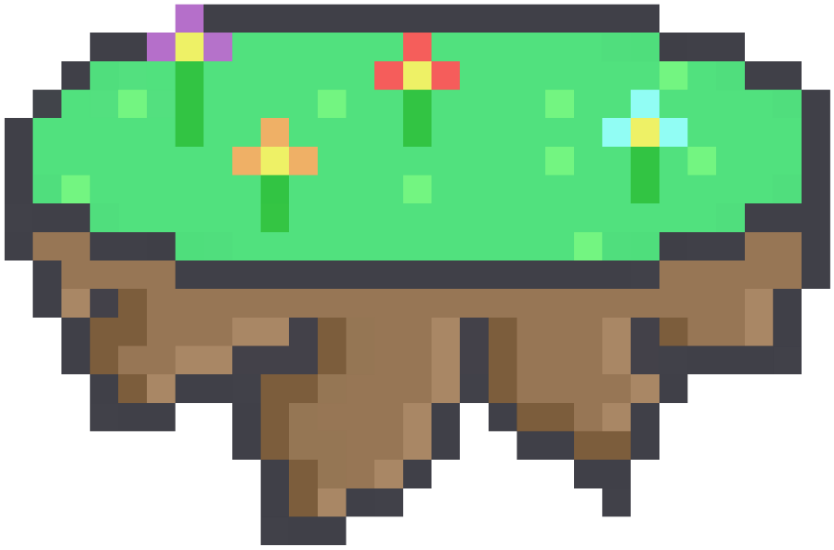

Floating Island: ein Multiplayer-Browser-Spiel

Maturaarbeit von Shnyar Muhamad, N19a

Betreuungsperson: Philipp Imhof

November 2022

Kantonsschule Solothurn

Vorwort

Schon seit klein auf interessiere ich mich für Computer, mit grossem Fokus auf Videospiele. Ich verbrachte früher grosse Teile meiner Freizeit damit, Videospiele zu spielen. Dabei fragte ich mich immer wieder, wie Videospiele eigentlich funktionierten. Vor ein paar Jahren packte mich diese Frage so sehr, dass ich entschied, mich darüber zu informieren. Dadurch stiess ich auf die Welt des Programmierens und der Spieleentwicklung. Schnell packte mich das Programmieren und das Entwickeln von Videospielen. Ich entwickelte über die Jahre mehrere kleine Videospiele, welche ich mit meinen Kollegen teilte. Da mir das Programmieren von Videospielen grosse Freude bereitet, entschied ich mich, für diese Maturaarbeit ein Videospiel zu programmieren. Dabei setzte ich mir das Ziel, ein Spiel zu programmieren, welches über einen Multiplayer verfügt und auf zufällig generierten Welten gespielt wird, da dies beides Funktionalitäten sind, mit welchen ich mich noch nie vorher auseinandergesetzt habe.

Einen Dank möchte ich an meine zwei Geschwistern geben, welche mich bei dem Schreiben dieser Maturaarbeit unterstützten, und an Simon Thalmann, der mir seinen Computer als Server für die Website des Spiels zur Verfügung stellt.

Inhaltsverzeichnis

1. Einleitung

In der heutigen Zeit werden überall Computer genutzt, um das Leben der Menschen zu vereinfachen. Sei das, um die Arbeitseffizienz zu erhöhen, soziale Kontakte aufrechtzuerhalten oder für Unterhaltungszwecke. Dabei machen viele letzteres mithilfe von Videospielen. Die Videospielindustrie ist dabei einer der grössten Bereiche in der Unterhaltungsindustrie und ist stetig am Wachsen. Doch obwohl Millionen von Menschen jeden Tag Videospiele spielen, ist den wenigsten bewusst, wie ein solches Spiel eigentlich funktioniert. Ein Videospiel ist wie jedes andere Computerprogramm, welches Code auf dem Computer ausführt und Output auf das Display generiert. Doch für diesen Prozess muss im Hintergrund konstant eine extreme Rechenleistung aufgewendet werden, von dem der Spieler gar nichts erfährt.

Um die Funktionsweise eines Spiels zu veranschaulichen, entschied ich mich im Rahmen dieser Maturaarbeit ein Multiplayer-Spiel mit unendlich grossen Spielwelten samt Server zu programmieren, welches im Browser unter der Domain https://floatingisland.ch [Stand: 03.11.2022] spielbar ist. Dabei wurde bei der Entwicklung auf die Hilfe einer Spiel-Engine verzichtet.

Bei „Floating Island“ handelt es sich um ein Top-Down Survival-Spiel, bei dem sich der Spieler auf einer zufällig generierten Welt befindet. Diese Welt besteht aus unendlich vielen fliegenden Inseln, auf denen Ressourcen gesammelt werden kann, um sich danach von Feinden verteidigen zu können. Da es ein Multiplayer-Spiel ist, kann das Spiel allein, sowie mit Freunden gespielt werden. Alles, was zum Spiel gehört, das heisst Grafiken, Code und Weiteres, wurde selbst gemacht. Die Entwicklung des Spiels dauerte etwa acht Monate und der Code vom Spiel zusammen mit dem Server besteht letztendlich aus etwa 4‘000 Zeilen Code.[1]

In der folgenden Arbeit wird die allgemeine Funktionsweise eines Multiplayer-Spiels angeschaut, sowie das Prinzip von unendlichen Spielwelten erklärt. Anschliessend wird, um das Ganze noch anhand eines Beispiels zu sehen, die Funktionsweise der wichtigsten Elemente von Floating Island genauer erklärt, um der Frage nachzugehen: „Wie funktioniert ein Multiplayer-Browser-Spiel mit zufällig generierten Welten?“.

[1] Github. FloatingIsland. Verfügbar unter https://github.com/ShnyarM/FloatingIsland [Stand: 02.11.2022]

2. Theorie

2.1 Multiplayer

Wird auf dem eigenen Computer ein Videospiel gespielt, so wird der Code des Spiels auf dem eigenen Rechner ausgeführt. Das heisst, dass die Daten für das Spiel bei einem selbst gespeichert sind. Wenn ein Spieleentwickler jetzt ein Multiplayer-Spiel haben möchte, muss ein Weg gefunden werden, die Daten der einzelnen Spieler miteinander auszutauschen. Das Prinzip eines Multiplayer-Spiels ist es die Daten der verschiedenen Spieler miteinander zu synchronisieren. Dadurch entsteht die Illusion, dass zusammen in einem Spiel gespielt wird, obwohl die Spiele getrennt voneinander laufen. Diese Verbindung zwischen den Spielern wird, in den meisten Fällen, über das Internet gemacht, da dies, für die meisten, oft der einzige Weg ist, eine zuverlässige Verbindung über weite Distanzen aufzubauen. Dies ist auch der Grund, warum Multiplayer-Spiele eine Internetverbindung benötigen. Dabei gibt es aber verschiedene Wege, eine Verbindung über das Internet aufzubauen.

2.1.1 Peer-to-Peer Netzwerk

Mit einem Peer-to-Peer Netzwerk wird ein Netzwerk bezeichnet, bei dem die Teilnehmer direkt miteinander verbunden sind und die gleiche Rechte besitzen.[2] Im Kontext eines Multiplayer-Spiels bedeutet das, dass die Rechner der Spieler miteinander verbunden sind und sich die Daten direkt zuschicken. Dabei wird es oftmals so gemacht, dass ein Spieler der Host des Spiels ist. Als Host läuft der gesamte Datenfluss über ihn. Die Spiele der anderen Spieler senden die Daten dem Host, der diese Daten an die anderen Spiele weiterleitet (vgl. Abb. 1). Ein Peer-to-Peer Netzwerk kann aber auch ohne Host aufgebaut werden. In diesem Fall wären alle Rechner miteinander verbunden und würden das Spiel zusammen am Laufen halten (vgl. Abb. 2).

Ein Peer-to-Peer Netzwerk ist nicht auf einen zusätzlichen Server angewiesen. Dies ist vor allem für die Entwickler eines Videospiels nützlich, da sie nicht selbst einen Server zur Verfügung stellen müssen. Zudem können Spieler die Multiplayer-Funktionalität eines Spiels auch gebrauchen, falls die Entwickler die Unterstützung des Spiels und dessen Server eingestellt haben. Nachteil dieser Art von Verbindung ist, dass die Stabilität dieses Netzwerks nicht garantiert ist. Die Spieler, und gegebenenfalls vor allem der Host, brauchen einen guten und stabilen Internetanschluss, damit das Netzwerk gut funktioniert, was jedoch nicht immer garantiert ist. Zudem existieren Sicherheitsprobleme mit dieser Art von Verbindung, da der Host die Daten manipulieren und somit unfaire Vorteile erlangen könnte. Ausserdem hat der Host automatisch einen Vorteil in Spielen, wo eine geringe Latenz bevorzugt wird,

[2] XOVI. Was bedeutet Peer-to-Peer? Verfügbar unter https://www.xovi.de/was-bedeutet-peer-to-peer/ [Stand: 18.10.2022]

wie zum Beispiel in Ego-Shootern, da das Spiel über seinen Rechner läuft und er somit keine Latenz hat.

Beispiele von Spielen, welche eine Peer-to-Peer Verbindung benutzen, sind „GTA Online“, „Animal Crossing: New Horizons“ oder „Super Smash Bros. Ultimate“.[3]

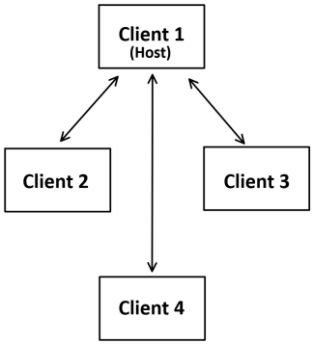

Abbildung 1: Peer-to-Peer Netzwerk mit Host, Spieler als „Client“ dargestellt

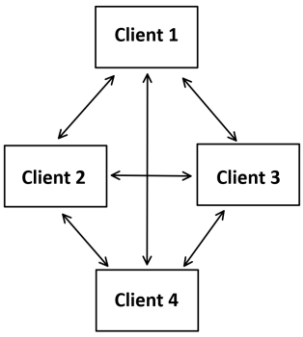

Abbildung 2: Peer-to-Peer Netzwerk ohne Host, Spieler als „Client“ dargestellt

2.1.2 Dedizierter Server

Eine weitere Art, eine Verbindung zwischen den Spielern eines Multiplayer-Spiels herzustellen, ist mithilfe eines dedizierten Servers. Dies ist heutzutage die meistbenutzte Art bei Videospielen. Mit einem Server wird ein Rechner in einem Netzwerk bezeichnet, der für andere im Netzwerk verbundene Teilnehmer bestimmte Aufgaben übernimmt.[4] Im Kontext eines Multiplayer-Spiels übernimmt der Server die Aufgabe, die einzelnen Spieler miteinander zu verbinden. Die Spieler tauschen ihre Daten nicht untereinander aus, sondern senden diese an den Server, der die Daten an alle Spieler zuschickt. Also ist ein Spieler, der bei einer Verbindung mit einem Server oft als „Client“ bezeichnet wird, nur mit dem Server verbunden und nicht mit den anderen Spielen (vgl. Abb. 3). Bei dieser Methode wird die Entwicklung der Software und die Bereitstellung des Servers in den meisten Fällen vom Spieleentwickler übernommen.

Für die Spieler selbst ist diese Methode angenehmer, da eine stabile Verbindung meistens garantiert ist. Für den Entwickler ist diese Methode jedoch teurer, da dieser einen Server bereitstellen muss, was wiederum Kosten verursacht. Deshalb besteht bei älteren Videospielen, welche einen Server benötigen, oft das Problem, dass die Multiplayer-Funktionalität nicht mehr funktioniert. Entwickler stellen Server von älteren Spielen oft ab, weil die Kosten den Ertrag überschreiten.

[3] Roxl, Rhett. (2021). What is Peer-to-Peer Gaming, and How Does it Work? Verfügbar unter https://vgkami.com/what-is-peer-to-peer-gaming-and-how-does-it-work/ [Stand: 18.10.2022]
[4] Duden. Server. Verfügbar unter https://www.duden.de/rechtschreibung/Server [Stand: 18.10.2022]

Beispiele von Spielen, welche einen dedizierten Server nutzen, sind „Fortnite“, „Minecraft“ oder „Among Us“.[5]

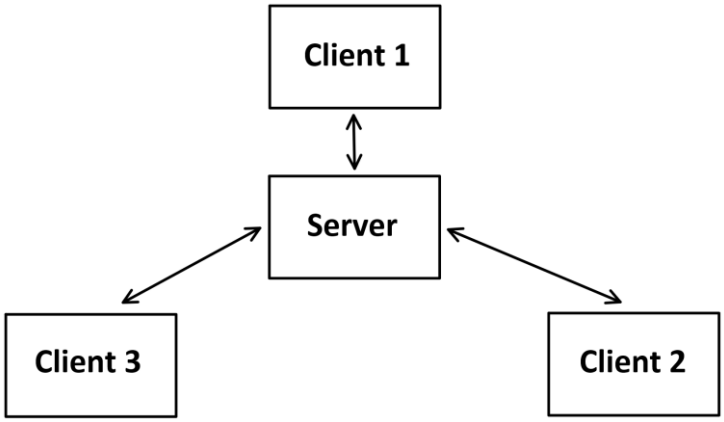

Abbildung 3: Netzwerk mit dediziertem Server, Spieler als „Client“ dargestellt

2.2 Prozedurale Synthese

Mit der Prozeduralen Synthese wird eine Methode bezeichnet, welche während der Ausführung des Computerprogramms Programminhalte erzeugt, ohne dass diese vorher vom Entwickler festgelegt wurden. Dabei folgt die Generierung deterministischen Algorithmen, womit immer wieder dieselben Inhalte erzeugt werden können. Mit prozeduraler Synthese können verschiedene Programminhalte, wie zum Beispiel Texturen, 3D-Objekte und virtuelle Welten, generiert werden.[6]

2.2.1 Beispiel von Prozedurale Synthese anhand von „Minecraft“

Ein Beispiel für prozedurale Synthese in einem Videospiel sind die Welten in „Minecraft“ (vgl. Abb. 4). Diese werden während des Spielens von einem Algorithmus generiert. Somit wird es ermöglicht, auf scheinbar unendlich grossen Welten zu spielen, da die Teile der Welt erst generiert werden, wenn diese benötigt werden. Die Welten können beim Spielen zwar zufällig wirken, da die Generierung aber einem deterministischen Algorithmus folgt, sind diese nicht zufällig. Das einzig zufällige an der Generierung ist die Bestimmung des Startwerts, dem sogenannten „Seed“, welcher bei Bedarf auch manuell bestimmt werden kann. Mit demselben Startwert wird immer die gleiche Welt generiert, womit es einem ermöglicht wird, auf der gleichen Welt mehrmals zu spielen. Die Oberfläche der Welt wird dabei mit einer Perlin-Noise Funktion modelliert. Mit Perlin-Noise wird eine pseudozufällige Rauschfunktion bezeichnet. Mit der Perlin-Noise Funktion werden natürlich aussehende

[5] Roxl, Rhett. (2021). What is Peer-to-Peer Gaming, and How Does it Work? Verfügbar unter https://vgkami.com/what-is-peer-to-peer-gaming-and-how-does-it-work/ [Stand: 18.10.2022]
[6] Wikipedia. Prozedurale Synthese. Verfügbar unter https://de.wikipedia.org/wiki/Prozedurale_Synthese [Stand: 18.10.2022]

Strukturen von Höhen und Senken erzeugt, wodurch die Oberfläche der Welten im Spiel ein natürliches Aussehen erhält.[7]

Abbildung 4: Beispiel einer Minecraft-Welt

[7] Minecraft Wiki. Weltgenerierung. Verfügbar unter https://minecraft.fandom.com/de/wiki/Weltgenerierung [Stand: 18.10.2022]

3. Entwicklung von „Floating Island“

3.1 Auswahl der Laufzeitumgebung

Da für das Spiel entschieden wurde, mit einem dedizierten Server zu arbeiten, musste eine Laufzeitumgebung für den Server gesucht werden, bevor mit dem eigentlichen Programmieren angefangen werden konnte. Da das Spiel im Browser laufen sollte, war es klar, dass das Spiel in JavaScript geschrieben werden sollte. Aus diesem Grund wurde Node.js als Laufzeitumgebung gewählt. Node.js ist eine Open-Source-JavaScript-Laufzeitumgebung, die es einem ermöglicht, JavaScript Code ausserhalb eines Webbrowsers auszuführen.[8] Somit konnte das Spiel und der Server in JavaScript geschrieben werden, wodurch das Programmieren um einiges vereinfacht wurde.

3.2 Frameworks und Bibliotheken

3.2.1 p5.js

Für das Spiel wurde hauptsächlich mit der p5.js Bibliothek gearbeitet. Die p5.js JavaScript Bibliothek ist eine Ansammlung von vorgeschriebenem Code, welche das Erstellen von interaktiven Visualisierungen im Webbrowser vereinfacht.[9] Es bietet einem die Möglichkeit, ohne viel Arbeit Elemente auf einem Canvas zu zeichnen und User Inputs aufzunehmen, weswegen es für das Programmieren eines Spiels sehr hilfreich ist. Ausserdem habe ich persönlich diese Bibliothek schon öfters benutzt. Das bedeutet, dass ich die Bibliothek nicht noch lernen musste, was die Entscheidung für diese Bibliothek noch weiter vereinfachte.

3.2.2 Socket.IO

Für die Kommunikation zwischen den Clients und dem Server wurde vor allem die Socket.IO Bibliothek benutzt. Dabei gibt es zwei Versionen der Bibliothek: eine für den Client und eine für den Server. Die Socket.IO Bibliothek ermöglicht einem eine bidirektionale Echtzeit-Kommunikation zwischen dem Client und dem Server. Dies wird mithilfe von WebSockets ermöglicht. Zusätzlich wird die Stabilität der Verbindung garantiert, indem es, falls ein WebSocket nicht möglich ist, auf HTTP Long Polling zurückfällt oder, im Falle einer verlorenen Verbindung, sich automatisch wiederverbindet.[10] Somit kann ein stabiler Austausch von Daten zwischen den Clients und dem Server mit nur kleiner Verzögerung ermöglicht werden, was für ein Multiplayer-Spiel essenziell ist.

---

[8] Wikipedia. Node.js. Verfügbar unter https://de.wikipedia.org/wiki/Node.js [Stand: 18.10.2022]
[9] Codeacademy. Introduction to Creative Coding. Verfügbar unter https://www.codecademy.com/learn/learn-p5js/modules/p5js-introduction-to-creative-coding/cheatsheet [Stand: 18.10.2022]
[10] Socket.IO. Introduction. Verfügbar unter https://socket.io/docs/v4/ [Stand: 18.10.2022]

3.2.3 Express.js

Für den Webserver der Website war zuerst geplant, einen Apache HTTP Server zu benutzen. Danach wurde jedoch entschieden, den Webserver und den Spielserver, der die Multiplayer-Sitzungen ermöglicht, in einem Skript laufen zu lassen, um die Arbeit zu vereinfachen. Da schon Node.js als Laufzeitumgebung entschieden wurde, wurde schliesslich das Express Framework für Node.js benutzt, um den Webserver laufen zu lassen. Das Express Framework enthält zudem noch zusätzliche Features, wie die Handhabung von HTTP GET- und POST-Anfragen, welche gebraucht wurden.

3.2.4 Axios

Um das Erstellen von HTTP-Requests zu vereinfachen, wurde auf der Client-Seite die Axios Bibliothek benutzt. Axios ist ein HTTP-Client für Node.js und für den Browser, der auf Promises basiert ist.[11] Obwohl die Bibliothek auch für Node.js verfügbar ist, lauft die Bibliothek nur auf der Client-Seite, da auf dem Server keine HTTP-Requests gemacht werden.

3.3 Synchronisation der Spielerdaten

Um in einem Multiplayer-Spiel das Gefühl zu erwecken, dass zusammen mit anderen Spielern gespielt wird, ist eines der wichtigsten Features, die anderen Spieler in seinem eigenen Spiel sehen zu können. Deswegen war die Synchronisation der Spieler das erste Feature, welches im Spiel implementiert wurde. Für die Illusion eines anderen Spielers im eigenen Spiel, muss der andere Spieler an der Stelle in der Spielwelt gezeichnet werden, an der sich dieser in seinem Spiel befindet. Zusätzlich müssen noch weitere Merkmale, wie zum Beispiel die Richtung, in welche der Spieler schaut, berücksichtigt werden, um die Illusion noch glaubhafter zu machen.

Da die Spiele der einzelnen Spieler jedoch nicht miteinander verbunden sind, ist dies nur möglich, indem diese Informationen untereinander ausgetauscht werden. Deswegen wird bei jeder Veränderung einer für die Darstellung wichtigen Variable diese mit dem neuen Wert per Websocket an den Server gesendet. Dieser neue Wert wird anschliessend vom Server gespeichert und an die restlichen Spieler im Spiel gesendet. Die Spieler erhalten schliesslich diesen neuen Wert und speichern diesen selbst (vgl. Abb. 5). Diese Variablen kann das Spiel dann brauchen, um die anderen Spieler korrekt darzustellen. Gespeichert werden diese Variablen in Objekten. Für jeden fremden Spieler wird ein Objekt einer bestimmten Klasse kreiert. Diese beinhaltet alle Variablen, die gespeichert werden müssen. Diese Spieler-Objekte werden wiederum in einem Objekt gespeichert. Um die einzelnen

[11] Axios. Getting Started. Verfügbar unter https://axios-http.com/docs/intro [Stand: 18.10.2022]

Spieler identifizieren zu können, wird der Spieler unter dem Key seiner Socket-ID gespeichert. Die Socket.IO Bibliothek gibt jeder Verbindung eine bestimmte Kennzeichnung, die Socket-ID. Somit kann jedem Spieler eine ID zugeordnet werden, um die einzelnen Spieler zu identifizieren.

Der Server benutzt dieselbe Methode wie das Spiel, um die Daten der einzelnen Spieler zu speichern (vgl. Abb. 6). Der Server muss die Spielerdaten auch speichern, damit, im Falle einer Anfrage von einem Spieler nach den Daten eines anderen Spielers, der Server diese schicken kann. Dies ist zum Beispiel der Fall, wenn ein Spieler einem Spiel beitritt und die Daten der anderen Spieler benötigt. Zusätzlich braucht der Server diese Daten, um diese auf der Festplatte zu speichern. Aus diesem Grund gibt es zusätzlich bestimmte Variablen, welche dem Server zwar zugeschickt und von ihm gespeichert werden, jedoch nicht an die anderen Spieler weitergeleitet werden, da diese für die anderen Spieler keinen Nutzen haben. Diese Daten werden dann auf der Festplatte gespeichert, sobald der Spieler das Spiel verlässt.

```
//Updates data of other player when data is changed
function playerDataUpdate(data){
  //first key is id, rest is data with keyname being changed var
  for(let i = 1; i < Object.keys(data).length; i ++){
    //Get name of changed key and put in value
    players[data.id][Object.keys(data)[i]] = Object.values(data)[i]
    if(Object.keys(data)[i] == "x" || Object.keys(data)[i] == "y") players[data.id].movedThisFrame = true //for walking animation
  }
}
```

Abbildung 6: Clientseitiger Code für die Handhabung von einkommenden Spielerdaten

```
//Update data about Player and send to other Players, if serverOnly is true it wont be sent to other players
playerChange(socket, data){
  //keyname of value in data is changed var and value is new value
  for(let i = 0; i < Object.keys(data).length; i++){
    //Get name of changed key and put in value
    if(Object.keys(data)[i] != "serverOnly" && this.players[socket.id][Object.keys(data)[i]] != null)
      this.players[socket.id][Object.keys(data)[i]] = Object.values(data)[i]
  } //Emit to others
  if(!data.serverOnly) socket.to(this.room).emit("playerDataChange", {id: socket.id, ...data}); //Add id to data and send to other players
}
```

Abbildung 5: Serverseitiger Code für die Handhabung von einkommenden Spielerdaten

3.3.1 Speichern der Spielerdaten

Die Spielerdaten müssen auch auf der Festplatte gespeichert, damit der Spieler zu einem späteren Zeitpunkt mit dem gleichen Spielstand weiterspielen kann. Sobald der Spieler die Welt verlässt und die Daten gespeichert werden müssen, nimmt der Server die Daten und schreibt diese in eine JSON-Datei, welche speziell für den Spieler gemacht wird. Diese Datei befindet sich im selben Verzeichnis, wo die restlichen Dateien zu der Spielwelt gespeichert werden. Sobald der Spieler wieder die Welt betritt, liest der Server die Daten aus der JSON-Datei und sendet diese dem Spieler, wodurch er mit dem gleichen Spielstand, den er beim Verlassen der Welt hatte, weiterspielen kann.

3.4 Spielwelten

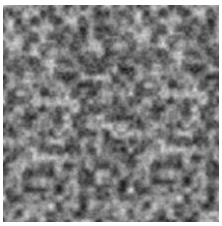

Abbildung 7: Zweidimensionale Perlin-Noise Karte

Für die Spielwelten wollte ich mit prozeduraler Synthese generierte Welten, welche bei jedem neuen Spielstand unterschiedlich ist. Sie sollten aus vielen fliegenden Inseln bestehen, welche voneinander getrennt sind. Zusätzlich sollten sie eine unendliche Grösse haben und bei Bedarf sollte der Spieler die gleiche Welt auf mehreren Spielständen haben können. Dies bedeutet, dass die Welten zufällig, jedoch auch duplizierbar sein müssen. Um dieses Problem zu lösen, wurde eine zweidimensionale Perlin-Noise benutzt. Ein zweidimensionales Perlin-Noise wird dabei als eine Karte vorgestellt, welche an jedem Punkt einen bestimmten Wert hat (vgl. Abb. 7). Diese Werte haben eine bestimmte Struktur, welche für die Generierung von realistisch aussehenden Inseln sehr passend ist. Mit Perlin-Noise sieht zusätzlich die Welt für den Spieler zufällig aus, kann aber ohne Probleme genau gleich rekreiert werden.

Die Spielwelten bestehen aus zwei Ebenen, welche aus Blöcken bestehen. Die erste Ebene ist der Boden, auf dem der Spieler sich bewegt. Die zweite Ebene ist über der ersten Ebene. Diese ist auf der gleichen Höhe wie der Spieler. Zusätzlich ist die Welt in Stücke unterteilt, sogenannte „Chunks“. Diese Chunks sind 10x10 Blöcke gross und formen ein Raster. Sowie jeder Block eine Koordinate hat, hat auch jeder Chunk eine Koordinate. Die Chunks sind wiederum in Regionen unterteilt. Diese sind 15x15 Chunks gross und bilden, wie die Chunks, ein Raster.

3.4.1 Generierung der Chunks

Wie unter Kapitel 3.4 erwähnt, werden die Welten mithilfe von Perlin-Noise generiert. Dabei werden die Welten in Chunks geschaffen. Ein Chunk wird erst erzeugt, wenn ein Spieler in der Nähe von diesem ist. Die Erstellung der Chunks wird dabei vom Computer des Spielers übernommen. Zuerst war geplant, dass die Erstellung vom Server übernommen wird, jedoch fiel auf, dass die Generierung der Welten für den Server sehr leistungsaufwendig sein könnte, wenn mehrere Spiele mit je mehreren Spielern gleichzeitig laufen würden. Deswegen wurde entschieden, dass der Spieler, der den Chunk geladen haben möchte, die Erstellung übernimmt.

Die Generierung der Chunks verläuft über einen Web Worker. Der Code eines Web Workers läuft getrennt vom Hauptthread, womit der Code für das eigentliche Spiel weiter ausgeführt werden kann, während im Hintergrund die Chunks berechnet werden. Sobald ein Chunk generiert werden muss, wird dem Web Worker eine Nachricht mit den Koordinaten des Chunks geschickt. Mithilfe dieser Daten berechnet er anschliessend mittels des Perlin-Noise, wie der Chunk aussieht. Am Anfang wurde die in p5.js eingebaute Perlin-Noise

Funktion gebraucht. Schnell fiel aber auf, dass diese für den Zweck der Weltgenerierung nicht gebräuchlich ist, da die Karte an den Koordinatenachsen gespiegelt ist und somit die genau gleiche Struktur vier Mal gespiegelt generiert wurde. Aus diesem Grund wurde schliesslich die noise.js Bibliothek von Joseph Gentle benutzt.[12]

Bei der Generierung des Chunks wird jeder Block des Chunks durchgegangen. Für jeden Block holt es den Wert des Perlin-Noise an der entsprechenden Koordinate. Der Wert des Perlin-Noise entspricht dabei immer einem Wert zwischen -1 und 1. Für den Block der ersten Ebene wird dann überprüft, ob die Zahl höher oder tiefer als -0.15 ist. Falls der Wert tiefer oder gleich gross ist, wird an der Stelle ein Block platziert, ansonsten ist dort kein Block. Anschliessend wird der Block der zweiten Ebene bestimmt. Auf der zweiten Ebene können verschieden Blöcke generiert werden. Jeder dieser Blöcke hat ein oder mehrere Wertebereiche. Falls der Wert des Perlin-Noise in einem dieser Bereiche ist, wird dann der entsprechende Block auf der zweiten Ebene platziert. Ansonsten bleibt es leer (vgl. Abb. 8).

Sobald der Web Worker durch alle Blöcke des Chunks gegangen ist, sendet es dem Hauptthread die berechneten Blöcke, welche dann gespeichert und dem Server zugeschickt werden, der diese dann auch speichert.

```
function calculateChunk(x, y, seed){
  noise.seed(seed)
  let blocks1 = []
  let blocks2 = []
  //Go through every y and x and apply block based on noise Map
  for(let i = 0; i < chunkSize; i++){
    for(let j = 0; j < chunkSize; j++){
      const noiseLevel = noise.perlin2((chunkSize*x+i)*noiseSmooth, ((chunkSize*y+j)*noiseSmooth))
      blocks1[j*chunkSize+i] = noiseLevel > -0.15 ? 0 : 1; //Determine if grass or air

      let blockPlaced = false //Says if a block has been placed
      for(const block in generationInfo){ //Go through all possible blocks
        const blockspawns = generationInfo[block]
        for(let k = 0; k < blockspawns.length; k+=2){ //Check if noiselevel is within noise range of block
          if(noiseLevel > blockspawns[k] && noiseLevel < blockspawns[k+1]){
            blocks2[j*chunkSize+i] = parseInt(block)
            blockPlaced = true
            break
          }
        }
        if(blockPlaced) break //Stop loop if block has been placed
      }
      if(!blockPlaced) blocks2[j*chunkSize+i] = 0 //add empty block if none has been added
    }
  }
  postMessage({x: x, y:y, blocks1: blocks1, blocks2: blocks2}) //pass data to map.js
}
```

Abbildung 8: Code für die Generierung eines Chunks

[12] Github. noisejs. Verfügbar unter https://github.com/josephg/noisejs [Stand: 18.10.2022]

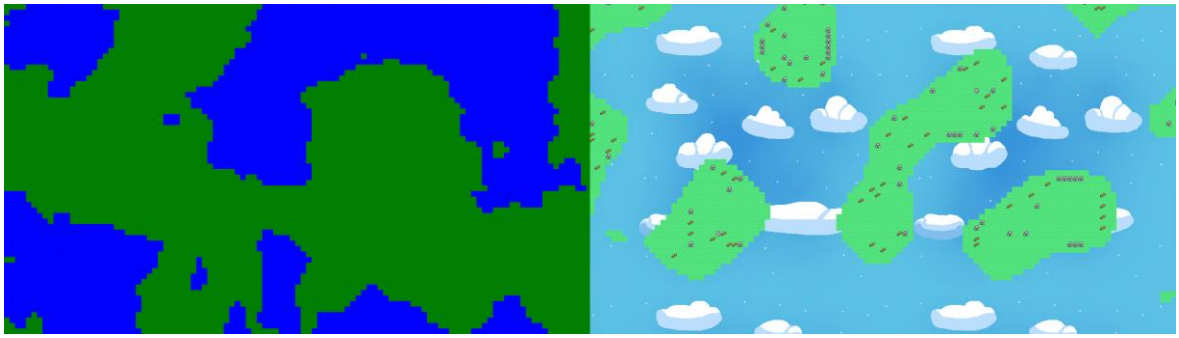

Abbildung 9: Vergleich, links: erste Version der Weltgenerierung, rechts: finale Version der Weltgenerierung

3.4.2 Seeds

Mit der unter Kapitel 3.4.1 erwähnten noise.js Bibliothek wird es einem ermöglicht, 65‘536 verschiedene Perlin-Noise Karten zu benutzen. Dies bedeutet, dass es im Spiel 65‘536 verschiedene Welten geben kann. Um eine bestimmte Karte zu erhalten, muss der „Seed“ dieser Karte verwendet werden. Der Seed ist dabei eine natürliche Zahl zwischen 0 und 65‘536. Die Möglichkeit, einen bestimmten Seed für eine Spielwelt zu verwenden, wird dem Spieler mithilfe eines Input-Felds gegeben. Dabei ist der Spieler nicht beschränkt auf eine Zahl zwischen 0 und 65‘536, sondern kann einen beliebigen String eingeben. Falls der String eine Zahl ist, wird die Zahl, mithilfe einer Modulo-Operation, zu einer Zahl zwischen 0 und 65‘536 umgerechnet. Falls es sich beim Input nicht um eine Zahl handelt, wird der String mithilfe eines Algorithmus in eine Zahl zwischen 0 und 65‘536 umgerechnet (vgl. Abb. 10). Das heisst, dass jeder String eine bestimmte Welt generiert, aber mehrere Strings die gleiche Welt ergeben können.

Dieses System wurde implementiert, damit Spieler auf der gleichen Welt mehrmals spielen können und bei der Auswahl eines Seeds nicht auf eine Zahl zwischen 0 und 65‘536 beschränkt sind, sondern einen beliebigen String benutzen können.

```
//Convert seed into something usable
const value = worldSeedInput.value()
let seed = 1
if(value!=""){
  if(isNaN(value)){ //Seed is string, convert into int with algorithm using ascii code
    for(let i = 0; i < value.length; i++){
      seed *= (value.charCodeAt(i)/10)
      if(seed>1000000)seed/=100000
    }
    seed = int(seed)%65536
  }else{
    seed = int(value)%65536 //Seed is number, convert into something between 0, 65536
  }
}else seed = int(random(0, 65536)) //No seed Specified, choose random seed
```

Abbildung 10: Code für das Umkonvertieren eines Seeds

3.4.3 Weltspeicherung

Da die Blöcke der Welt von Spieler manipuliert werden können, kann der Chunk nicht jedes Mal neu generiert werden, sondern muss abgespeichert werden. Dabei kann der Server diese nicht einfach im RAM speichern, da dadurch das Risiko bestehen würde, dass der RAM gefüllt werden könnte oder die Daten, im Falle eines Serverabsturzes, verloren gehen würden. Aus diesem Grund müssen die Chunk-Daten, wenn sie nicht mehr gebraucht werden, auf der Festplatte abgespeichert werden. Dabei müssen diese Daten mit einer sinnvollen Struktur abgespeichert werden, damit diese schnell abgerufen werden können. Infolgedessen gibt es die unter Kapitel 3.4 erwähnten Regionen.

Für jede Region, welche mindestens einmal geladen wurde, gibt es zwei Dateien. Die erste Datei ist eine Binärdatei, in der die Blöcke der einzelnen Chunks aufgeschrieben sind. Im Spiel hat jeder Block eine bestimmte Zahl, mit welcher der Block im Spielcode dargestellt wird. In der Binärdatei werden die Blöcke des Chunks gespeichert, indem die Zahlen der Blöcke als 8-bit Binärzahlen abgespeichert werden (vgl. Abb. 11). Der Grund dafür ist, dass so mit einem Byte 256 verschiedene Blöcke abgespeichert werden können und auf diese Weise weniger Speicherplatz aufgenommen wird, als wenn die Zahlen mit der standardmässigen UTF-8 Kodierung abgespeichert werden würden. An welcher Stelle ein einzelner Chunk in der Binärdatei abgespeichert ist, steht in der zweiten Datei, welche eine JSON-Datei ist. Zusätzlich beinhaltet diese Datei Informationen über Gegenstände und computergesteuerten Feinde, welche sich in einem Chunk befinden. Sobald ein neuer Chunk generiert wurde und abgespeichert werden muss, werden die Informationen zu den Blöcken am Ende der Binärdatei hinzugefügt. Anschliessend wird die Stelle, an der es in der Binärdatei gespeichert wurde, in der JSON-Datei gespeichert, um es später wieder aufrufen zu können. Mit dieser Methode können die Blöcke eines Chunks ausgelesen werden, ohne die Blöcke anderer Chunks auslesen zu müssen. Da der Server weiss, an welcher Stelle die Blöcke eines Chunks abgespeichert sind, kann es diesen Teil der Datei lesen, während der Rest der Datei ignoriert wird.

Die zweite Datei wird als JSON-Datei abgespeichert, da eine JSON-Datei ohne wirklichen Aufwand in ein JavaScript-Objekt umgewandelt werden kann und die Informationen in der Datei somit einfach aufgerufen werden können. Es ist zwar vom Speicherplatz her nicht das effektivste, jedoch muss immer ein Kompromis zwischen Effizienz und Nutzbarkeit eines Systems gefunden werden, wobei hier mehr auf die Nutzbarkeit Acht gegeben wurde.

Zusätzliche Informationen zur Welt, wie zum Beispiel den Namen, den Seed oder die Uhrzeit, werden in einer separaten JSON-Datei abgespeichert, welche beim Öffnen der Welt aufgerufen wird. Alle Dateien einer Welt werden im gleichen Verzeichnis abgespeichert. Die verschiedenen Welten werden dabei mit einer bestimmten Zahl identifiziert, welche einer Welt bei der Erstellung zugewiesen wird.

```
let buffer = new Buffer.alloc(200) //Create buffer where the written content will be saved, 200 bytes because 200 blocks
//Save values from Array in buffer as 8bit binary numbers
for(let i = 0; i < 100; i++) buffer[i] = "0b" + to8Binary(chunk.blocks1[i])
for(let i = 100; i < 200; i++) buffer[i] = "0b" + to8Binary(chunk.blocks2[i-100])

const path = this.path + "chunkData/blocks/"+chunk.lRegion+"-"+chunk.sRegionX+"_"+chunk.sRegionY
if(!fs.existsSync(path)) fs.closeSync(fs.openSync(path, 'w')); //create file if it doesnt exist

openFile(path, data => { //Open File
  //Write data to file
  fs.write(data, buffer, 0, 200, chunk.index*200, (err) =>{
    if(err){console.log("error trying to write bytes to save Chunk Blocks"); return;}
    closeFile(data) //Close the file
  });
})
```

Abbildung 11: Code für das Speichern der Blöcke eines Chunks

3.4.4 Laden der Chunk-Daten

Der Spieler hat zu jeder Zeit eine bestimmte Anzahl Chunks geladen. Sobald ein neuer Chunk betreten wird, werden neue Chunks geladen und jene, welche nicht mehr benötigt werden, gelöscht. Auf diese Weise sieht es für den Spieler aus, als wäre er in einer unendlichen Welt, ohne tatsächlich eine unendliche Welt geladen zu haben.

Sobald neue Chunks geladen werden, sendet das Spiel des Spielers eine Anfrage nach den Daten der Chunks an den Server. Der Server kontrolliert anschliessend für jeden Chunk, ob die Daten zu dem Chunk existieren. Falls der Chunk schon geladen ist, da ein anderer Spieler den Chunk geladen hat, sendet der Server die Daten an den Spieler zurück. Ansonsten muss der Server kontrollieren, ob die Region des Chunks geladen ist. Falls diese nicht geladen ist, muss er diese laden, indem die unter Kapitel 3.4.3 erwähnte JSON-Datei in ein JavaScript-Objekt umgewandelt wird. Falls die JSON-Datei nicht existiert, heisst das, dass die Region noch nie geladen wurde und neu kreiert werden muss. Nun wird in den Informationen zu der Region nachgeschaut, an welcher Stelle der Chunk in der Binärdatei abgespeichert ist. Entsprechend wird diese Stelle in der Datei abgelesen, in brauchbare Daten umgewandelt und an den Spieler geschickt. Falls keine Informationen in der Region zum Chunk existieren, bedeutet dies, dass der Chunk noch nie geladen wurde. In diesem Fall schickt der Server nicht die Daten des Chunks zurück, sondern sagt dem Spieler, dass es den Chunk selbst generieren muss.

Das System der Regionen wird benutzt, da der Server somit nur die Informationen der Chunks geladen hat, welche am wahrscheinlichsten abgefragt werden. Zudem sollte nicht für jeden Chunk eine einzelne Datei gemacht werden, da mit jeder neuen Datei in einem Verzeichnis das Lesen dieser Dateien langsamer wird. Aus diesem Grund sollten die

Chunks irgendwie zusammengruppiert werden. Das System mit den Regionen schien der beste Kompromiss zwischen Effizienz und Nutzbarkeit.

3.4.5 Weltmanipulation

Dem Spieler wird die Möglichkeit gegeben die Spielwelt, in der er sich befindet, zu manipulieren. Das heisst, dass der Spieler Blöcke abbauen und neue platzieren kann. Dies muss wiederum mit dem Server kommuniziert werden. Sobald ein Spieler ein Block in der Spielwelt verändert, wird der veränderte Block an den Server gesendet. Der Server speichert anschliessend den neuen Block ab und leitet die Daten an die anderen Spieler weiter. Dabei wird es nicht an jeden Spieler gesendet, sondern nur an die, welche den Chunk geladen haben. Diese Spieler verändern den Block nachfolgend auch in ihrem Spiel, damit auch ihre Spielwelt korrekt angezeigt wird.

3.4.6 Zeichnen der Welt

Nun wurde angeschaut, wie die Weltdaten generiert werden und wie diese herumgesendet, beziehungsweise gespeichert werden. Jetzt müssen diese Daten noch korrekt dargestellt werden, damit die Spieler die Welt auch sehen können. Dabei ist allgemein das Zeichnen von Elementen auf dem Canvas vergleichsweise zu anderen Operationen sehr ressourcenaufwendig. Da bei jedem Frame eine gesamte Spielwelt gezeichnet werden muss, ist es wichtig, dass dies effizient gemacht wird, weil ansonsten das Spiel auf schwächeren Computern nicht mehr flüssig laufen würde. Dies war der Grund, warum das Zeichnen der Welt eines der Probleme war, mit der ich, vor allem anfangs, am meisten Zeit verbringen musste.

Anfangs wurde die Welt gezeichnet, indem jeder Block einzeln auf das Canvas gezeichnet wurde. Es war aber schnell bemerkbar, dass die Leistung des Spiels stark darunter litt, weswegen eine andere Methode gefunden werden musste. Die zweite Methode involvierte das p5.Renderer Objekt. Jedes Mal, wenn ein neuer Chunk geladen wurde, wurde mithilfe eines p5.Renderer Objekt ein Bild des Chunks gezeichnet. Danach wurden die Bilder der Chunks zu einem Bild kombiniert, welches dann bei jedem Frame gezeichnet wurde. Mit dieser Methode lief das Spiel viel flüssiger. Nur bestand nun das Problem, dass beim Laden neuer Chunks es immer einen kurzen Sturz in der Leistung des Spiels gab, da alle Bilder in einem Frame gezeichnet werden mussten. Dies versuchte ich zu beheben, indem kein Bild der ganzen Welt gezeichnet wurde, sondern die Bilder der einzelnen Chunks bei jedem Frame gezeichnet wurden, was aber nur minimal half. Zusätzlich gab es das Problem, dass

für jedes p5.Renderer Objekt ein HTML-Canvas kreiert wurde, was ich gerne vermieden hätte.

```
async function drawChunkOnCanvas(blocks1, blocks2, x, y){
  cCtx.clearRect(0, 0, cCanvas.width, cCanvas.height)
  for(let i = 0; i < blocks1.length; i++){
    let block; //Info of block
    let level; //height of block
    const row = Math.floor(i/chunkSize) //Find out y of block inside Chunk

    if(blocks2[i] == 0){
      if(blocks1[i] != 0) {block = blockInfo[blocks1[i]]; level = 1} //No Block on second
      else level = 0//No block on eiter
    }else { //Block on second
      block = blockInfo[blocks2[i]];
      level = 2
    }

    //Add to Canvas
    if(level == 1) cCtx.drawImage(block.image, (i-row*chunkSize)*blockRes, row*blockRes, blockRes, blockRes) //Draw block on level 1
    else if(level == 2){ //block on level 2
      if(block.translucent){ //Draw block below if block on level 2 is translucent
        const blockBelow = blockInfo[blocks1[i]]
        if(blocks1[i] != 0) cCtx.drawImage(blockBelow.image, (i-row*chunkSize)*blockRes, row*blockRes, blockRes, blockRes)
      }

      cCtx.drawImage(block.image, (i-row*chunkSize)*blockRes, row*blockRes, blockRes, blockRes) //Draw block on level 2
      if(!block.noStroke) cCtx.drawImage(strokeImg, (i-row*chunkSize)*blockRes, row*blockRes, blockRes, blockRes)//Add stroke if needed
    }
  }
  const bitmap = await createImageBitmap(cCanvas)
  postMessage({x:x, y:y, image: bitmap})
  bitmap.close()
}
```

Abbildung 12: Code für das Zeichnen eines Bilds von einem Chunk

Schliesslich wurde ein Web Worker benutzt. Mit einem Web Worker wird es ermöglicht, im Hintergrund ein Bild eines Chunks zu zeichnen, während der Code für das eigentliche Spiel weiter ausgeführt wird. Somit wird das Problem der kurzzeitigen Reduktion der Leistung beim Laden neuer Chunks gelöst. Sobald ein neuer Chunk geladen wird oder der Chunk sich verändert, wird dem Web Worker eine Nachricht mit den Blöcken des Chunks gesendet. Dieser zeichnet dann ein Bild des Chunks mithilfe eines OffscreenCanvas (vgl. Abb. 12). Dieses Bild schickt der Web Worker anschliessend in der Form einer Bitmap zurück an den Hauptthread. Dieser wandelt schliesslich die erhaltene Bitmap in ein p5.Image Objekt um, welche auf dem Canvas gezeichnet werden kann. In der ersten Version des Web Workers wurde ein Bild der gesamten Welt gezeichnet, welche dann an den Hauptthread zurückgeschickt wurde. Das Problem war aber, dass die Konversion in ein p5.Image Objekt zu lange dauerte und wieder zu einer kurzen Reduktion der Leistung führte, weswegen letztlich nur Bilder der einzelnen Chunks vom Web Worker gezeichnet werden.

Mit dem implementierten System wird bei jedem Frame des Spiels jeder Chunk einzeln gezeichnet. Die einzelnen Bilder der Chunks werden nur gezeichnet, wenn der Chunk zuerst geladen wird oder ein Block sich im entsprechenden Chunk verändert.

3.5 Accounts

Der Server benötigt einen Weg, verschiedene Personen, welche das Spiel spielen, voneinander zu entscheiden. Falls ein Spieler zurückkehrt, sollte er in der Lage sein in seiner Welt weiterspielen zu können. Aber dafür wird ein Weg benötigt, die Welten und Daten einer Person zuordnen zu können. Ausserdem, weil es ein Multiplayer-Spiel ist, muss es einen Weg haben, bestimmten Personen die Möglichkeit zu geben, der eigenen Spielwelt beizutreten. Diese Probleme wurden mithilfe eines Account-Systems gelöst.

Im Hauptmenü des Spiels wird einem die Möglichkeit gegeben, einen Account für das Spiel zu erstellen. Unter diesem Account werden sämtliche Daten, wie Welten und Spielstände, gespeichert. Das gibt zusätzlich den Vorteil, dass auf dem gleichen Spielstand auf verschiedenen Computern gespielt werden kann. Auch gibt es die Möglichkeit, andere Personen zu seiner Freundesliste hinzuzufügen. Spieler, welche in der eigenen Freundesliste sind, erhalten somit die Erlaubnis der eigenen Spielwelt beizutreten, falls von einem selbst auf dieser Welt gespielt wird. Die Freundschaft muss jedoch zuerst von beiden Parteien durch eine Freundesanfrage akzeptiert werden.

Ein Account ist jedoch optional. Das Spiel ist auch ohne Account spielbar. Jedoch ist die Funktionalität des Spiels ohne Account eingeschränkt, da sämtlicher Fortschritt nicht gespeichert wird und die Möglichkeit, mit einer anderen Person zusammen zu spielen, nicht vorhanden ist.

Für das Account-System hätte es mehrere Möglichkeiten gehabt. Es hätte ein selbstgeschriebenes System oder ein schon vorhandenes System, wie zum Beispiel „Google Sign-In“, benutzt werden können. Schliesslich wurde aber ein selbst geschriebenes System genutzt. Somit wird mehr Kontrolle und Überblick über das Ganze geschaffen. Zudem braucht es für ein Projekt dieser Grösse kein grosses und hoch entwickeltes System, weswegen das Programmieren eines eigenen Systems nicht viel Zeit in Anspruch nahm.

3.5.1 Funktionsweise des Account-Systems

Wie schon unter Kapitel 3.5 erwähnt, wird einem auf dem Hauptmenü des Spiels die Möglichkeit gegeben, sich zu registrieren oder anzumelden. Für das Registrieren sind dabei nur zwei Daten notwendig: der Benutzername und das Passwort. Mehr Daten braucht es nicht, da ich ein System haben wollte, welches so wenig Daten wie möglich benötigt. Dies ist der Fall, weil viele Menschen nicht gerne Informationen, wie zum Beispiel ihre E-Mail, auf Seiten teilen. Sobald das Formular für das Registrieren ausgefüllt wird, wird das Passwort zuerst mit der SHA-256 Hashfunktion in einen Hash umgewandelt. Dies wird gemacht, damit der Server das tatsächliche Passwort nie erhält und somit das Passwort der Person geschützt bleibt. Der Hash zusammen mit dem Benutzernamen wird anschliessend an den

Server gesendet, der daraus einen Account kreiert. Beim Erstellen eines Accounts wird immer eine zufällige natürliche Zahl als ID für den Account generiert. Mit der ID wird bestätigt, dass eine Person tatsächlich die Person ist, für die es sich ausgibt. Die ID wird dabei vom Server, zusammen mit dem Benutzernamen, beim Client als Cookie abgespeichert. Jedes Mal, wenn der Server eine Anfrage erhält, für welche eine Anmeldung bestehen muss, wird die Identität überprüft, indem die ID, welche in den Cookies gespeichert ist, mit der ID verglichen wird, welche der Server bei sich abgespeichert hat (vgl. Abb. 13). Falls die zwei IDs übereinstimmen, bedeutet das, dass die Person tatsächlich die Person ist, für die es sich ausgibt.

Um sich anzumelden, muss der Benutzername sowie das Passwort eingegeben werden. Das Passwort wird danach wieder in einen Hash umgewandelt und die Daten werden an den Server gesendet. Dieser schaut dann, ob der Hash des Passworts mit dem Hash übereinstimmt, der auf dem Server abgespeichert ist. Falls die zwei Hashes übereinstimmen, bedeutet das, dass das Passwort stimmt. In diesem Fall wird das Ganze bestätigt, indem der Server die ID und den Benutzernamen des Accounts in den Cookies des Clients abspeichert, wodurch der Client angemeldet ist.

Die Daten, wie zum Beispiel die ID und den Hash des Passworts, werden dabei ganz simpel als JSON-Datei abgespeichert. Somit kann der Server einfach auf die Daten des Accounts zugreifen und diese verändern.

```
//Check if user is logged in by validating id
function checkLogin(user, id){
  if(!Number.isInteger(parseInt(id)) || typeof user != "string") return false //Check values
  if(removeSpecialChars(user) != user) return false //Remove forbidden Chars

  const userObject = getUserData(user) //Get object from JSON file of user
  if(userObject == false) return false

  if(id == userObject.id) return true
  else return false
}
```

Abbildung 13: Code für die Verifizierung der Identität

3.5.2 Freundessystem

Das Freundessystem erlaubt es einem, eine andere Person in seine Freundesliste zu tun, um mit ihnen zusammen spielen zu können. Dies funktioniert, indem eine Person einer anderen eine Freundesanfrage schickt, welche dann von der anderen Person angenommen werden muss. Sobald sie angenommen wird, haben beide Personen sich gegenseitig in der Freundesliste. Dabei ist die Freundesliste lediglich ein Array von Benutzernamen, welche in der JSON-Datei des Accounts abgespeichert wird. Sobald eine neue Person zu der Freundesliste hinzugefügt werden muss, wird der Benutzername dieser Person zum Array hinzugefügt wird. Genau wie die Freundesliste, sind Freundesanfragen auch nur Namen in einem Array. Wenn eine Freundesanfrage versendet wird, wird der Benutzername

des Senders zum Array des Empfängers hinzugefügt. Nachdem die Freundesanfrage akzeptiert oder abgelehnt wurde, wird der Name wieder aus dem Array entfernt.

3.6 Serversicherheit und -stabilität

Ein wichtiger Bestandteil eines Servers ist dessen Sicherheit und Stabilität. Es soll verhindert werden, dass Personen Zugriff auf Daten erhalten, welche ihnen nicht zugänglich sein sollten. Ausserdem muss versichert werden, dass der Server so wenigen Abstürzen und Fehlermeldungen wie möglich begegnet, damit die Spieler das Spiel ungestört spielen können.

Während der Entwicklung des Spiels fiel auf, dass der Server oft abstürzte, da er bei Anfragen von Clients ungültige Daten bekam. Zudem konnte manuell der Server zum Abstürzen gebracht werden, indem ihm bewusst falsche Daten geschickt wurden, was natürlich nicht möglich sein sollte. Zusätzlich war es möglich, mit manuellen Anfragen Daten zu erhalten, auf welche der Zugriff nicht bestehen sollte. Um dieses Problem so gut wie möglich zu bewältigen, überprüft der Server bei jeder Anfrage zuerst, ob die Daten den Datentypen entsprechen, welche er erwartet und ob der Client Berechtigung auf diese Anfrage hat. Falls dies nicht der Fall ist, wird die Anfrage abgebrochen. Zwar ist dies nicht vollkommen sicher, jedoch sind mit dieser Massnahme viele der Sicherheitslücken gelöst.

Eine weitere Sicherheitslücke, welche auffiel, sind von Clients gesendete Strings. Wenn ein Client einen String sendet, mit dem nachher vom Server auf ein Verzeichnis zugegriffen wird, kann der Client den Server auf andere Verzeichnisse mithilfe von speziellen Characters führen, wie zum Beispiel dem „/“ Character oder der Characterfolge „..“. Somit kann der Client Daten von Verzeichnissen erhalten, auf welche er keinen Zugriff haben sollte. Um dies zu lösen, werden bei jedem einkommenden String alle speziellen Characters unter Zuhilfenahme von Regex vom String entfernt (vgl. Abb. 14). Somit beinhalten alle einkommenden Strings keine speziellen Characters, mit welchen der Server auf andere Verzeichnisse geführt werden kann.

```
function removeSpecialChars(string){
  return string.replace(/[^a-zA-Z0-9 _-]/g, "")
}
```

Abbildung 14: Code für das Entfernen von speziellen Characters von einem String

4. Fazit und Rückblick

Dieses Projekt war nicht das erste Videospiel, welches ich mit JavaScript und der p5.js Bibliothek programmiert habe. Doch ich merkte schnell, wie viel mehr Arbeit in einem bisschen grösserem Projekt wie diesem liegt, da bis jetzt alle Projekte, welche ich vor dem geschrieben habe, nur kleine Spiele waren. Zudem wurde der Entwicklungsprozess durch den Server und der Multiplayer-Funktionalität um einiges komplexer. Wenn ich ein Feature ins Spiel einbaute, reichte es nicht, dass es beim Spieler selbst funktionierte. Jedes Mal musste noch die Synchronisation mit dem Server und den anderen Spielern berücksichtigt werden, was oftmals viel länger dauerte als die Entwicklung des Features selbst. Deswegen sah die Entwicklung dieses Spiels sehr anders aus als die meiner anderen Projekte, an was ich mich zuerst gewöhnen musste. Was ich während der Entwicklung stark bemerkte, war, dass es manchmal nicht schwierig ist ein Feature so zu schreiben, dass es funktioniert, sondern das Feature so zu schreiben, dass es effizient funktioniert. Dies war mir zwar schon vor der Entwicklung dieses Projekts bewusst, jedoch hatte ich es noch nie in diesem Ausmass erlebt. Vor allem war dies der Fall bei dem Speichern und Zeichnen der Spielwelten. Was ich durch das Lösen dieser schwierigeren Probleme lernte, war, dass falls keine Lösung für ein Problem gefunden wird, durch das Probieren und einfaches Anfangen ein Ansatz gefunden werden kann. Zwar ist der erste Versuch nicht immer gut oder funktioniert vielleicht gar nicht, jedoch kann darauf aufgebaut und verbessert werden, bis es funktioniert und das Resultat zufriedenstellend ist. Während der Entwicklung hatte ich das Problem, dass ich für längere Zeit nicht am Spiel arbeitete, da ich vor einem grossen Problem stand und ich nicht wusste, wie ich dieses hätte lösen sollen. Dies führte zu einer Reduktion meiner Motivation, wodurch ich nicht mehr am Spiel arbeitete und so das Problem vorzeitig nicht löste. Doch als ich es dann einfach probierte, kam ich dann langsam, aber sicher, auf eine passende Lösung. Am meisten Probleme hatte ich mit dem Speichern und Laden der Welten. Es war das erste Mal, dass ich in einem Projekt Datenmengen dieser Grösse abspeichern musste. Dabei war das grösste Problem einen effizienten Weg dafür zu finden.

Auch wenn es mir schon vorher bewusst war, merkte ich mit diesem Projekt wieder, wie viel Arbeit in einem Videospiel steckt. Ich arbeitete etwa acht Monate an diesem Projekt und das Resultat würden viele Personen als simples Spiel mit nicht viel Inhalt bezeichnen. Dies stimmt aber auch. Als ich es selbst mit Kollegen spielte, merkte ich schnell, dass es nach etwa 10 Minuten langweilig wurde und es nichts mehr zu tun gab. Trotzdem bin ich froh, dieses Spiel erstellt zu haben, weil ich vieles dabei gelernt habe, was ich später sicher gebrauchen kann. Zudem war es nicht mein Ziel, ein Spiel zu programmieren, welches von vielen Menschen gespielt wird, sondern einmal ein Spiel dieser Grösse zu programmieren, was ich erfolgreich erreicht habe. Anhand eines solchen Projekts ist schnell sichtbar, warum

normalerweise für ein Videospiel eine Spiel-Engine und ein Team benötigt wird. Die Spielentwicklung würde ansonsten zu lange dauern und sich nicht rentieren.

Ich werde sicher zu einem späteren Zeitpunkt noch einige kleine Features zu dem Spiel hinzufügen und immer wieder Fehler beheben, welche gefunden werden. Jedoch wird das Spiel im Groben so bleiben, wie es jetzt ist. Es ist schwierig und kompliziert, neue Features hinzuzufügen, welche ohne Probleme funktionieren und keine anderen Features kaputt machen. Für mich lohnt es sich nicht, diese Zeit dafür zu investieren, da ich keinen wirklichen Nutzen daraus habe. Da ich aber mein Ziel mit diesem Projekt erreicht habe, bin ich froh darüber, dieses Projekt geschrieben zu haben und mit der Welt teilen zu können.

Quellen- und Literaturverzeichnis

- Axios. Getting Started. Verfügbar unter https://axios-http.com/docs/intro [Stand: 18.10.2022]
- Codeacademy. Introduction to Creative Coding. Verfügbar unter https://www.codecademy.com/learn/learn-p5js/modules/p5js-introduction-to-creative-coding/cheatsheet [Stand: 18.10.2022]
- Duden. Server. Verfügbar unter https://www.duden.de/rechtschreibung/Server [Stand: 18.10.2022]
- Github. FloatingIsland. Verfügbar unter https://github.com/ShnyarM/FloatingIsland [Stand: 02.11.2022]
- Github. noisejs. Verfügbar unter https://github.com/josephg/noisejs [Stand: 18.10.2022]
- Minecraft Wiki. Weltgenerierung. Verfügbar unter https://minecraft.fandom.com/de/wiki/Weltgenerierung [Stand: 18.10.2022]
- Roxl, Rhett. (2021). What is Peer-to-Peer Gaming, and How Does it Work? Verfügbar unter https://vgkami.com/what-is-peer-to-peer-gaming-and-how-does-it-work/ [Stand: 18.10.2022]
- Socket.IO. Introduction. Verfügbar unter https://socket.io/docs/v4/ [Stand: 18.10.2022]
- Wikipedia. Node.js. Verfügbar unter https://de.wikipedia.org/wiki/Node.js [Stand: 18.10.2022]
- Wikipedia. Prozedurale Synthese. Verfügbar unter https://de.wikipedia.org/wiki/Prozedurale_Synthese [Stand: 18.10.2022]
- XOVI. Was bedeutet Peer-to-Peer? Verfügbar unter https://www.xovi.de/was-bedeutet-peer-to-peer/ [Stand: 18.10.2022]

Anhang

Website von Floating Island: https://floatingisland.ch [Stand: 03.11.2022]

Code von Floating Island: https://github.com/ShnyarM/FloatingIsland [Stand: 02.11.2022]

Abbildungsverzeichnis