

Fakultät 4 – Elektrotechnik und Informatik

**Medieninformatik 2**

**Projekt: Webgame mit Phaser.js**

Jump-Adventure: „Possible Game“

**Verfasser:**  Philipp Moritzer, 5034255

Hannes Lesemann, 5017055

Pascal Seegers, 5051429

**Modul:**  Medieninformatik 1

**Dozent:** Prof. Dr. Volker Paelke

**Labor:** Dipl.-Inf. Andreas Lochwitz

**Abgabedatum:** 28.06.2019

Inhalt

[1 Spielidee 1](#_Toc9613644)

[2 Handlung 1](#_Toc9613645)

[2.1 Story 1](#_Toc9613646)

[2.2 Interaktion 1](#_Toc9613647)

[3 Grafische Gestaltung 1](#_Toc9613648)

[3.1 Layout und Gestaltungsmerkmale 1](#_Toc9613649)

[3.1.1 Hauptmenü 1](#_Toc9613650)

[3.1.2 Level-Selektion 1](#_Toc9613651)

[3.1.3 Spiel 2](#_Toc9613652)

[3.2 Parallax 2](#_Toc9613653)

[3.3 Mobile Ansichten 2](#_Toc9613654)

[3.4 Schriften 2](#_Toc9613655)

[3.5 Tiles / Sprites 2](#_Toc9613656)

[3.6 Musik 2](#_Toc9613657)

[4 Verwendete Technologien 2](#_Toc9613658)

[4.1 Verwendete Sprachen 2](#_Toc9613659)

[4.2 Bibliotheken 3](#_Toc9613660)

[4.3 Editor und Tools 3](#_Toc9613661)

[4.4 Versionsverwaltung 3](#_Toc9613662)

[4.5 Konventionen 3](#_Toc9613663)

[5 Architektur / Implementierung 3](#_Toc9613664)

[5.1 Aufbau des Projektes 3](#_Toc9613665)

[5.2 Einstiegspunkt 4](#_Toc9613666)

[5.3 Scenes 4](#_Toc9613667)

[5.3.1 Preload 4](#_Toc9613668)

[5.3.2 Menu 5](#_Toc9613669)

[5.3.3 Stages 6](#_Toc9613670)

[5.4 JavaScript 6](#_Toc9613671)

[5.4.1 game-state.js 7](#_Toc9613672)

[5.4.2 state-managing.js 7](#_Toc9613673)

[5.4.3 music-config.js, constants.js, util.js 7](#_Toc9613674)

[5.4 Skalierung 7](#_Toc9613675)

[5.5 JSON-Tilemaps 7](#_Toc9613676)

[5.6 Collision-Debugging 8](#_Toc9613677)

[6 Projektplanung und Arbeitsteilung 9](#_Toc9613678)

[6.1 Zeitlicher Ablauf und Arbeitsteilung 9](#_Toc9613679)

[7 Post Mortem 9](#_Toc9613680)

[7.1 Was hast gut funktioniert 9](#_Toc9613681)

[7.2 Was hat nicht funktioniert 9](#_Toc9613682)

[7.3 Rückblick 9](#_Toc9613683)

[7.4 Ausblick 10](#_Toc9613684)

[Anhang I](#_Toc9613685)

[i. Projektplanung I](#_Toc9613686)

[ii. Projektstruktur II](#_Toc9613687)

[iii. Config-Objekt III](#_Toc9613688)

[iv. LoadingBar()-Methode III](#_Toc9613689)

[iii. Tiled Map Building IV](#_Toc9613690)

[iv. Tileset V](#_Toc9613691)

[vi. Auszug Tilemap als .json V](#_Toc9613692)

[iv. Collision debugging VI](#_Toc9613693)

[iii. Desktop Menu Screenshots VII](#_Toc9613694)

[iv. Desktop Game Screenshots VIII](#_Toc9613695)

[v. Mobile Ansichten X](#_Toc9613696)

**Abbildungsverzeichnis**

[Abbildung 1: index.html Auszug 3](#_Toc9605172)

[Abbildung 2: Scene-Klasse am Beispiel Stage1 4](#_Toc9605173)

[Abbildung 3: Scene-Konstruktor am Beispiel Stage1 4](#_Toc9605174)

[Abbildung 4: GamePreload preload()-Methode 4](#_Toc9605175)

[Abbildung 5: Start Menu-Scene 4](#_Toc9605176)

[Abbildung 6: Menüpunkt laden am Beispiel vom "Play"-Button 5](#_Toc9605177)

[Abbildung 7: Interaktion von Menüpunkten am Beispiel vom "Play"-Button 5](#_Toc9605178)

[Abbildung 8: update()-Method Auszug 6](#_Toc9605179)

[Abbildung 9: GANTT-Diagramm Planung I](#_Toc9605180)

[Abbildung 10: Projektstruktur II](#_Toc9605181)

[Abbildung 11: config-Objekt III](#_Toc9605182)

[Abbildung 12: LoadingBar()-Methode III](#_Toc9605183)

[Abbildung 13: Erstellen von Tiledmaps IV](#_Toc9605184)

[Abbildung 14: Tiled layers IV](#_Toc9605185)

[Abbildung 15: Tiled Attribute V](#_Toc9605186)

[Abbildung 16: Tileset V](#_Toc9605187)

[Abbildung 17: Auszug Tilemap .json-Export VI](#_Toc9605188)

[Abbildung 18: Collision Debugging VI](#_Toc9605189)

[Abbildung 19: Desktop-Hauptmenü VII](#_Toc9605190)

[Abbildung 20: Dektop-Level-Select VII](#_Toc9605191)

[Abbildung 21: Stage 1 – Screenshot VIII](#_Toc9605192)

[Abbildung 22: Stage 2 – Screenshot VIII](#_Toc9605193)

[Abbildung 23: Stage 3 – Screenshot IX](#_Toc9605194)

[Abbildung 24: Stage 4 – Screenshot IX](#_Toc9605195)

[Abbildung 25: Menu-Screenshot Phone X](#_Toc9605196)

[Abbildung 26: Select-Level Screenshot Phone X](#_Toc9605197)

[Abbildung 27: Game Screenshot Phone XI](#_Toc9605198)

[Abbildung 28: Game Screenshot Tablet XI](#_Toc9605199)

# 1 Spielidee

Die Idee des Spiels ist es eine Art „Platformer“ mit nur einer Eingabemöglichkeit zu erstellen. Die Herausforderung hierbei kommt über das perfekte Timing dieser Eingabe und mit dieser Möglichkeit kann man dieses Spiel interaktiv / schwierig machen. Die Eingaben sollen hierbei bewusst gegen den üblichen Rhythmus des Menschen verstoßen. Hier muss die Balance zwischen Frustration und Spielspaß gefunden werden. Die Level sollen fiese Obstacles und Twists enthalten, die das Spielgeschehen interessanter und interaktiv machen. Das Ziel des Spiels soll es sein, einen „Parkour“ bzw. mehrere Level innerhalb von möglichst wenigen Versuchen zu durchdringen. Dabei sollen die Maps / Level jeweils einem bestimmten Theme nachgehen. Gegebenenfalls entsteht durch das Theming weiterhin eine kleinere Story. Das Spiel so primär als Lückenfüller für Desktop / als Spiel nebenbei fürs Handy funktionieren.

# 2 Handlung

## 2.1 Story

- Astronaut

- Zürck zum Mond

- mehr noch unsicher

## 2.2 Interaktion

Das Spielen des Hauptspiels definiert sich durch drücken einer Taste oder durch einen einfachen Touch-Input. Der Charakter durchläuft die level automatisch, die Aufgabe des Spielers ist es den Abgründen und den Stacheln auszuweichen und so ins Ziel zu kommen. Im Menü kann man sich durch klick entscheiden ein neues Spiel anzufangen. Hier beginnt man bei Level 1 und arbeitet sich bis Level 4 durch. Ziel ist es in diesem Modus so wenig wie möglich zu sterben. Der Highscore (wenigste Versuche) wird gespeichert. „Stirbt“ man in einem Level, so wird man an den Anfang des Levels zurückgesetzt. Mit der „Space“-Taste kann man das ganze geschehen pausieren. Dort gibt es dann auch eine Möglichkeit zum Menü zurückzukehren. Drückt man erneut „space“, wird das Spiel fortgesetzt.

Neben einfachen Sprungaufgaben wird der Spieler in den Leveln mit umgekehrter Anziehungskraft, Rückwärtslaufen und wenig Anziehungskraft gefordert.

In dem Menü „Select Level“ kann man den Übungsmodus starten, in dem der Spieler die Möglichkeit hat, die Level einzeln zu durchlaufen und zu üben. Diese zählen nicht zu dem lokalen Highscore.

# 3 Grafische Gestaltung

## 3.1 Layout und Gestaltungsmerkmale

### 3.1.1 Hauptmenü

Für Screenshots der Intro-Seite siehe im Anhang ii. Screenshots - Intro.

### 3.1.2 Level-Selektion

Für Screenshots der Charakter-Selektions-Seite siehe im Anhang iii. Screenshots - Charakterauswahl.

### 3.1.3 Spiel

Für Screenshots der Game-Seite siehe im Anhang iv. Screenshots – Game.

## Parallax

## 3.3 Mobile Ansichten

Für Screenshots der mobilen Ansicht siehe xyz

Responsive

## 3.4 Schriften

Im Spiel werden vier unterschiedliche Schriftarten verwendet. Diese sind Arcade Classic, Zorque, Arial und Helvetica. Während Arcade Classic einen gewissen nostalgischen Effekt mit sich bringt

## Tiles / Sprites

Das benutzte Tileset (s. Abbildung 19: Tileset) ist ein Standard Platformer Tilesets mit 16x16 Tiles. Es bietet verschiedene Environments als Pixel-Art, wodurch es möglich ist

* „Pixel“-Art
* Theming durch verschiedene Enviromental-Tiles
* 16x16
* Astronaut

## Musik

Für die passende Untermalung des Menüs und der jeweiligen Stages wurde jeweils unterschiedliche Musik gewählt.

Für das Menü wurde eine relativ repetitive Musik gewählt, die das Menü ein wenig untermalt und lebendiger macht.

Für die 1. Stage, die in den Hügeln/Wald spielt ist ebenfalls ein recht simples Soundbild. Es hört sich sehr einheitlich. Die 1. Stage ist vom Design auch wenig reißerisch, weshalb diese Musik gewählt wurde.

Die 2. Stage ist mit einer relativ minimalistischen Musik untermalt, die sich orientalisch anhört und dementsprechend gut in das Ambiente einer Wüste einpasst.

Die Musik der 3. Stage erinnert durch den Sound an Schneefall und Weihnachten und mischt sich mit diesem Soundbild in das optische der Stage ein.

Für den Mond gibt es eine Musik, die eher ein Rauschen nachempfunden ist.

Die Sounds für das Springen des Charakters und für das Verlieren eines Spieles sind an klassische Arcade-Sounds angelegt, die meisten Menschen würden also an dem Sound bereits erkennen, was passiert.

# 4 Verwendete Technologien

## 4.1 Verwendete Sprachen

Für dieses Projekt werden die drei gängigen Sprachen in der Webentwicklung verwendet: HTML, JavaScript und CSS. Die Sprachen wurden in ihrer Grundform genutzt und es werden keine Abwandlungen, wie TypeScript oder SASS verwendet. Der Anteil von HTML und CSS ist sehr gering, da die Elemente über die Game-Engine Phaser geregelt werden, welche es erlaubt die Implementierung fast ausschließlich in JavaScript vorzunehmen. Es muss lediglich ein Container und die Rahmenbedingungen der Spielszene in HTML und CSS umgesetzt werden. Es werden Teile der ES6-Syntax verwendet, beispielsweise Arrow-Funktionen.

## 4.2 Bibliotheken

Als einzige Bibliothek wurde Phaser verwendet. Phaser ist ein Framework, welches zur Entwicklung von Spielen in HTML5 dient. Als Renderer wird WebGL verwendet.

## 4.3 Editor und Tools

Als Editor wurde Visual Studio Code mit folgenden Plugins verwendet:

* Live Server
* Prettier
* Debugger for Chrome

Hauptsächlich wurde die Applikation auf Google Chrome getestet, von Zeit zu Zeit auf Mozilla Firefox Safari (iOS) oder dem Standard Android Browser. Es ist kein Support für Internet Explorer oder Microsoft Edge vorgesehen, dementsprechend wurden diese Browser auch nicht zum Testing verwendet.

Zur Erstellung von Tilemaps wurde das Programm „Tiled“ verwendet.

## 4.4 Versionsverwaltung

Zur Versionsverwaltung wurde die git-Technologie verwendet. Hier wurde ein Repository auf GitHub (vgl. <https://www.github.com/> ) erstellt. Der Vorteil ist, dass parallel an dem Projekt gearbeitet werden kann, und der Source-Code in der aktuellsten Version zentral verfügbar ist. Des Weiteren können bei einer Entwicklung in die falsche Richtung oder einem unlösbar scheinenden Bug alte Entwicklungsstände zurückgeholt werden.

## 4.5 Konventionen

Es wurden folgende Naming-Konventionen vor dem Beginn des Projekts besprochen:

**CSS-Klassen:** Camel-Case, **Dateinamen** (inkl. Bilder und Audio, mit Ausnahme von Klassen): kleingeschrieben, mehrere Wörter mit Bindestich verbunden, **Dateien die nur Klassen sind** und keine Komponenten sind: Camel-Case mit großem Anfangsbuchstaben, **Variablen und Methoden**: Camel-Case.

# 5 Architektur / Implementierung

## 5.1 Aufbau des Projektes

Das Projekt ist übersichtlich und erweiterbar aufgebaut. Die verschiedenen Dateien sind nach verschiedenen Funktionen getrennt und liegen in nach Funktionen sinnvoll benannten Ordnern. Die Projektstruktur (s. Anhang TODO) ist folgendermaßen aufgebaut: Die Quelltext-Dateien liegen im Ordner „/src/“, die Assets (Bilder, Audio, Tilemaps) liegen im Ordner „/assets/“, die verwendeten Bibliotheken unter „/lib/“ und die Dokumenation unter „/doc/“.

## 5.2 Einstiegspunkt

Zunächst wurde in der index.html ein Container erzeugt, in dem das Spielgeschehen ablaufen soll. Über den Script-Tag im Head wird die Phaser-Bibliothek geladen. Außerdem wurden die weiteren benötigten Abhängigkeiten geladen.

...

<script src="./lib/phaser.min.js"></script>

...

<body>

<div id="game-container"></div>

<script src="./src/managing/game-state.js"></script>

......

</body>

...

Abbildung : index.html Auszug

In der main.js wird zunächst ein config-Objekt (s. Abbildung 4: config-Objekt) initialisiert, welches dann dem Phaser-Game übergeben wird.

const game = new Phaser.Game(config);

In der Konfiguration werden beispielsweise der Container mitgegeben, die Größe des Spieles und wie das Spiel in verschiedenen Auflösungen skaliert wird. Außerdem wird der „physics“-Mode mitgegeben. Es wurde sich für Arcade entschieden, da die Hauptfunktionalitäten des Spieles um Bewegung und Collision handelt, welches durch „Arcade“ gewährleistet wird. Die Szenen werden jeweils in einem Array mitgegeben. Bei einer Szene handelt es sich um eine Klasse, die von Phaser.Scene erbt (s. Abbildung 2: Scene-Klasse) und die Methoden preload(), create() und update() implementieren kann.

class Stage1 extends Phaser.Scene {

...

}

Abbildung : Scene-Klasse am Beispiel Stage1

## 5.3 Scenes

Einer Scene wird zunächst im Konstruktor ein „Key“ übergeben, mit dem man auf diese Scene zugreifen kann.

constructor() {

super({ key: "Stage1" });

}

Abbildung : Scene-Konstruktor am Beispiel Stage1

Dieses Projekt wurde in drei verschiedene Arten von Scenes aufgeteilt: Preload, Menu und Stages, welche im Folgenden genauer betrachtet werden.

### 5.3.1 Preload

Die Preload-Szene wird als erste Szene beim Aufrufen der Applikation ausgeführt, da sie im config-Objekt an erster Stelle kommt. Sie implementiert die preload()-Methode und dient dafür alle Assets zu laden.

preload() {

//load spritesheet + tileset

this.load.spritesheet("dude", "assets/sprites/character.png", {

frameWidth: 48,

frameHeight: 48

});

this.load.image("tileset", "assets/tilesets/Platformer tiles.png");

//load menu assets

this.load.image("options\_button", "./assets/gui/options\_button.png");

this.load.image("play\_button", "./assets/gui/play\_button.png");

...

}

Abbildung : GamePreload preload()-Methode Ausschnitt

Die Assets (Bilder, Sounds, Tilemaps) beim Start der Applikation zu laden hat den Vorteil, dass diese während des Spiels nicht nachgeladen werden müssen und dementsprechend die Performance nicht darunter leidet. Dies führt dazu, dass das Spiel zum Starten eine gewisse Zeit benötigt. Damit der Benutzer weiß, dass das Spiel lädt, wurde auch ein Ladebalken implementiert (s. Abbildung 8: LoadingBar()-Methode). Wurde alles geladen, wird die Menu-Scene gestartet.

create() {

this.scene.start("Menu");

}

Abbildung : Start Menu-Scene

### 5.3.2 Menu

Die Menu-Scene implementiert die Methoden create()und update(). Die update-Methode bewegt hierbei lediglich den Hintergrund. In der create-Methode werden die jeweiligen Menüpunkte als Bild geladen.

let playButton = this.add.image(

this.game.renderer.width / 2,

this.game.renderer.height \* 0.45,

"play\_button"

);

Abbildung : Menüpunkt laden am Beispiel vom "Play"-Button

Außerdem wird die Interaktion mit den jeweiligen Menüelementen implementiert. Als Indikator wird hier ein Sprite neben dem jeweiligen Menüpunkt gezeigt um zu signalisieren, welcher Menüpunkt angeklickt wird. Durch „Pointer“-Events ist auch eine Touch-Eingabe möglich.

playButton.setInteractive();

playButton.on("pointerover", () => {

hoverSprite.setVisible(true);

hoverSprite.play("walk");

hoverSprite.x = playButton.x - playButton.width;

hoverSprite.y = playButton.y;

});

playButton.on("pointerout", () => {

hoverSprite.setVisible(false);

});

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

practiceMode = false;

this.music.stop();

this.scene.start("Stage1");

});

Abbildung : Interaktion von Menüpunkten am Beispiel vom "Play"-Button

Außerdem wird der gesetzte lokale Highscore, falls gesetzt, aus der localStorage geladen und angezeigt.

Das Select-Level-Menü ist nach demselben Schema aufgebaut.

### 5.3.3 Stages

Es gibt vier Stages die jeweils die create() und update()-Methode implementieren. In jeder Stage wird zunächst die Methode initBackgroundLayerArea() aufgerufen um den Hintergrund zu laden, sowie createBaiscLevelSetup(scene) aufgerufen.

createBasicLevelSetup(this);

Die Methode createBasicLevelSetup(scene), die in der Datei stage-managing.js implementiert ist, bietet einen Standardaufbau für die jeweiligen Stages. Hier werden die jeweiligen Layer aus Tiled geladen, die bei den Stages gleich sind (Boden, Spikes, Finish, Gravity), die Spielfigur initialisiert, die Animationen, die Texte erstellt und die jeweiligen Inputs definiert. Jede Scene hat dann noch die Möglichkeit in ihrer create()-Methode für diese Szene spezifische Sachen aufzurufen. Beispielsweise hat die Stage1 noch einen Info-Text, der kurz den Input des Spiels erklärt.

Die update()-Methode der jeweiligen Stage-Scenes ist dafür verantwortlich den Input zu verarbeiten, die Win-/Lose-Condition zu überprüfen und verschiedene Sachen, wie z.B. in Stage 3 die Laufgeschwindigkeit zu verändern.

Außerdem wird der Parallax-Effekt im Hintergrund durch die update()-Methode gesteuert.

update(time, delta) {

...

moveTilesArea1(this);

...

if (this.player.body.blocked.down) {

playJumpSound(this);

this.player.setVelocityY(-1100);

}

}

Abbildung : update()-Method Auszug

## 5.4 JavaScript

Um die Scenes nicht übermäßig groß zu machen, Duplizierung von Code zu vermeiden und Quelltext an die sinngemäße Stelle zu schreiben, wurden verschiedene JavaScript-files erstellt und Quelltext in diese ausgelagert.

### 5.4.1 game-state.js

Das game-state.js File hält ein Objekt, welches den aktuellen Spielstand (aktuelles Level und Versuche hält). Dieser wird in die localStorage geschrieben, damit man das Spiel aufhören und beim erneuten öffnen der Applikation dort weitermachen kann, wo man aufgehört hat. Außerdem hält es ein Objekt, welches benutzt wird, um den Spielstand zurückzusetzen. Des Weiteren ist ein Flag enthalten, welches indiziert, ob man in einem aktuellen Spiel spielt oder ob man den Practice-Mode benutzt.

### 5.4.2 state-managing.js

Das state-managing.js-File enthält Methoden, die für die jeweilgen Stages interessant sind. So enthält es jeweils Methoden um den Hintergrund einer Stage zu initialiseren oder das Parallax-Movement des Hintergrunds für die Stages. Außerdem hält es die oben genannte createBasicLevelSetup()-Methode und

### 5.4.3 music-config.js, constants.js, util.js

Die music-config.js, constants.js und util.js sind kurze Helferdateien, die Konstanten wie die Größe des Game-Fensters, die Konfiguration für die Musik und Sounds und Methoden wie die zufällige Generierung einer Zahl in einem bestimmten Zahlenraum enthalten.

## 5.4 Skalierung

Die Skalierung des Games wird über den von Phaser implementierten ScaleManger erledigt. Im config-Objekt besteht die Möglichkeit verschiedene Parameter mitzugeben. Es wurde sich in diesem Projekt für den Modus Phaser.Scale.FIT entschieden mit der Bildschirmgröße des jeweiligen Geräts als Breite und Höhe des Spiels. So versucht Phaser das Spiel an jeweilige Endgeräte zu skalieren, was für mobile Geräte, Tablets und Desktops in diesem Spiel jeweils gut funktioniert hat.

scale: {

mode: Phaser.Scale.FIT,

autoCenter: Phaser.Scale.CENTER\_BOTH,

width: GAME\_WIDTH,

height: GAME\_HEIGHT

},

Abbildung : scale-Property im config-Objekt

## 5.5 JSON-Tilemaps

Aus Tiled werden die Tilemaps, die für die jeweiligen Stages verwendet wurden als .json-Struktur exportiert (s. Anhang Abbildung 18: Auszug Tilemap .json-Export). So kann man über Phaser auf die jeweilge Struktur zurückgreifen und auf die jeweils mitgegebenen Attribute reagieren. Hat ein tile beispielsweise die property „collides“ kann Phaser davon ein Tile machen, mit dem der Spieler kollidiert.

scene.belowLayer.setCollisionByProperty({

collides: true

});

...

scene.physics.add.collider(scene.player, scene.belowLayer);

Abbildung : CollisionByProperty

## 5.6 Collision-Debugging

Um die Kollisionen zu debuggen, wurde ein jeweiliges Layer in der Farbe verändert. So konnte man sehen mit welchen Objekten der Spieler jeweils kollidieren sollte und so auf Richtigkeit und Abfragegenauigkeit überprüfen.

this.spikeLayer.renderDebug(debugGraphics, {

tileColor: null, // Color of non-colliding tiles

collidingTileColor: new Phaser.Display.Color(243, 134, 48, 255), // Color of colliding tiles

faceColor: new Phaser.Display.Color(40, 39, 37, 255) // Color of colliding face edges

});

Abbildung : Collision-Debugging Implementierung

Für ein Beispiel, wie die Welt mit dem Code aus Abbildung 11 aussieht, siehe im Anhang Abbildung 21: Collision Debugging.

# 6 Projektplanung und Arbeitsteilung

## 6.1 Zeitlicher Ablauf und Arbeitsteilung

Der zeitliche Ablauf wurde in im Anhang unter i. Projektplanung mithilfe eines Gantt-Diagramms dargestellt. Die Arbeitsteilung erfolgte zunächst nach dem was derjenige sich zutraute. So wurde die Architektur des Programmes an den Entwickler gegeben, der das meiste Vorwissen hatte, allerdings immer in Absprache mit den Gruppenmitgliedern. Nach und nach konnten die Aufgaben besser verteilt werden und in fester Absprache konnten einzelne Teilpakete des Projektes getrennt bearbeitet werden. Am Ende zum „Zusammenfügen“ der Puzzleteile und Testen wurde die Arbeit wieder gemeinsam erledigt. Das Projekt wurde in erfolgreicher Zusammenarbeit abgeschlossen.

# 7 Post Mortem

## 7.1 Was hast gut funktioniert

Die Komponentenbasierte Architektur wurde sehr ordentlich durchgeplant und hat letztendlich auch den Rahmen des Projektes getragen. Durch den strukturierten Aufbau war es leicht neue Komponenten hinzuzufügen und die Arbeit aufzuteilen.

Die Versionsverwaltung sorgte ebenfalls für ein gutes teamorientiertes Arbeiten und einen Austausch von Entwicklungsergebnissen, ohne dass man nebeneinandersitzen musste. Dadurch hat die Aufteilung der Aufgaben auch gut funktioniert und die Arbeit gleichmäßig aufgeteilt.

Die Versionsverwaltung sorgte auch einmal dafür, dass Projekt zu „retten“, als die Applikation sich immer in einer Endlosschleife befand, und nicht herausgefunden werden konnte, woran das Problem lag. Hier wurde das Projekt auf den vorherigen Commit zurückgespielt.

## 7.2 Was hat nicht funktioniert

Durch das dynamische Laden der Daten gab es etwas größere Schwierigkeiten mit der Asynchronität. Es musste sich Wissen über „Promises“ angeeignet werden, welches im Nachhinein schwierig war in das Projekt einzupflegen. So kommt es sehr selten immer noch vor, dass das Projekt nicht startet, da die Daten nicht rechtzeitig geladen werden. Ein einfacher Reload löst das Problem allerdings meistens schon.

Des Weiteren hat die Eigengestaltung von Grafiken nicht funktioniert, da zeitlich wenig Platz und generell viel Misserfolg in der Kreation stattfand. So wurde entschieden auf fertige Icons, Bilder und Musik aus dem Internet zuzugreifen und diese zu verwenden und die Autoren zu kreditieren. Das macht das Spiel nicht ganz „stimmig“ auch wenn auf identischen Stil der Icons geachtet wurde.

Des Weiteren gab es kleinere Probleme mit dem Balancing. Dadurch dass die Aktionen zufällig geladen werden, ist es möglich, dass ein Spiel unschaffbar ist.

## 7.3 Rückblick

Obwohl die Architektur dieses Projekts gut getragen hat und gut geplant und strukturiert wurde, würde nächstes Mal mehr Wert auf Data-Binding gelegt werden. Jede Änderung muss über den UI-Manager aufgerufen werden. Einen Weg die Oberfläche zu aktualisieren, wenn sich die Daten ändern, würde die Menge an Code deutlich schmälern und das Projekt leichter erweiterbar machen.

Des Weiteren muss beim nächsten Projekt darauf geachtet werden, Probleme von Asynchronität und großen Dateien (Stichworte Lazy-Loading, Pre-Loading, Promises) mit einzuplanen und die Architektur und das Gesamtprodukt stabiler zu machen und zu verbessern.

Außerdem würde, wenn dieses Projekt erneut durchgeführt werden würde, mehr Wert auf Grafik gelegt werden, um eine stimmigere Atmosphäre und ein besser abgestimmtes Produkt zu erstellen.

Um das Balancing zu verbessern, ist es gegebenenfalls sinnvoll den Aktionen eine Häufigkeit zu geben, damit mehr Kontrolle vorhanden ist, die Aktionen und Belohnungen besser ins Gleichgewicht zu bringen und das Spiel auch durch mehr Strategie zu gewinnen.

## 7.4 Ausblick

Denkbare Erweiterungen des Spiels sind weitere Charaktere, die unterschiedliche Schwierigkeitsstufen beinhalten. Diese Charaktere können wiederum mit weiteren Aktionen, die charakterspezifisch sind erweitert werden.

Außerdem können die Locations und die Items erweitert werden, um mehr Möglichkeiten im Spiel zu haben und das Universum zu erweitern.

Denkbar ist auch, eine neue Art von Ressourcenleiste hinzuzufügen, um die Komplexität des Spiels zu erweitern und interessanter und ggf. schwieriger oder leichter zu machen.

# Anhang

# i. Projektplanung

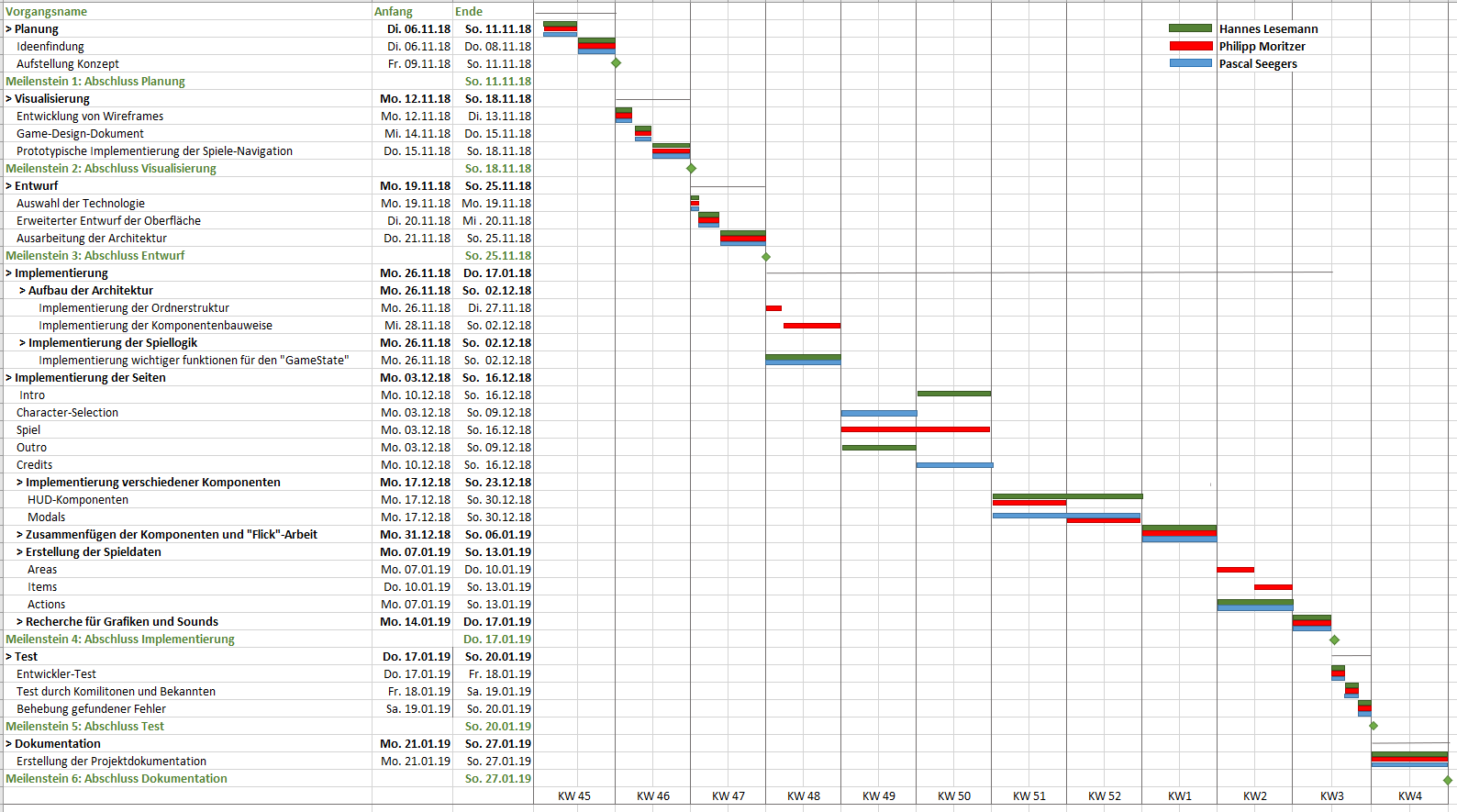


Abbildung 12: GANTT-Diagramm Planung

# ii. Projektstruktur

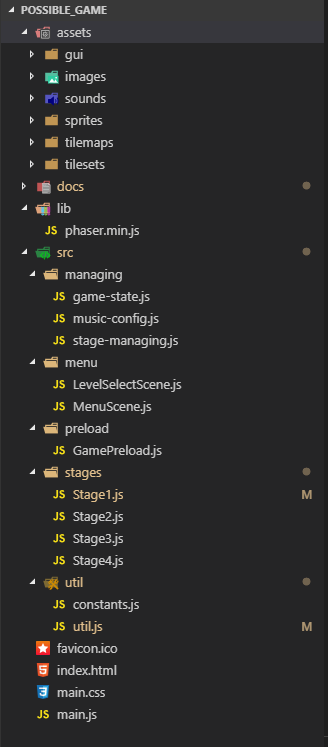


Abbildung : Projektstruktur

# iii. Config-Objekt

const config = {

type: Phaser.AUTO,

scale: {

mode: Phaser.Scale.FIT,

autoCenter: Phaser.Scale.CENTER\_BOTH,

width: GAME\_WIDTH,

height: GAME\_HEIGHT

},

parent: "game-container",

backgroundColor: "#E4F6F8",

zoom: 1,

physics: {

default: "arcade",

arcade: {

tileBias: 32,

gravity: { y: 5000 }

}

},

scene: [

GamePreload,

MenuScene,

LevelSelectScene,

Stage1,

Stage2,

Stage3,

Stage4

]

};

Abbildung : config-Objekt

# iv. LoadingBar()-Methode

function loadingBar(scene) {

let loadingBar = scene.add.graphics({

fillStyle: {

color: 0x000000 // white;

}

});

scene.load.on("progress", percent => {

loadingBar.fillRect(

0,

scene.game.renderer.height / 2,

scene.game.renderer.width \* percent,

50

);

});

}

Abbildung : LoadingBar()-Methode

# iii. Tiled Map Building

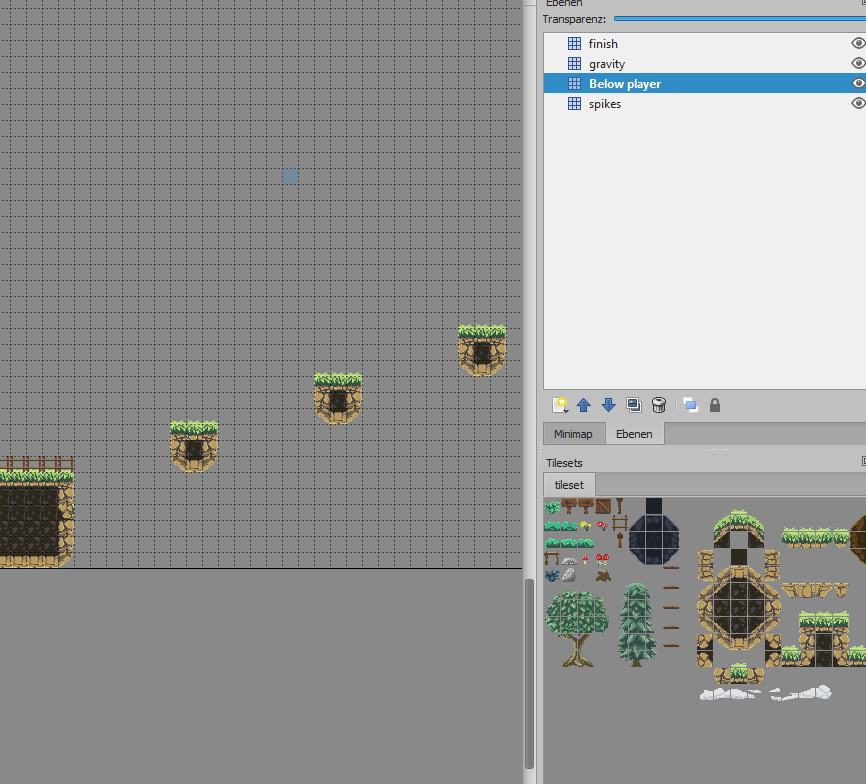


Abbildung : Erstellen von Tiledmaps

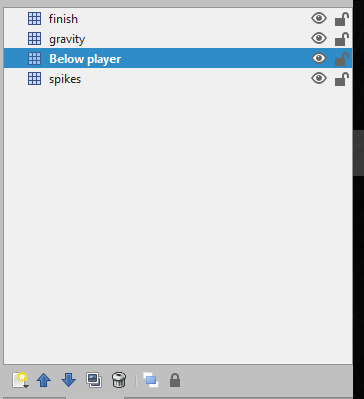


Abbildung : Tiled layers

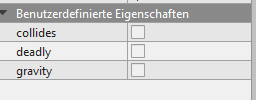


Abbildung : Tiled Attribute

# iv. Tileset



Abbildung : Tileset

# vi. Auszug Tilemap als .json

{ "height":150,

"infinite":false,

"layers":[

{

"data":[154, 155, 155, 3221225627, 155, 155, 3221225627, 155, 155, 3221225627, 155, 155, 3221225627, 155, 155, 3221225627, 155, 155, 3221225627

...

"height":150,

"id":16,

"name":"gravity",

"opacity":1,

"type":"tilelayer",

"visible":true,

"width":550,

"x":0,

"y":0

...

"id":35,

"properties":[

{

"name":"collides",

"type":"bool",

"value":false

}]

},

...

Abbildung : Auszug Tilemap .json-Export

# iv. Collision debugging

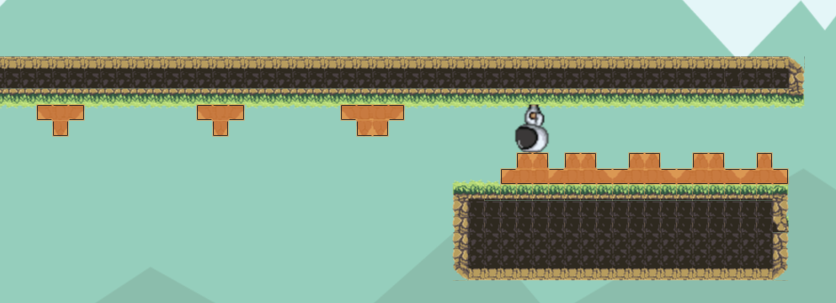


Abbildung : Collision Debugging

# iii. Desktop Menu Screenshots



Abbildung : Desktop-Hauptmenü



Abbildung : Dektop-Level-Select

# iv. Desktop Game Screenshots



Abbildung : Stage 1 – Screenshot



Abbildung : Stage 2 – Screenshot

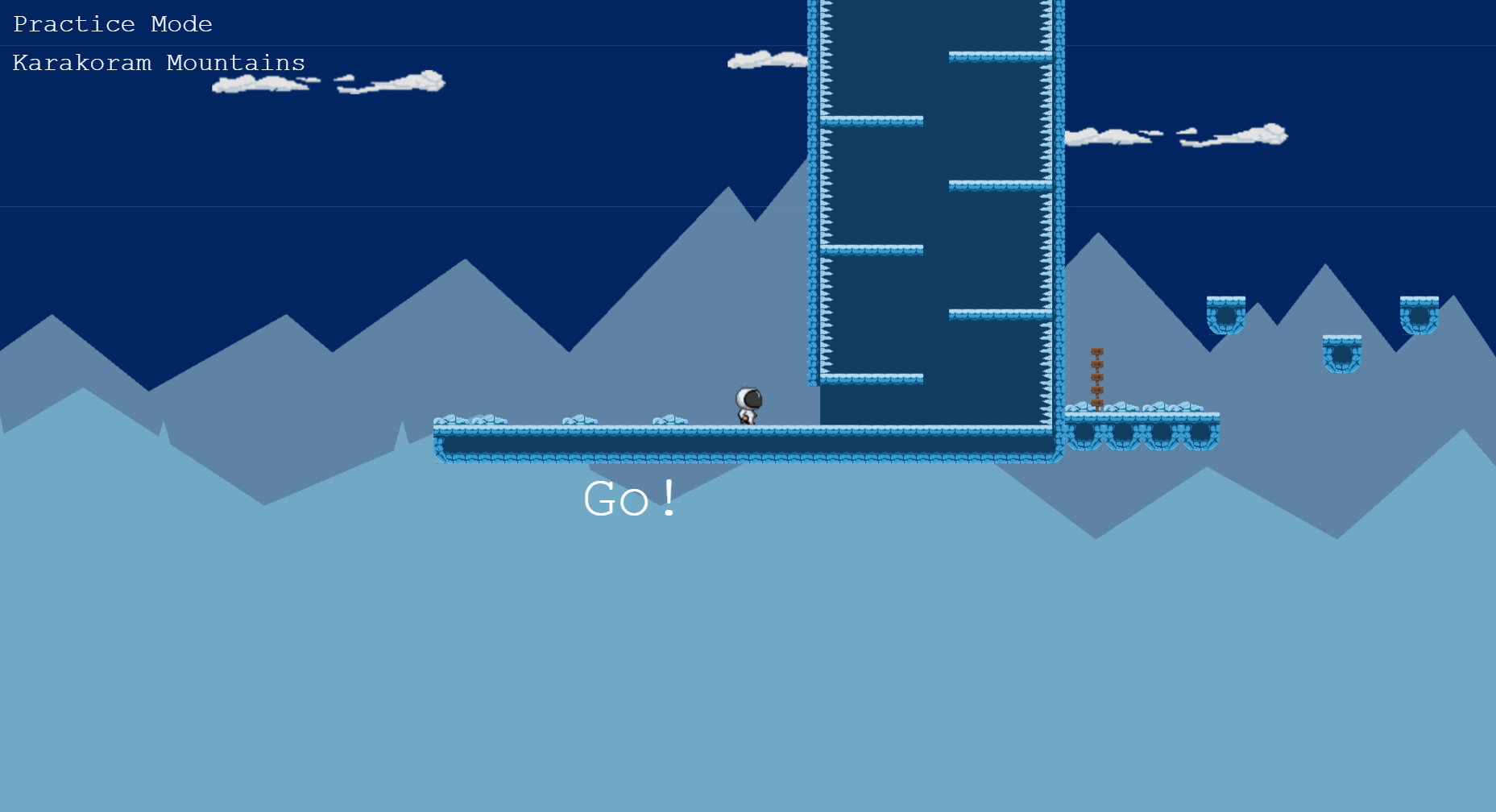


Abbildung : Stage 3 – Screenshot

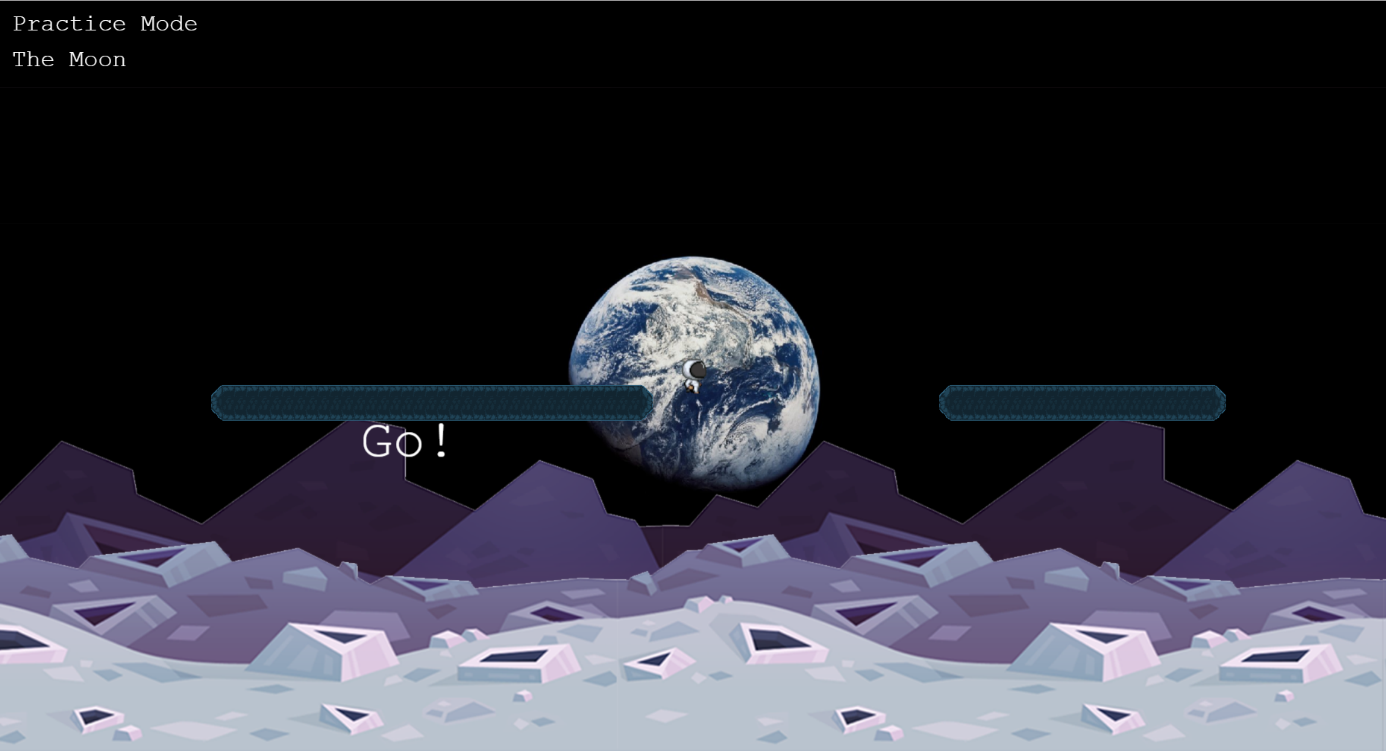


Abbildung : Stage 4 – Screenshot

# v. Mobile Ansichten

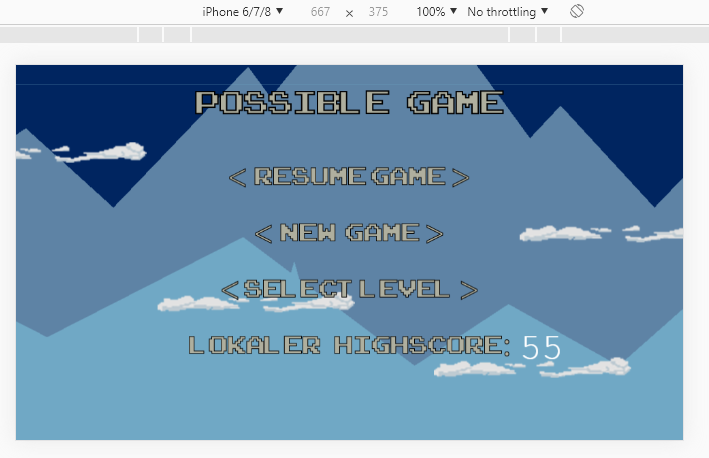


Abbildung : Menu-Screenshot Phone

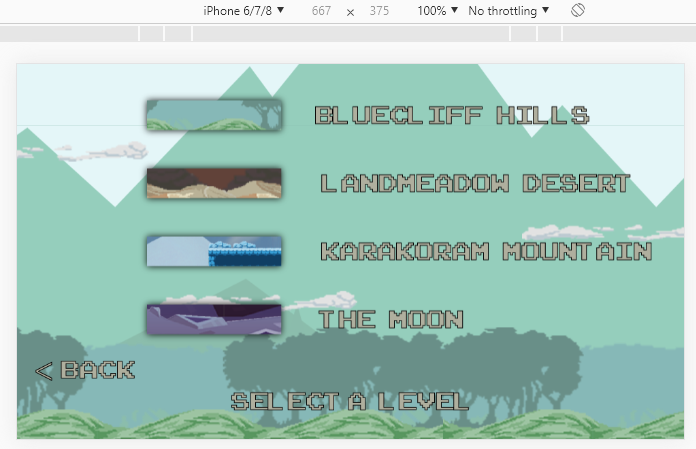


Abbildung : Select-Level Screenshot Phone

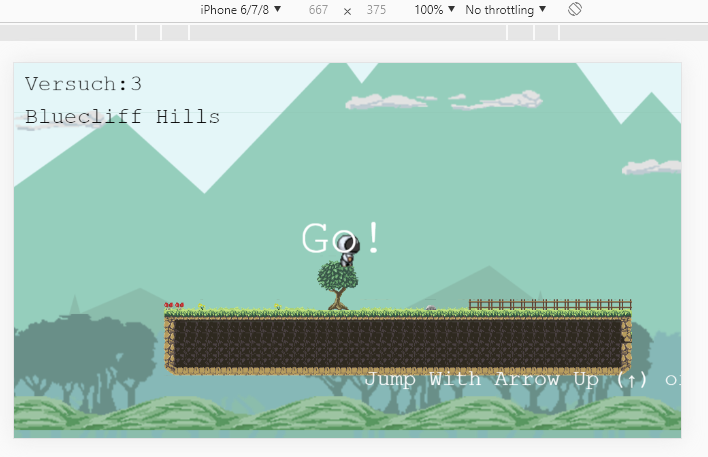


Abbildung : Game Screenshot Phone

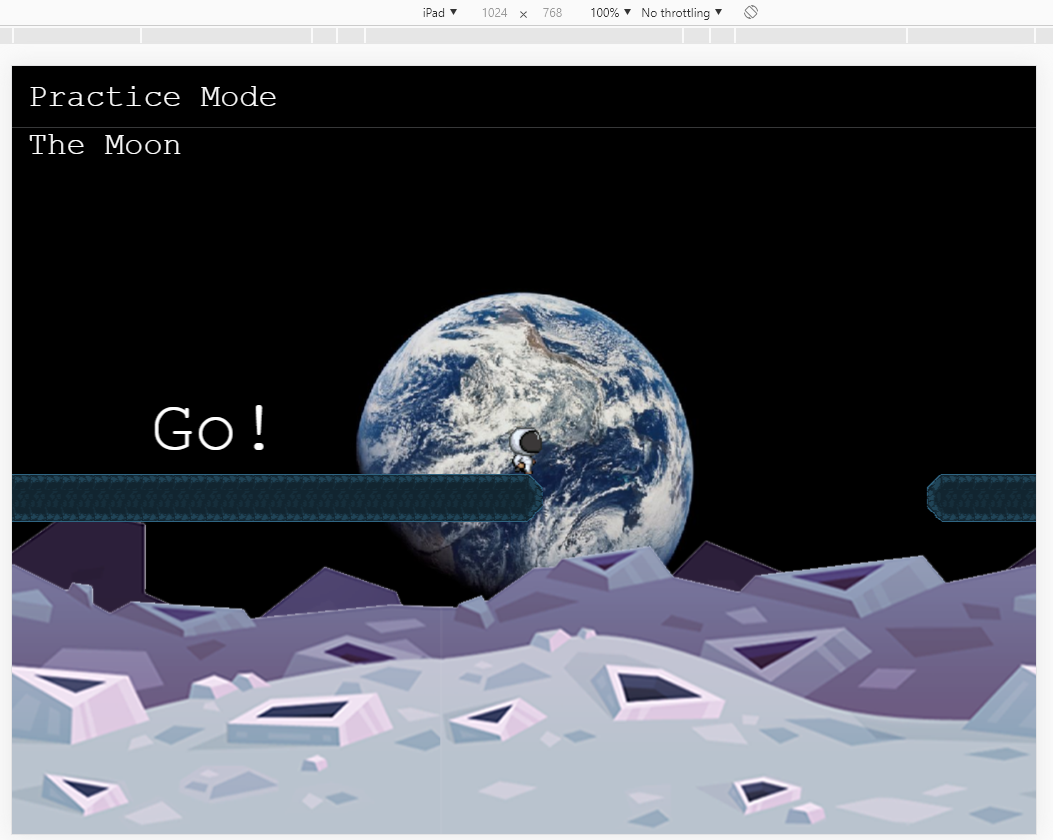


Abbildung : Game Screenshot Tablet