Dokumentation CrazyMaze

# Inhalt

[1. Inhalt 1](#_Toc205495817)

[2. Im Vorhinein 2](#_Toc205495818)

[3. Konzept – Kurzfassung 2](#_Toc205495819)

[4. Technologie Stack 3](#_Toc205495820)

[4.1 Programmiersprachen 3](#_Toc205495821)

[4.2 Mobile-Framework 3](#_Toc205495822)

[4.3 JavaScript-Bibliotheken 3](#_Toc205495823)

[4.4 Backend 3](#_Toc205495824)

[4.5 Besondere Web-APIs 3](#_Toc205495825)

[4.6 Künstliche Intelligenz 3](#_Toc205495826)

[4.7 Sonstiges 4](#_Toc205495827)

[5. Datenbankschema 4](#_Toc205495828)

[6. UI/UX 5](#_Toc205495829)

[6.1 User Flow 5](#_Toc205495830)

[6.2 UI-Design 6](#_Toc205495831)

[7. Code 6](#_Toc205495832)

[7.1 Navigation 6](#_Toc205495833)

[7.2 Game Loop 6](#_Toc205495834)

[7.2.1 Kollisionschecks 7](#_Toc205495835)

[7.3 Sounds 7](#_Toc205495836)

[7.4 Backend-Verbindung 8](#_Toc205495837)

[7.5 Laden der Levels 8](#_Toc205495838)

# Im Vorhinein

Warum CrazyMaze und nicht ömi?

Mein ursprüngliches Projektkonzept beschreibt die App „ömi“, die für Migrant\*innen in Österreich Informationen bereitstellen und sie mit Expert\*innen verbinden sollte. An dieser App habe ich bis letzte Woche gearbeitet. Die Entscheidung des JavaScript-Frameworks war auf [Next.js](https://nextjs.org/) gefallen. Nach einer kurzen (wie sich herausstellt nicht ausreichend langen) Recherche war ich überzeugt gewesen, dass sich Next.js ohne Probleme in eine native App umwandeln lassen wird. Wie sich später herausstellt ist das nahezu unmöglich, da der clientseitige Code eng mit den Server-Components verbunden ist.

Statischer Export ist für die App nicht möglich, da sie Server-Funktionen nutzt. Capacitor wäre nur ein Wrapper und würde die Anwendung nicht nativ machen.

<https://www.reddit.com/r/nextjs/comments/1fh8vw4/turning_nextjs_project_into_native_app_capacitor/>

<https://nextjs.org/docs/app/getting-started/deploying#static-export>

CrazyMaze ist jetzt die Last-Minute-Lösung, denn die Anforderung „Endprodukt Native App“ ist unumgänglich. Das Projektkonzept ömi wurde der Abgabe beigefügt.

# Konzept – Kurzfassung

Im Spiel wird eine Murmel durch ein Labyrinth ans Ziel gebracht, während Löchern im Boden ausgewichen werden muss. Das Steuern der Murmel erfolgt durch die Device Orientation API oder die Pfeiltasten auf Desktop-Geräten.

Schriftart: [Barriecito](https://fonts.google.com/specimen/Barriecito)

Hintergrundfarbe: #000000

Akzentfarben: #ff7e5f, #feb47b

Allgemein: Abgerundete Ecken, leicht durchsichtiger Overlay-Hintergrund

# Technologie Stack

## Programmiersprachen

Die App wird mit purem [VanillaJS](http://vanilla-js.com/), HTML und CSS entwickelt. Es wird bewusst auf ein JavaScript-Framework verzichtet, um überflüssige Komplexität zu vermeiden und da das Spiel kein kompliziertes State-Management erfordert.

## Mobile-Framework

Damit die Webanwendung als native mobile App läuft, wird das Framework Cordova eingesetzt.

Cordova-Plugins:

* *Screen Orientation*, damit die App im „Portrait-Mode“ fixiert wird.
* *Device*, um Metainformationen über das aktuelle Gerät zu laden.
* *Network Information*, damit erkannt wird, wann die App Daten vom Server laden kann, und wann auf lokale Ersatzdaten zugegriffen werden soll.

## JavaScript-Bibliotheken

Das Programm nutzt die Bibliotheken [JSConfetti](https://github.com/loonywizard/js-confetti) für Konfetti-Effekte und [JSZip](https://stuk.github.io/jszip/). Letztere wird zum Entpacken der Leveldaten aus GeoGebra-Dateien verwendet, mehr dazu unter [Laden der Levels](#_Laden_der_Levels).

## Backend

Als Backend dient [Supabase](https://supabase.com/). Supabase bietet „Database as a Service“ mit großzügigen Gratisoptionen an (2 aktive Datenbanken pro Account) und lässt sich schnell einrichten. Das Frontend kommuniziert mit [PostgREST](https://docs.postgrest.org/) über XMLHttpRequest mit der PostgreSQL Datenbank.

## Besondere Web-APIs

* *XMLHttpRequest*: Backend-Kommunikation und Laden von Leveldateien
* *Web Audio API*: Generierung von Geräuschen für Rollen, Fallen und Kollisionen der Murmel
* *Device Orientation Events*: Steuerung der Murmel

## Künstliche Intelligenz

Es wird mit dem Copilot-Code-Vervollständigungs-Feature entwickelt. Zusätzlich wurde der Code zur Kollisionsdetektion von KI verbessert (ursprünglich instabil bei Kollision mit Ecken) und die Erzeugung der Murmelgeräusche „Rollen“, „Fallen“ und „Kollision“ sind vollständig KI-generiert, da mir in diesem Bereich (Audiofilter, Pink Noise, Oszillatoren, …) Wissen fehlt, welches sich nicht so schnell hätte aneignen lassen.

## Sonstiges

* Editor: Visual Studio Code
* Versionsmanagement: Git
* Automatische Code-Formatierung: Prettier

Zum Testen wird die Webanwendung mit dem VSCode-Plugin [Five Server](https://marketplace.visualstudio.com/items?itemName=yandeu.five-server) auf localhost gestartet, [ngrok](https://ngrok.com/) leitet eine Domain auf localhost weiter und diese wird dann vom Handy aus aufgerufen. Hierfür müssen die beiden Geräte praktischerweise nicht im selben Netzwerk sein.

npx ngrok http --url=proven-ghost-social.ngrok-free.app 8085

Jedes Mal Cordova builden wäre zu zeitaufwändig und die genannte Methode unterstützt Live-Updates, wenn der Code bearbeitet wird.

# Datenbankschema

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Abbildung 1 – Einzige Tabelle der Datenbank, Screenshot aus Supabase

# UI/UX

## User Flow

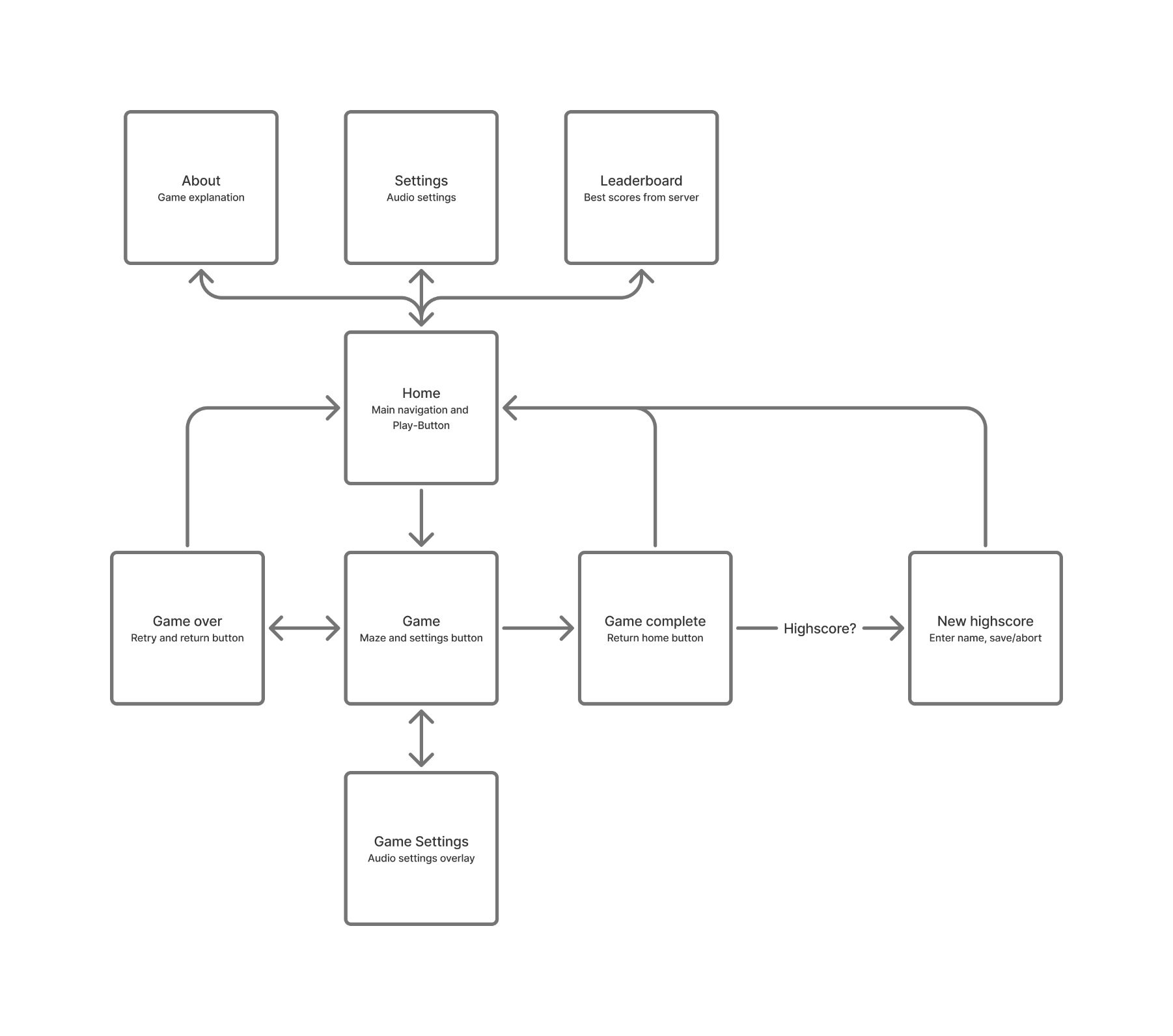


Abbildung 2 – User-Flow-Diagramm, mit Figma erstellt

## UI-Design

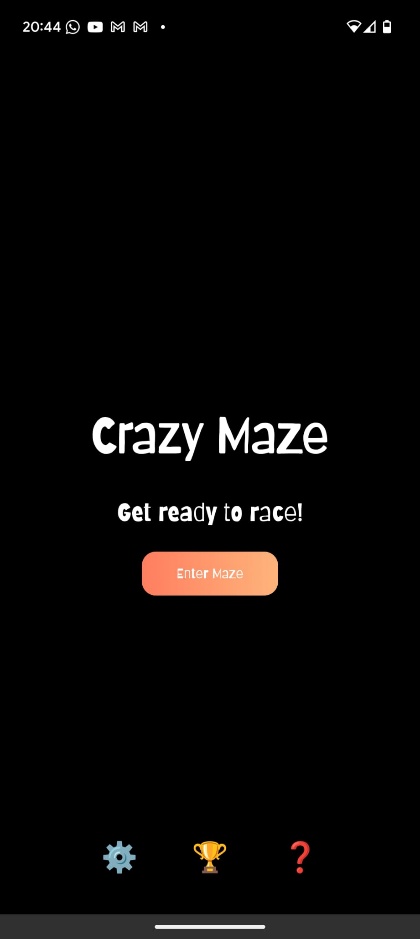
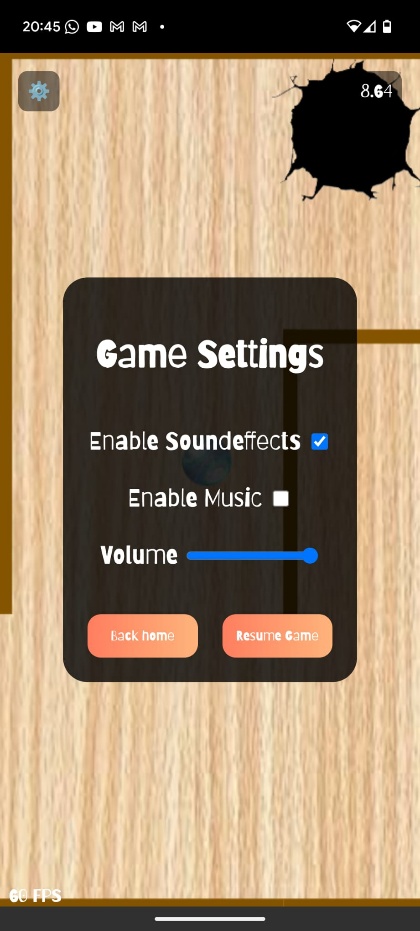


Abbildung 3 - Screenshots aus dem Spiel

# Code

## Navigation

Kern der Navigation ist die Funktion *navigateToPage* in index.js. Alle möglichen Seiten der App sind in der Klasse Pages angegeben und in die *navigateToPage*-Funktion wird eine dieser Pages weitergegeben. Anhand des Seitennamens wird das HTML aus dem */pages* Ordner geladen und über innerHTML in das Root-Element eingefügt. Danach wird der seitenspezifische Code ausgeführt, der am Anfang der Datei index.js für alle Seiten importiert wird. Jedes dieser „Page-Module“ hat eine Funktion *start*, in der die Seite initialisiert wird und eine Funktion *stop*, in der bei Bedarf EventListener entfernt oder laufende Audios gestoppt werden.

## Game Loop

Die Spiel-Schleife ist eine while-Schleife, die ausgeführt wird, solange *gameRunning* wahr ist. Nach jeder Iteration wird ein Frame-Zähler erhöht und es wird gemessen, wie lange das Frame gedauert hat. Liegt diese Dauer über der Ziel-Framerate (60 FPS), wird vor der nächsten Iteration für die verbleibende Zeit gewartet – im seltenen anderen Fall werden am Bildschirm zur Warnung die effektiven FPS in orange angezeigt.

In der Spiel-Schleife passiert in chronologischer Reihenfolge:

* Berechnung der Lautstärke des Kugel-Roll-Geräuschs
* Anpassung der Spielerposition und -geschwindigkeit
* Kollisionschecks
  + Kollision mit Wänden (in mehreren dynamisch berechneten Sub-Schritten, um Tunneln durch Wände zu vermeiden)
  + Kollision mit dem Ziel
  + Kollision mit einem der Löcher. Bei Kollision wird die Spiel-Schleife noch für einige Iterationen wiederholt (Death-Animation) und bricht dann ab.
* Anpassung der Kameraposition (folgt dem Spieler)
* Rendering
  + Das Spiel arbeitet in einem eigenen, unabhängigen Koordinatenbereich. Durch Konstanten wird angegeben, wieviel Prozent der Map-Größe der Viewport ausmachen soll. Anhand dieses Skalierungs-Faktors und der Kameraposition werden die Objekte auf den Canvas projiziert und mit *drawImage* oder *fillRect* gezeichnet. Mögliche zukünftige Verbesserung: Nur zeichnen, wenn wirklich innerhalb des Viewports (Frame-Dauer momentan jedoch sowieso bei nur 1-2ms / Frame)
* Berechnung und Stabilisierung der Iterationsdauer

### Kollisionschecks

Klasse „Rect“ in utils.js. Zuerst wird der Punkt mit geringster Entfernung zum Spieler am Rechteck ausgerechnet. Danach wird die Länge des Distanz-Vektors mit dem Radius des Spielers verglichen. Falls der Spieler im Rechteck ist, wird er um die Penetrationstiefe in die entgegengesetzte Richtung hinausbewegt. Daraufhin wird die Geschwindigkeit reflektiert, abhängig vom Skalarprodukt der Geschwindigkeit und des Distanzvektors.

## Sounds

Einige Sounds werden mit der Web Audio API selbst generiert. Dadurch klingt das Rollen der Kugel echter und kann an die Spielergeschwindigkeit angepasst werden. Die App-Lautstärke wird mit einer globalen GainNode kontrolliert. Mögliche zukünftige Verbesserung: Kollisionsgeräusche auch an die Geschwindigkeit anpassen.

## Backend-Verbindung

Wie erwähnt erfolgt die Kommunikation mit der PostgreSQL Datenbank mit XMLHttpRequest und der PostgREST API. Der Code dafür ist in server.js. Highscores werden mit einem INSERT und ON CONFLICT UPDATE eingefügt, die Tabelle hat einen kombinierten Unique-Index über die Spalten „player\_id“ und „level“, da immer nur der beste Score pro Spieler pro Level gespeichert wird. Falls die Anfrage fehlschlägt oder das Gerät offline ist wird localStorage als Fallback verwendet.

## Laden der Levels

Das Leveldesign erfolgt mit GeoGebra.

Ein Bild, das Text, Diagramm, Screenshot, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Abbildung 4 - Level 1 in GeoGebra

Die entstehenden ggb-Dateien werden über XMLHttpRequest geladen, mit JSZip entpackt und als XML-Dokument geladen. In diesem Dokument werden mit querySelector die Wände, Map, Startposition, Löcher und Zielposition ausgelesen. Die ausgelesenen Daten werden als Instanz der Klasse „Level“ im levelsCache gespeichert.