des Spiels zu gestalten. Einige Beispiele:

* *Super Mario Bros*: Verwendet Tilemaps, um seine Level zu erstellen, wobei jedes Tile ein Element wie einen Block, eine Plattform oder ein Hindernis darstellt.
* *The Legend of Zelda*: Die Oberwelt und Dungeons sind aus Tiles zusammengesetzt, was die Spielwelt zusammenhängend und trotzdem speicherfreundlich macht.
* *Stardew Valley:* Nutzt Tilemaps für die Bauernhoflandschaft und andere Kartenbereiche. Durch das Hinzufügen von Schichten und Objekten bietet die Welt Komplexität und Tiefe.

Installieren Sie das Programm Tiled und sehen sich die beigefügten Beispiele aus dem Installationsordner an (*Abbildung 2*).

Abbildung\_2.png

Abbildung 2: Beispiel für eine Tilemap im Programm Tiled.

Experimentieren Sie gern auch etwas mit einer neuen Map. Beispielsweise platzieren Sie einige Tiles und exportieren danach die Datei im TMX-Format. Eine neue Tilemap erstellen Sie beispielsweise mit den folgenden Schritten:

1. *Tileset erstellen und importieren:* Bereiten Sie ein Tileset vor, eine Bilddatei (beispielsweise *tileset.png*), die alle benötigten Kacheln enthält, wie Gras, Wasser, Wände etc. Hinweis: Im Internet finden sich unzählige Ressourcen in Form von Tiles, Tilemaps und Bilddateien, welche explizit für Spiele-Entwickler zur Verfügung stehen. Es gibt zahlreiche kostenfreie und kostenpflichtige Ressourcen. Öffnen Sie Tiled und wählen Sie den Menüpunkt File | New Tileset aus, um das Tileset zu importieren. Geben Sie die Kachelgröße an, beispielsweise 32x32 Pixel, die mit dem Tileset übereinstimmt.
2. *Neue Karte erstellen*: Erstellen Sie eine neue Karte in Tiled über File > New Map und geben Sie die Kartenabmessungen (beispielsweise 20x20 Tiles) und die Kachelgröße (beispielsweise 32x32 Pixel) an. Wählen Sie das zuvor erstelltes Tileset als Quelle für die Kacheln auf der Karte.
3. *Tiles platzieren:* Verwenden Sie das Stiftwerkzeug in Tiled, um die Kacheln aus dem Tileset auf das Raster der Karte zu zeichnen und die gewünschte Welt oder Struktur zu erstellen (beispielsweise ein Boden aus Gras, Wände, Pfade).
4. *Optional:* Erstellen Sie zusätzliche Layer (Schichten) für komplexere Strukturen (beispielsweise ein Hintergrund- und ein Kollisionselement).

Wenn Sie nur wenige Tiles auf der Map angeordnet haben, dann können Sie diese in der Datenmatrix gut ausmachen (*Abbildung 3*). Das ist wahrscheinlich der beste Weg, um die Arbeit mit den Tilemap-Dateien zu verstehen.

Abbildung\_3.png

Abbildung 3: Erstellung einer Tilemap in Tiled und Anzeige der generierten TMX-Datei.

Die Dateien eines Tileset

Bei der Erstellung einer Tilemap mit Tiled kommen in der Regel mehrere Dateiformate zusammen, um die Struktur, die verwendeten Grafiken und die Kollisionseigenschaften zu definieren. Die wichtigsten Formate dabei sind:

* *TMX-Datei:* Die eigentliche Karte, die die Anordnung der Tiles und ihre Eigenschaften speichert.
* *TSX-Datei:* Das Tileset, das die Grafiken und Informationen zu den einzelnen Tiles enthält.
* *PNG-Datei:* Die Bilddatei, die die visuellen Ressourcen für das Tileset bereitstellt.

Diese Dateiformate arbeiten zusammen, um eine vollständige Tilemap zu erzeugen, die später in einer Game-Engine genutzt werden kann.

Die *TMX-Datei (Tile Map XML)* ist das Hauptdokument für die Tilemap. Das TMX-Format [4] ist das native Datenformat des Tiled Editors und wird verwendet, um 2D-Karten und Tilemaps zu speichern. Es basiert auf XML und speichert detaillierte Informationen über die Struktur und die Inhalte der Karte, einschließlich der verwendeten Tilesets, Ebenen und speziellen Objekteigenschaften. Eine TMX-Datei besteht aus mehreren Hauptabschnitten, die jeweils unterschiedliche Informationen zur Karte und ihren Bestandteilen enthalten (Beispiel in *Listing 1*):

* *Map-Element:* Es definiert grundlegende Eigenschaften der Karte, wie die Breite und Höhe der Karte, die Größe jedes Tiles und die verwendete Ausrichtung.
* *tileset-Tag*: Verweist auf die TSX-Datei, welche die Kacheln enthält (Tileset), sowie auf die ID und die Größe der einzelnen Kacheln.
* *Layer (Ebenen):* Jede Tile-Ebene (*<layer>*) definiert eine Schicht der Karte, die entweder aus sichtbaren Tiles, Hintergründen oder kollidierbaren Objekten bestehen kann. Die Ebenen enthalten eine *<data>*-Sektion, die Informationen über die Anordnung der Tiles auf dieser Ebene enthält. Die Daten können als CSV, Base64 oder einfache XML-Tags gespeichert werden.
* *Object Layers:* Objekt-Ebenen *(<objectgroup>*) werden verwendet, um interaktive Elemente auf der Karte zu platzieren, wie Startpunkte oder Trigger-Zonen. Jedes Objekt hat eine eindeutige ID, Position und optional eine Form, beispielsweise ein Rechteck oder ein Polygon. Objekte können auch benutzerdefinierte Eigenschaften enthalten, die im Spiel genutzt werden können.
* *Properties*: Das Properties-Tag *(<properties>*) wird verwendet, um benutzerdefinierte Eigenschaften hinzuzufügen. Eigenschaften können fast überall in einer TMX-Datei definiert werden, d.h. auf Ebenen und Objekten. Sie erlauben es, zusätzliche Daten wie „begehbar“ festzulegen.

Die *TSX-Datei (Tile Set XML)* definiert die Eigenschaften des Tilesets und stellt die Verbindung zur *PNG-Datei* her:

* *Referenz auf das Bild (PNG-Datei)*: Die TSX-Datei enthält die Referenz zur *PNG*-Datei, welche die Grafiken der einzelnen Tiles speichert. Dadurch weiß die TMX-Datei, wie die Tiles grafisch dargestellt werden sollen.
* *Tilegrößen und -abstände*: Die TSX-Datei definiert die Größe jedes Tiles (beispielsweise 32x32 Pixel) sowie eventuelle Abstände zwischen den Tiles, falls sie in der PNG-Datei vorhanden sind.
* *Tile-Eigenschaften*: In der TSX-Datei können auch zusätzliche Eigenschaften für jedes Tile definiert werden, wie beispielsweise Kollisionseigenschaften (*solid*), Animationen oder andere benutzerdefinierte Attribute. Wenn diese Eigenschaften im Tileset definiert sind, werden sie in der TMX-Datei einfach vererbt und müssen nicht separat für jedes Tile auf der Karte gespeichert werden.

Die TSX-Datei fungiert als Beschreibung für das Tileset und enthält alle notwendigen Informationen, um die Bilddatei korrekt in der Tilemap darzustellen und zu verwenden.

Die *PNG-Datei* ist die eigentliche Bilddatei, die die grafischen Ressourcen enthält, die für die Tiles verwendet werden. Diese Datei enthält alle visuellen Darstellungen der Tiles in einem Raster:

* *Tile-Grafiken*: Jedes Tile in der PNG-Datei ist ein Bildausschnitt, der in der TSX-Datei durch seine Position und Größe definiert wird.
* *Mehrfache Verwendung von Tiles*: Da die Tiles als Bilder in der PNG-Datei gespeichert sind, kann ein einziges Tile mehrfach in der TMX-Datei verwendet werden, was Speicherplatz spart und die Performance der Engine optimiert.

Die PNG-Datei enthält nur die Grafik und keine Informationen über die Position oder die Eigenschaften der Tiles. Diese Bilddatei wird lediglich als Ressource verwendet, auf die die TSX-Datei verweist.

Das Zusammenspiel von *TMX*-, *TSX*- und *PNG*- ist wie folgt: Die *TSX*-Datei verweist auf die *PNG*-Datei und definiert die Größe und Position der einzelnen Tiles im Bild, sowie eventuelle Eigenschaften (wie „solid“ für Kollisionsabfragen). Die *TMX*-Datei legt das Layout und die Struktur der Karte fest, indem sie das Raster definiert und die Positionen der Tiles in der Karte speichert. Eine Game-Engine wie Excalibur lädt die TMX-Datei als Hauptdatei und analysiert die Referenzen auf die TSX- und PNG-Dateien. So wird die gesamte Karte mit den jeweiligen Eigenschaften und Grafiken geladen. Durch diese klare Trennung der Zuständigkeiten bleibt die Tilemap flexibel und leicht anpassbar, da Änderungen an der Grafik oder den Eigenschaften im Tileset (TSX und PNG) vorgenommen werden können, ohne das Layout der Karte (TMX) anzupassen.

Nutzung von Tilemaps in Excalibur.js

Da Tiled-Maps häufig als JSON- oder TMX-Dateien exportiert werden, benötigen wir in *Excalibur.js* ein Plugin, um diese Dateien zu laden und zu interpretieren. Ein beliebtes Plugin hierfür ist das *Excalibur Tiled Plugin*. Das Excalibur Tiled Plugin ermöglicht das einfache Laden und Verwenden von Tiled-Karten in Excalibur. Das Plugin liest die in Tiled erstellten Karten und wandelt sie in Excalibur-Tilemaps um. Als erstes müssen wir das Plugin für das Projekt installieren. Das erreichen wir über den folgenden Kommandozeilenbefehl:

npm install --save-exact @excaliburjs/plugin-tiled

Jetzt wollen wir die Tilemap anzeigen. Dazu müssen wir die Tilemap verfügbar machen. Es handelt sich um die Datei mit der Extension *\*.TMX*. Diese wiederum verweist auf eine *\*.TSX*-Datei, welche ihrerseits eine Bilddatei, d.h. eine *\*.PNG*-Datei, referenziert. All diese Dateien werden im Projekt über die Datei *resources.ts* verfügbar gemacht (*Listing 2*). Der Quellcode zeigt, wie man in Excalibur.js eine *Tilemap (TMX-Datei)* lädt und dafür das *Parcel-Bundling* nutzt. Da Parcel Ressourcen zur Laufzeit auf dynamische Pfade umleitet, ist die Konfiguration im Code so aufgebaut, dass Excalibur die von Parcel bereitgestellten Dateien korrekt findet und lädt. Mit anderen Worten, damit die Dateipfade zu den Ressourcen korrekt verarbeitet werden, sind einige „Klimmzüge“ notwendig:

* *Loader* und *TiledResource*: *Loader* stammt aus der Excalibur-Bibliothek und ist der Standardlader für Ressourcen. *TiledResource* wird aus dem *@excaliburjs/plugin-tiled*-Plugin importiert, das speziell für das Laden von Tiled Maps (TMX-Dateien) entwickelt wurde.
* *Parcel Resource Imports:* Parcel behandelt bestimmte Dateitypen (beispielsweise Bilder, TMX- und TSX-Dateien) als Module und erstellt zur Laufzeit dynamische Pfade für sie. Durch die *import*-Anweisungen (*import tilesetPath from '../res/elements.png'*) erstellt Parcel Pfadvariablen (*tilesetPath*, *tmxPath*, *tsxPath*), die beim Laden verwendet werden, um sicherzustellen, dass Excalibur auf die korrekten, von Parcel gebündelten Ressourcen, zugreift.
* *TiledMap Resource*: *Resources.TiledMap* erstellt ein neues *TiledResource*-Objekt, das die TMX-Datei lädt. Dabei wird *tmxPath* verwendet, der dynamische Pfad, den Parcel für die TMX-Datei generiert hat.
* *Path Map (pathMap):* Die *pathMap* ist eine spezielle Einstellung im *TiledResource*, die hilft, statische Bundles von Parcel zu korrigieren. Sie stellt sicher, dass alle internen Verweise in der TMX-Datei auf andere Ressourcen (wie das Tileset oder die TSX-Datei) korrekt auf die von Parcel generierten Pfade umgeleitet werden.
* *path*: Dies ist der Originalpfad, der in der TMX-Datei steht, beispielsweise *elements.tmx*, *elements.png* oder *elements.tsx*.
* *output*: Dies ist der von Parcel generierte dynamische Pfad (beispielsweise *tmxPath*, *tilesetPath*, *tsxPath*), der sicherstellt, dass Excalibur die Dateien an der richtigen Stelle findet.
* *Loader erstellen*: Der *Loader* von Excalibur wird instanziiert. Der *Loader* ist dafür verantwortlich, die Ressourcen zu laden, bevor das Spiel startet.
* *Ressourcen hinzufügen*: Mithilfe einer Schleife fügt der Code alle Ressourcen, die im *Resources*-Objekt definiert sind, dem *loader* hinzu. Dadurch wird *loader.addResource(resource)* für jede Ressource aufgerufen, in diesem Fall also für *Resources.TiledMap*. Wenn das Spiel startet, sorgt Excalibur dafür, dass alle Ressourcen im *loader* geladen sind, bevor das Spiel startet.

Warum ist dieser Ansatz wichtig bei der Nutzung von Parcel? Parcel erstellt beim Bauen des Projekts eindeutige Dateinamen und Pfade für jedes Asset, um Namenskonflikte zu vermeiden und Caching zu unterstützen. Das bedeutet, dass statische Pfade, die direkt in TMX-Dateien stehen (wie *elements.png*), nicht mehr korrekt wären, sobald Parcel die Dateien umbenannt hat. Die *pathMap*-Funktion des *TiledResource* löst dieses Problem, indem sie die Originalpfade (wie in der TMX-Datei definiert) auf die dynamischen Parcel-Pfade umleitet. Anstatt statische Pfade manuell anzupassen oder fest in den Code einzutragen, stellt der Ansatz mit Parcel sicher, dass die Pfade zur Laufzeit immer korrekt sind. Das reduziert Fehler und sorgt dafür, dass die Ressourcen auch in Produktionsumgebungen (wo Parcel-Bundles verwendet werden) zuverlässig geladen werden.

Nach dem Laden der Ressourcen kann der *loader* dann verwendet werden, um die Karte und andere Ressourcen vollständig zu laden, bevor das Spiel beginnt. Die zentrale *main.ts-*Datei des Spiels erstellt auf bekannte Art und Weise das Game, lädt die Ressourcen und startet das Spiel:

import \* as ex from 'excalibur';

import { Resources, loader } from './resources';

const game = new ex.Engine({

width: 800,

height: 600,

});

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

Resources.TiledMap.addToScene(game.currentScene);

});

Wenn wir das Spiel dann starten (*npm -start*), dann wird nach dem allgemeinen Bildschirm für den Loader die Tilemap angezeigt, welche wir über die *TMX-, TSX- und die PNG*-Datei konfiguriert haben (*Abbildung 4*).

Abbildung\_4.png

Abbildung 4: Anzeige der Tilemap im Spiel.

Kollisionsdefinition in der TSX-Datei, Bewegung eines Actors in der Tilemap

Sie möchten ein etwas umfangreiches Beispiel ausprobieren und studieren? Dann sehen Sie sich den Quellcode unter [7] an. Hier wird eine Tilemap definiert, die auch Tiles enthält, welche durch die Spielfigur nicht betreten werden können. Die Definition erfolgt in diesem Beispiel in der TMX-Datei. Für die Spielfigur (Klasse *Player*) wird dann definiert, dass diese auf Kollisionen achten soll. Das sieht dann auszugsweise wie folgt aus:

export class Player extends ex.Actor {

constructor(pos: ex.Vector) {

super({

pos,

width: 16,

height: 16,

collisionType: ex.CollisionType.Active

})

}

…

}

Konkret bedeutet es, dass wir einen Kollisionstyp definieren (*collisionType: ex.CollisionType.Active*). Dieser Kollisionstyp besagt: Akteure mit der Einstellung *CollisionType.Active* lösen Kollisionsereignisse aus und nehmen Teil in Kollisionen mit anderen Akteuren, welche die Einstellung *CollisionType.Active* oder *CollisionType.Fixed* aufweisen. Probieren Sie es aus, d.h. starten Sie das Beispiel (*npm install*, gefolgt von *npm start*). Die Spielfigur kann im Spielfeld (*Tilemap*) frei bewegt werden, bis außer an definierten Hindernissen (*Abbildung 5*).

Abbildung\_5.png

Abbildung 5: Spielfigur in Tilemap mit Hindernissen.

Dass die Spielfigur bewegt werden kann ist übrigen auch in der Datei *player.ts* definiert. Experimentieren Sie! Entfernen Sie die Eigenschaft *collisionType: ex.CollisionType.Active* in der Klasse *Player*. Jetzt beachtet die Spielfigur keine Hindernisse mehr, d.h. Sie durchschreitet alle *Tiles* auf der *Tilemap*.

Physik

In Excalibur.js bezieht sich „Physics" auf das physikalische System der Game-Engine, das für die Simulation von Bewegung, Kollisionen und Kräften in einer Spielwelt verantwortlich ist. Excalibur bietet ein einfaches Physiksystem, das es Entwickler:innen ermöglicht, grundlegende physikalische Effekte wie Schwerkraft, Reibung und Elastizität zu simulieren (*Tabelle 1*).

|  |  |
| --- | --- |
| Element | Beschreibung |
| *Physics* | Das Physiksystem bietet zwei Hauptsimulationen zur Auswahl. Die *Arcade-Physik* eignet sich für grundlegende Kollisionserkennung bei nicht-rotierenden rechteckigen Bereichen. Anwendungsbeispiele sind kachelbasierte Spiele oder Top-Down-Spiele. Die *Arcade-Physik* ist standardmäßig aktiviert. Einschränkungen: Die Arcade-Physik unterstützt keine Reibung (*body.friction*). Die zweite Variante ist die *Realistische Physik*. Diese Simulation ist für Spiele geeignet, die realistische Kollisionen und Bewegungen erfordern. Die Realistische Physik ist nicht standardmäßig aktiviert und muss explizit im Engine-Konstruktor gesetzt werden:  const game = new Engine({  physics: {  solver: SolverStrategy.Realistic  },  // weitere Konfigurationen  });  Durch die Auswahl der passenden Physik-Simulation können Entwickler das Verhalten ihrer Spielobjekte entsprechend den Anforderungen ihres Spiels steuern. |
| *Bodies (Körper):* | Das sind zentrale Komponenten des Physiksystems, die für die Handhabung von Kollisionen und physikalischen Reaktionen verantwortlich sind. Sie repräsentieren die physikalischen Eigenschaften eines Objekts und bestimmen, wie es auf Kräfte und Kollisionen reagiert. Hauptmerkmale von Bodies sind *Masse (Mass)*, bestimmt, wie stark ein Objekt auf Kräfte reagiert); *Trägheit (Inertia)*, beeinflusst die Rotationsdynamik des Objekts; *Elastizität (Bounciness)*, legt fest, wie stark ein Objekt nach einer Kollision abprallt und *Reibung (Friction)* bestimmt den Widerstand, den ein Objekt bei Bewegung über eine Oberfläche erfährt. |
| *Kollisionsereignisse (Collision Events)* | Das sind zentrale Mechanismen, die es ermöglichen, auf Kollisionen zwischen Objekten in der Spielwelt zu reagieren. Diese Ereignisse werden ausgelöst, wenn zwei physische Körper interagieren, und bieten Hooks, um spezifische Logik während verschiedener Phasen einer Kollision zu implementieren.   * *collisionstart:* Dieses Ereignis wird ausgelöst, wenn zwei physische Körper erstmals kollidieren. Es wird nur einmal ausgelöst, solange die beiden Körper in Kontakt bleiben, beispielsweise wenn ein Spieler auf einer Plattform landet oder ein Projektil ein Ziel trifft.   actor.on('collisionstart', (event) => {  // Logik bei Kollisionsbeginn  });   * *collisionend*: Dieses Ereignis tritt ein, wenn zwei zuvor kollidierende Körper den Kontakt zueinander verlieren, beispielsweise um feststellen, wann ein Objekt aus einem bestimmten Bereich herausbewegt wird.   actor.on('collisionend', (event) => {  // Logik bei Kollisionsende  });   * *precollision*: Dieses Ereignis wird vor der eigentlichen Kollisionsauflösung ausgelöst. Hier kann man die Kollision überprüfen und ggf. anpassen, bevor die Spiel-Engine die Bewegung und die Reaktionen der kollidierenden Objekte endgültig berechnet. Dieses Ereignis wird in jedem Frame ausgelöst, in dem zwei Objekte kollidieren. Es findet vor der Berechnung der Kollision statt, was die Möglichkeit gibt, die Kollision zu modifizieren oder spezielle Aktionen durchzuführen, bevor die Physik-Engine die Reaktion bestimmt. Das precollision-Event kann genutzt werden, um die Kollision so zu beeinflussen, dass das Verhalten angepasst wird. Beispielsweise könnte man in einem Breakout-Spiel die Richtung eines Balls je nach Treffpunkt auf einem Schläger dynamisch anpassen. * *postcollision*: Dieses Event wird nach der Kollisionsauflösung ausgelöst, wenn die Engine die physikalischen Reaktionen, beispielsweise das Abprallen oder die Verlangsamung, abgeschlossen hat. Es ist ideal, um auf das finale Ergebnis der Kollision zu reagieren, wie das Auslösen von visuellen Effekten, das Aktualisieren des Status oder das Starten von Sounds. In einem Kampfspiel könnte das *postcollision*-Event verwendet werden, um eine Animation abzuspielen, wenn zwei Charaktere kollidieren, oder um Schadensberechnungen vorzunehmen, nachdem die Bewegungen durch die Kollision vollständig berechnet wurden. |
| *Collision Types* | Collision Types (Kollisionstypen) bestimmen, wie Objekte in der Spielwelt auf Kollisionen reagieren. Sie legen fest, ob und wie ein Objekt mit anderen kollidiert und welche Auswirkungen diese Kollisionen haben. Die vier Hauptkollisionstypen sind:   * *PreventCollision*: Objekte mit diesem Kollisionstyp nehmen nicht an Kollisionen teil und lösen keine Kollisionsereignisse aus. Anwendungsbeispiel: Dekorative Elemente oder Hintergrundobjekte, die keine physikalische Interaktion benötigen. * *Passive*: Diese Objekte lösen Kollisionsereignisse aus, werden jedoch weder von anderen Objekten beeinflusst noch beeinflussen sie andere. Anwendungsbeispiel: Trigger-Zonen oder Sensoren, die auf Kollisionen reagieren sollen, ohne physikalische Reaktionen zu verursachen. * *Active*: Aktive Objekte nehmen vollständig an Kollisionen teil. Sie lösen Kollisionsereignisse aus und werden sowohl von anderen Objekten beeinflusst als auch umgekehrt. Anwendungsbeispiel: Spielercharaktere oder bewegliche Hindernisse, die mit der Umgebung interagieren. * Fixed: Feste Objekte lösen Kollisionsereignisse aus und beeinflussen andere Objekte, werden jedoch selbst nicht bewegt oder beeinflusst. Anwendungsbeispiel: Statische Hindernisse wie Wände oder Plattformen, die unbeweglich bleiben sollen. |
| *Fixed Update* | Fixed Update ist eine Methode, die es ermöglicht, die Aktualisierungsrate der Spielphysik unabhängig von der Bildwiederholrate des Spiels festzulegen. Dies ist besonders nützlich, um eine stabile und konsistente Physik-Simulation zu gewährleisten, insbesondere in Spielen, die präzise Bewegungen und Kollisionen erfordern. Eine feste Aktualisierungsrate wird über die Eigenschaft *fixedUpdateFps* im Engine-Konstruktor definiert:  const game = new ex.Engine({  fixedUpdateFps: 30,  // weitere Konfigurationen  });  **Interpolation von Grafiken:**  Wenn die feste Aktualisierungsrate (*fixedUpdateFps*) niedriger ist als die tatsächliche Bildwiederholrate des Displays, interpoliert Excalibur automatisch die Grafiken, um ein gleichmäßiges und flüssiges visuelles Erlebnis zu bieten.  **Auswirkungen auf die Spielaktualisierung:**   * *Display-FPS > Fixed-Update-FPS:* In diesem Fall wird die *update*-Methode möglicherweise in einigen Frames übersprungen, um die festgelegte Aktualisierungsrate einzuhalten. Beispielsweise, wenn das Display mit 60 FPS läuft und die feste Aktualisierungsrate auf 30 FPS gesetzt ist, wird die *update*-Methode nur in jedem zweiten Frame ausgeführt. * *Fixed-Update-FPS > als Display-FPS*: Hier kann es vorkommen, dass die *update*-Methode innerhalb eines Frames mehrfach ausgeführt wird, um die festgelegte Aktualisierungsrate zu erreichen. Beispielsweise, wenn das Display mit 30 FPS läuft und die feste Aktualisierungsrate auf 60 FPS gesetzt ist, wird die *update*-Methode in jedem Frame zweimal ausgeführt.   **Leistungsüberlegungen:**  Eine niedrigere feste Aktualisierungsrate kann die Leistung verbessern, jedoch auf Kosten der Simulationsqualität. Umgekehrt kann eine höhere Aktualisierungsrate eine präzisere Physik-Simulation bieten, jedoch mit erhöhtem Leistungsaufwand verbunden sein. |
| *Colliders* | Das sind Abstraktionen geometrischer Formen, die zur Kollisionserkennung und -behandlung verwendet werden. Sie implementieren das Collider-Interface und bieten Funktionen wie die Erkennung von Überschneidungen mit anderen Kollidierern, Strahlabfragen (*Raycasts*) und Punktprüfungen:   * *Kreis-Kollidierer (CircleCollider):* Definiert eine kreisförmige Geometrie. * *Polygon-Kollidierer (PolygonCollider):* Ermöglicht die Definition beliebiger konvexer Polygone. * *Box-Kollidierer:* Eine spezielle Form des PolygonCollider, die rechteckige Formen repräsentiert.   Beispiel:  const actor = new ex.Actor({   pos: ex.vec(100, 100),   radius: 10,   collisionType: ex.CollisionType.Active   }); |
| *Collision Groups* | *Collision Groups (Kollisionsgruppen)* ermöglichen eine präzise Steuerung darüber, welche Objekte in der Spielwelt miteinander kollidieren können. Sie bieten eine granulare Ebene der Kollisionsfilterung, die über die standardmäßigen Kollisionstypen hinausgeht. Dieses Konzept ist auch als Kollisionsschichten oder Kollisionsfilterung bekannt. Hauptmerkmale von Kollisionsgruppen sind:   * *Kategorisierung von Objekten:* Jede Kollisionsgruppe repräsentiert eine spezifische Kategorie von Objekten, wie beispielsweise Spieler, Gegner oder Hindernisse. Durch die Zuweisung von Objekten zu bestimmten Gruppen kann definiert werden, welche Gruppen miteinander kollidieren sollen und welche nicht. * *Bitmasken zur Kollisionssteuerung:* Kollisionsgruppen verwenden Bitmasken, um festzulegen, mit welchen anderen Gruppen sie kollidieren können. Jede Gruppe erhält eine eindeutige 32-Bit-Maske, wobei jede Bitposition eine spezifische Gruppe repräsentiert. Durch das Setzen oder Löschen bestimmter Bits in der Maske wird definiert, ob eine Kollision zwischen zwei Gruppen möglich ist.   Beispiel: Angenommen, wir haben ein Spiel mit folgenden Gruppen:   * Spieler (*playerGroup*) * Gegner (*enemyGroup*) * Boden (*floorGroup*)   Spieler sollen nicht miteinander kollidieren, aber mit Gegnern und dem Boden. Gegner sollen nicht miteinander kollidieren, aber mit Spielern und dem Boden. Die Definition für die Spieler kann wie folgt aussehen:  const playerGroup = CollisionGroupManager.create('player')  playerGroup.collidesWith(enemyGroup, floorGroup);  const player = new Actor({  collisionGroup: playerGroup,  …  }); |

Tabelle 1: Merkmale der Physics-Engine von excalibur.js [5].

Interaktionen

In *Excalibur.js* stehen Entwicklern verschiedene Eingabemethoden zur Verfügung, um Interaktionen im Spiel zu ermöglichen. Diese umfassen Tastatur-, Maus-, Touch- und Gamepad-Eingaben [6]:

* *Tastatureingaben*: Über *engine.input.keyboard* kann man den Zustand von Tasten abfragen, um festzustellen, ob eine Taste gedrückt, gehalten oder losgelassen wurde. Häufig verwendete Tasten sind im *Keys-Enum* definiert, es können jedoch auch beliebige Zeichen-Codes verwendet werden.
* *Maus- und Touch-Eingaben*: Maus- und Touch-Ereignisse werden über das *engine.input.pointers*-API verwaltet, das dem W3C Pointer Events-Standard folgt. Dieses API normalisiert Maus- und Touch-Ereignisse zu einem einheitlichen *PointerEvent*, wodurch die Implementierung von Eingaben für verschiedene Geräte vereinfacht wird.
* *Gamepad- und Controller-Eingaben*: Excalibur.js nutzt die HTML5 Gamepad API, um Unterstützung für Gamepads und Controller bereitzustellen. Über *engine.input.gamepads* kann man auf angeschlossene Gamepads zugreifen und deren Eingaben verarbeiten. Dies ermöglicht die Implementierung von Controller-Unterstützung für Spiele.

Durch die Kombination dieser Eingabeoptionen kann man vielfältige Interaktionen in das Browser-Game implementieren.

Fazit und Ausblick

Wir haben in diesem Teil der Artikelserie weitere wichtige Basisfeatures der Spiele Engine Excalibur.js angesehen. Dabei sind wir im Bereich der Tilmaps technisch in Form von Quellcodebeispielen sehr weit eingetaucht. Ein möglicher Weg zu einem ersten Spielesetting wären beispielsweise die folgenden Schritte: (*1) Definition einer Tilemap, (2) Definition eines Actors* (Spielfigur) und *(3) Umsetzung einer minimalen Spielidee (Bewegung, Aktionen)*. Dabei haben wir auch gesehen, dass Excalibur.js viele Features aus dem Bereich der Physik bietet, welche in den meisten Spielen benötigt werden, beispielsweise Methoden zur Kollisionserkennung oder um Masse oder Reibung zu simulieren. Im kommenden Teil der Serie, wollen wir die „Dinge“ zusammenbringen und den Pfad zu einem ersten Spiel beschreiben. Dabei werden wir auch weitere interessante Features von Excalibur.js in den Blick nehmen.

*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://www.mapeditor.org/>

[3] <https://www.aseprite.org/docs/tilemap>

[4] <https://doc.mapeditor.org/en/stable/reference/tmx-map-format/>

[5] <https://excaliburjs.com/docs/category/physics>

[6] <https://excaliburjs.com/docs/category/input>

[7] <https://github.com/excaliburjs/sample-tiled-parcel>

Listing 1: Beispiel für eine TMX (Tilemap)-Datei:

<?xml version="1.0" encoding="UTF-8"?>

<map version="1.10" tiledversion="1.11.0" orientation="orthogonal" renderorder="right-down" width="30" height="20" tilewidth="32" tileheight="32" infinite="0" nextlayerid="3" nextobjectid="1">

<tileset firstgid="1" source="elements.tsx"/>

<layer id="1" name="Kachelebene 1" width="30" height="20">

<data encoding="csv">

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,40,0,0,0,0,0,0,0,0,0,0,0,0,0,

</data>

</layer>

</map>./.

./.

Listing 2: laden einer Tilemap:

import { Loader } from "excalibur";

import { TiledResource } from '@excaliburjs/plugin-tiled';

// Import paths to work with Parcel

import tilesetPath from '../res/elements.png';

import tmxPath from '../res/elements.tmx';

import tsxPath from '../res/elements.tsx';

export const Resources = {

    TiledMap: new TiledResource(tmxPath,{

      // Path map intercepts and redirects to work around parcel's static bundling

      pathMap: [

        { path: 'elements.tmx', output: tmxPath },

        { path: 'elements.png', output: tilesetPath },

        { path: 'elements.tsx', output: tsxPath }

      ]

    })

}

export const loader = new Loader();

for (let resource of Object.values(Resources)) {

    loader.addResource(resource);

}

./.