Projektarbeit

Erstellung des Videospiels „Snake“ als mehrspielerfähige Anwendung mit der Videospiel-Engine Godot

Vorgelegt am: 25.11.2024

Von: **Maurice Roscher** 4004838

Lukas Grünitz 4004987

Studiengang: Technische Informatik

Seminargruppe: 4TI22-1

Gutachter: Professor Dr. Mathias Sporer
(Staatliche Studienakademie Glauchau)

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
NAT	Network Adress Table
IP-Adresse	Internetprotokoll Adresse

1 Einleitung

1.1 Vorwort

In den letzten Jahrzenten hat die Videospielindustrie einen enormen Aufschwung erlebt. Im Jahr 2023 erzielte sie einen weltweiten Umsatz von ungefähr 184 Milliarden US-Dollar bei einer Spielerbasis von 3,38 Milliarden Menschen[1]. Dies zeigt, dass Videospiele kein Nischenhobby mehr sind, sondern sich längst zu einem Massenphänomen entwickelt haben. Dabei haben sich im Laufe der Zeit nicht nur die Genres und Spielemechaniken ausgeweitet, sondern auch die Plattformen, auf denen gespielt wird. Von Konsolen und PCs über mobile Geräte bis hin zu Virtual-Reality-Systemen bietet sich hier ein breites Spektrum.

Zu einem Klassiker der Videospielgeschichte gehört das Spiel *Snake*, bei welchem eine sich gerade oder rechtwinklig bewegende Schlange durch ein Spielfeld gesteuert wird. Das Ziel ist es, zufällig auf dem Spielfeld erscheinende Früchte aufzunehmen und nicht mit dem Spielfeldrand oder dem eigenen Körper zu kollidieren[2]. Letzterer wächst zunehmend mit dem Aufnehmen der Früchte, wodurch das erfolgreiche Manövrieren durch das Spielfeld immer schwieriger wird. Die Geschichte des Spiels geht bis in das Jahr 1976 zurück, wo das Konzept erstmals unter dem Name *Blockage* vom ehemaligen Entwicklerstudio *Gremlin Interactive* entworfen wurde. Große Bekanntheit erlangte Snake jedoch erst mit der Entwicklung des Nokia 6110, welches 1997 vom finnischen Konzern Nokia auf den Markt gebracht wurde. Diese Version wurde von Taneli Armanto programmiert und machte es zu einem der bekanntesten und meistgespieltesten Handyspiele aller Zeiten.[3]

1.2 Zielsetzung

Die Erstellung dieser Arbeit und der Anwendung findet im Rahmen des Moduls *Internettechnologien* eines Studiums an der Staatlichen Studienakademie Glauchau statt. Das Ziel des Moduls ist die Erstellung einer webbasierten Anwendung. Die Entscheidung für das Thema dieser Arbeit fiel dabei auf die Umsetzung des Videospielklassikers *Snake* als mehrspielerfähige Anwendung mit einer WebRTC-Verbindung. Der Entwicklungsprozess beinhaltet dabei drei zentrale Punkte:

[1] MASCELLINO, 2024
[2] online: Wikipedia, 2024 (26.09.2024)
[3] online: The history of Snake: How the Nokia game defined a new era for the mobile industry, 2024 (26.09.2024)

- Grundlegende Spiellogik

Der erste Schritt besteht darin, die grundlegende Spiellogik von Snake zu entwickeln. Dazu gehören der Aufbau des Spielfelds, die Erstellung und Steuerung der Schlange sowie das Einfügen von Früchten und das damit zusammenhängende Wachstum jener. Wichtig ist zudem die Kollisionserkennung mit dem Spielfeldrand oder der Schlange selbst. Um das Spielerlebnis abwechslungsreicher bzw. herausfordernder zu gestalten, soll das Spiel verschiedene Schwierigkeitsstufen beinhalten, welche die Geschwindigkeit der Schlange beeinflussen. Zudem werden mehrere Level mit unterschiedlichen Hindernissen erstellt, um die Herausforderung für den Spieler weiter zu variieren. Geplant sind zwei Spielmodi: Ein kompetitiver Spielmodus, bei welchem zwei Schlangen gegeneinander antreten und ein kooperativer Spielmodus, bei welchem ein Tier von beiden Spielern abwechselnd gesteuert wird.

- Aufbau einer WebRTC-Verbindung

Der zweite Schritt konzentriert sich auf den Aufbau einer WebRTC-Verbindung zwischen zwei Geräten. Diese Verbindung ermöglicht den Datenaustausch zwischen den Spielern und stellt sicher, dass das Spiel auf den Geräten synchronisiert abläuft. Durch den Einsatz von RTC wird der Server entlastet, da die Kommunikation und Synchronisation direkt zwischen den Endgeräten erfolgt. Dies ist wichtig, da der Server in diesem Fall ein gewöhnlicher PC oder Laptop ist und so die notwendige Rechenleistung und Bandbreite minimiert werden kann.

- Anpassung der Spiellogik an die RTC-Verbindung

Im dritten Schritt wird die zuvor entwickelte Spiellogik an die RTC-Verbindung angepasst. Dabei liegt der Fokus auf der Synchronisierung von Client und Server, um das Spiel möglichst latenzfrei zu gestalten. Um mögliche Probleme mit der Portfreigabe zu umgehen, wird auch der manuelle RTC-Aufbau in Betracht gezogen, bei dem eine externe Verbindung genutzt wird, um die RTC-Verbindung zwischen den Spielern herzustellen. Der Schwerpunkt dieser Arbeit liegt dabei auf dem Aufbau einer stabilen Online-Verbindung für das Mehrspieler-Erlebnis, da dies den Bezug zum Modulthema „Internettechnologien“ darstellt.

2 Technische Grundlagen

2.1 Die Videospiel-Engine Godot

Godot ist eine universelle Spiel-Engine für die Erstellung von 2D- und 3D Anwendungen bzw. Videospielen für Desktop, mobile Geräte und den Browser. Ursprünglich wurde Godot im Jahr 2001 intern von einem argentinischen Entwicklerstudio entwickelt und 2014 als Open-Source-Projekt veröffentlicht. Die Engine umfasst verschiedene Werkzeuge, darunter einen Code-Editor, einen Animationseditor, einen Tilemap-Editor und einen Debugger. Godot stützt sich auf das Paradigma der objektorientierten Programmierung und unterstützt mehrere Programmiersprachen. Dazu gehört *GDScript*, eine spezifisch für Godot entwickelte Programmiersprache, deren Syntax an Python angelehnt ist. Alternativ kann auch C# verwendet werden, inklusive der Integration von Visual Studio bzw. Visual Studio Code als Code-Editor. Mit GDExtension Technologie kann auch C oder C++ genutzt werden.[4]

Zu den wichtigsten Konzepten in Godot gehören *Nodes* (Knoten), *Szenen* und *Signale*. *Nodes* sind die kleinsten Bestandteile einer Godot-Anwendung und können verschiedene Aufgaben übernehmen, wie z.B. das Abspielen von Musik (*AudioStreamPlayer2D*), das Steuern von Animationen (*AnimatedSprite2D*) oder das Gestalten der Benutzeroberfläche (*Button, Label, Popup*). Nodes können zudem hierarchisch in einer Baumstruktur angeordnet werden. Dabei fungiert ein Node als Eltern-Knoten, diesem können beliebig viele Kind-Knoten untergeordnet werden.[5] Zur Veranschaulichung zeigt die folgende Abbildung eine beispielhafte Anordnung mehrerer Nodes.

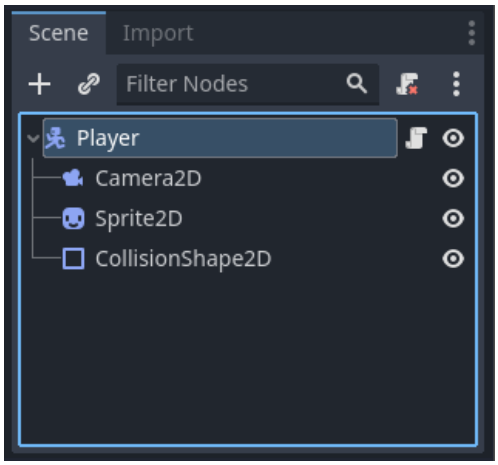


Abbildung 1: Beispielhafte Anordnung von Nodes in Godot[6]

[4] online: Godot Engine documentation, 2024a (30.09.2024)
[5] Vgl. online: Godot Engine documentation, 2024b (30.09.2024)
[6] online: Godot Engine documentation, 2024b (30.09.2024)

In der Abbildung ist ein *CharacterBody2D*-Node mit dem Namen „Player“ der Eltern-Knoten. Ihm untergeordnet sind drei weitere Nodes, welche jeweils verschiedene Funktionen erfüllen: Ein *Camera2D*-Node, ein *Sprite2D*-Node und ein *CollisionShape2D*-Node. Mit dem Eltern-Knoten ist zudem ein Skript verknüpft, welches sein Verhalten steuert. Zusammen bilden diese vier Nodes eine *Szene*.

Eine Szene ist eine Sammlung bzw. Anordnung von Nodes, welche als eine Einheit gespeichert wird und als eigener Node-Typ in anderen Szenen verwendet werden kann. Dies sorgt für eine modulare Struktur beim Entwickeln von Anwendungen mit Godot. So kann beispielsweise die „Player“-Szene aus Abbildung 1 einer anderen Szene hinzugefügt werden und wird dann als eigener Node behandelt. Abbildung 2 zeigt ein Beispiel für diesen Fall.

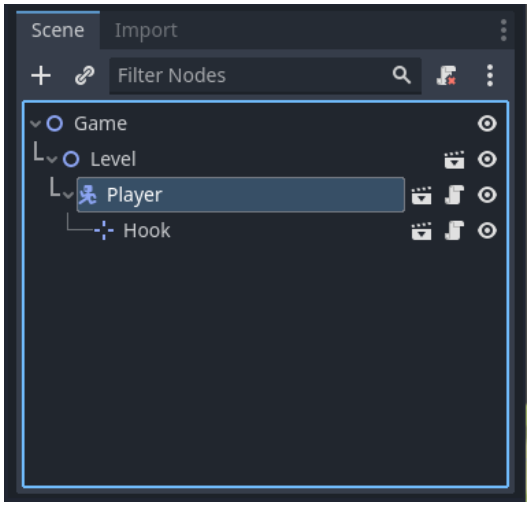


Abbildung 2: Instanziierung der "Player"-Szene in einer anderen Szene[7]

Das dritte Grundkonzept in Godot sind *Signale.* Signale werden von Nodes bei bestimmten Ereignissen ausgelöst und können mit Methoden verbunden werden, um bestimmte Funktionalitäten zu schaffen. So besitzen beispielsweise Buttons in Godot das Signal *pressed*, welches ausgelöst wird, wenn sie angeklickt werden. Dieses Signal kann mit einer Methode aus einer Skript-Datei verknüpft werden, sodass diese aufgerufen wird, wenn das Ereignis ausgelöst wird. Die erwünschte Funktionalität ist dem Programmierer überlassen. So kann als Reaktion auf das Signal eine Szene gewechselt oder ein Spiel gestartet werden. Auch können benutzerdefinierte Signale erstellt werden, die den Anforderungen und Bedürfnissen des Entwicklers entsprechen.[8]

[7] online: Godot Engine documentation, 2024b (30.09.2024)
[8] online: Godot Engine documentation, 2024b (30.09.2024)

2.2 WebSockets

Das WebSocket Protokoll ist ein Netzwerkprotokoll, welches auf TCP[9] basiert, einem verbindungsorientierten Transportprotokoll. TCP stellt dabei eine Verbindung zwischen zwei Kommunikationsendpunkten, den sogenannten *Sockets* her. Der Prozess beginnt damit, dass ein Client sich über einen *WebSocket-Handshake* mit einem Webserver verbindet. Dabei sendet der Client alle benötigten Identifikationen in einer Anfrage über die TCP-Verbindung an den Server. Wenn dieser die Anfrage akzeptiert, sendet er eine Antwort zurück und die WebSocket-Verbindung ist eingerichtet. Danach bleibt der Kommunikationskanal geöffnet. Für einen Informationsaustausch muss dann, im Gegensatz zu HTTP[10], keine Clientanfrage mehr gesendet werden. WebSocket ist zudem eine bidirektionale Verbindung, was bedeutet, dass der Austausch von Daten in zwei Richtungen möglich ist.[11] Abbildung 3 veranschaulicht dabei das Konzept des Protokolls.

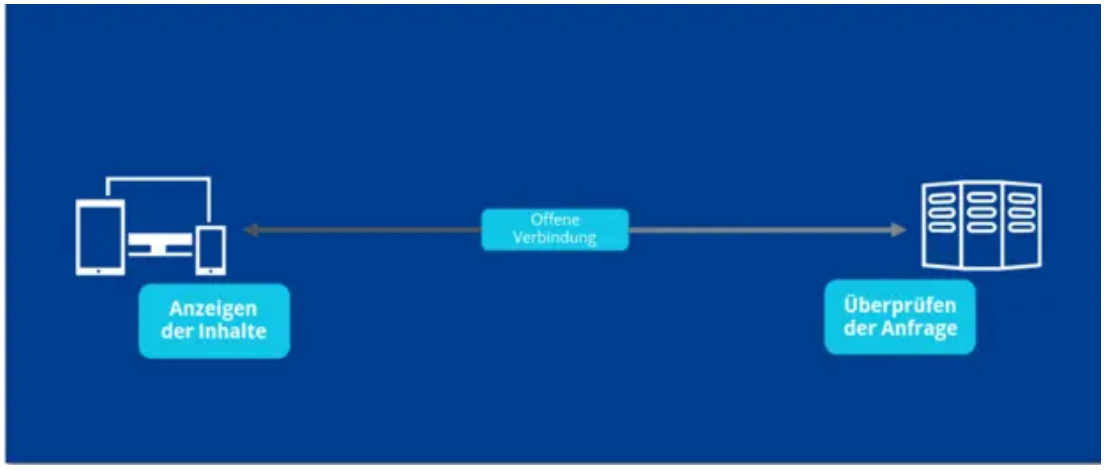


Abbildung 3: Veranschaulichung einer WebSocket-Verbindung[12]

WebSockets werden genutzt, wenn ein schneller Verbindungsaufbau benötigt wird, z.B. bei Live-Chats, Onlinespielen oder Nachrichtentickern[13]. Bei einem Leistungsvergleich zwischen HTTP und WebSocket wurde festgestellt, dass die durchschnittliche Dauer einer HTTP-Anfrage bei 107 Millisekunden liegt, die einer WebSocket-Anfrage bei 83 Millisekunden. Laufen mehrere Anfragen parallel, vergrößert sich diese Differenz: 50 parallele HTTP-Anfragen dauern ungefähr 5 Sekunden, während die gleiche Anzahl Web-Socket-Anfragen nur 180 Millisekunden benötigt.[14]

[9] Abkürzung für Transmission Control Protocol
[10] Abkürzung für Hypertext Transfer Protocol
[11] online: IONOS Digital Guide, 2024 (01.10.2024)
[12] online: IONOS Digital Guide, 2024 (01.10.2024)
[13] Vgl. online: IONOS Digital Guide, 2024 (01.10.2024)
[14] LUECKE, 2018

Um eine WebSocket Verbindung zu schließen, muss ein sogenannter *Closing-Handshake* veranlasst werden. Dieser kann sowohl vom Server als auch vom Client ausgehen. Haben jeweils der Client und der Server einen Closing-Handshake empfangen bzw. gesendet, wird der TCP-Kanal vom Server beendet.[15]

2.3 WebRTC

RTC steht für Real Time Communication. Übersetzt bedeutet es Echtzeitkommunikation. Genau für diesen Zweck wurde diese Technologie 2011 von Google als quellcodeoffenes Projekt veröffentlicht. Seit 2021 ist WebRTC ein von der Internet Engineering Task Force (IETF) und dem World Wide Web Consortium (W3C) anerkannter Standard. Die Technologie wird insbesondere dann verwendet, wenn Daten schnell zwischen zwei oder mehreren Geräten übertragen werden müssen. So zum Beispiel wird das Kommunikationsprotokoll zur Datenübertragung von Videoanrufen, Chats, Screensharing-Anwendungen oder internetbasierten Computerspielen benutzt. Ähnlich wie WebSockets kann diese Technologie ohne zusätzliche Software oder Plugins direkt im Browser gestartet werden. WebRTC basiert allerdings auf einem Peer-to-Peer-System, bei dem die Teilnehmer direkt miteinander kommunizieren können, ohne einen zwischengeschalteten Server.[16,17]

Für WebRTC gibt es zahlreiche *JavaScript-APIs* (Application Programming Interfaces). Die Schnittstelle *RTCPeerConnection* ermöglicht den Verbindungsaufbau. Bereits hier müssen später verwendete *STUN*- oder *TURN*-Server angegeben werden. Eine Erklärung folgt in den kommenden Abschnitten. Für die eigentliche Datenübertragung existieren die Komponenten *MediaStream* oder *RTCDataChannel.* Sie greifen auf Quellen zu, die Mediendaten erzeugen. Dies können beispielsweise Mikrofon oder Kamera sein.

Diese Schnittstellen müssen in jeder Implementierung enthalten sein. Die Bezeichnungen können leicht variieren. Für den Verbindungsaufbau müssen im Vorhinein Informationen über die einzelnen Peers ausgetauscht werden. Dieser Vorgang wird als Signalisierungsprozess bezeichnet. Die genaue Funktionsweise dieses Datentransfers ist nicht Teil des WebRTC-Standards und kann frei gewählt werden. Üblicherweise wird dazu eine WebSocket-Verbindung aufgebaut. In Sonderfällen könnten aber auch externe Messenger-Dienste oder E-Mails verwendet werden. Zuerst generiert ein Teilnehmer ein *Session Description Protocol* (kurz SDP) Angebot. Es enthält Informationen über die unterstützten Medientypen (Audio, Video), Transportprotokolle, Verschlüsselungsverfahren, die verwendete WebRTC-Version und Verbindungsdaten wie zum Beispiel die IP-Adresse und Port, über welche die

[15] GORSKI, 2015
[16] online: GISBERTZ, 2023 (17.10.2024)
[17] online: ComputerWeekly.de, 2024 (05.11.2024)

Applikation erreichbar ist. Dabei ist zu beachten, dass sich all die Informationen nur auf den einen Teilnehmer beziehen. Diese generierten Daten werden dann über den frei wählbaren Signalisierungsprozess an alle anderen Teilnehmer zugestellt. Sie speichern die empfangenen Informationen und ermitteln ihrerseits eine SDP-Antwort. Jene enthält, wie das Angebot, dieselben Daten nur in Bezug auf das eigene Gerät. Danach werden alle gesammelten Informationen an den Initiator zurückgesendet. Dieser speichert sie wiederum.[18,19,20,21,22,23,24]

Die meisten potenziellen Verbindungsteilnehmer befinden sich aus Sicherheitsgründen oder aufgrund des Mangels an IPv4-Adressen hinter NAT-Boxen oder Firewalls. In einem privaten Haushalt sind beispielsweise alle Geräte in einem lokalen Netzwerk. Abgesehen vom Router hat jedes Gerät eine private IP-Adresse, die für externe Geräte nicht erreichbar ist, da für diese Art von Adressen kein Routing existiert. Der Router besitzt eine *Network Adress Translation* (NAT)-Tabelle. Mithilfe dieser übersetzt er die privaten Adressen in öffentliche und umgekehrt. So können zwei Geräte in unterschiedlichen Haushalten miteinander kommunizieren, obwohl beide nur private Adressen besitzen.[20,25]

In den ausgetauschten SDP-Informationen wird unter anderem festgelegt, an welche Netzwerkadresse und Port die Daten gesendet werden sollen. Das Protokoll ist im OSI-Referenzmodell auf der Anwendungsschicht angesiedelt. Besitzt ein Teilnehmer eine private Adresse, wird diese in den SDP-Daten angegeben. Der Router übersetzt nur die Adressen auf der Internetprotokollebene. Die lokale Netzwerkadresse in den SDP-Daten bleibt jedoch aufgrund der unterschiedlichen Protokollschichten unverändert. Beim Senden der eigentlichen WebRTC-Nachrichten wird dann die lokale Adresse des Kommunikationspartners als Ziel angegeben. Da für private Netzwerkadressen kein Routing existiert, verwirft der Router diese und die gesendeten Informationen erreichen nie den anderen Teilnehmer. Sollten alle Kommunikationspartner eine öffentliche Netzwerkadresse besitzen, kann bereits nach dem Austausch der SDP-Daten eine Verbindung hergestellt werden.[21,24,25]

Für den Verbindungsaufbau zwischen Teilnehmern mit privaten IP-Adressen wird das ICE (Interactive Connectivity Establishment) Protokoll verwendet. Dies ermittelt eine Route, über welche die Teilnehmer untereinander erreichbar sind. Zuerst wird die öffentliche IP-Adresse ermittelt. Hierfür muss eine Anfrage an den zu Beginn des Verbindungsaufbaus angegebenen STUN-Servers geschickt werden. Dieser antwortet

[18] online: Digital Samba, 2023 (05.11.2024)
[19] online: ATAUL, 2018 (05.11.2024)
[20] online: GISBERTZ, 2023 (17.10.2024)
[21] online: Chris, 2020 (05.11.2024)
[22] online: ComputerWeekly.de, 2024 (05.11.2024)
[23] online: READ-MCFARLAND, 2023 (05.11.2024)
[24] online: ICHI.PRO, 2024 (05.11.2024)
[25] online: CURRIER, 2022 (05.11.2024)

mit einer Adresse und Port, über welchen das Gerät öffentlich erreichbar ist. Diese Angaben bilden einen ICE-Kandidaten, welcher Bestandteil der aufzubauenden Routeninformationen ist. Nach dem SDP-Austausch werden diese Daten von jedem Teilnehmer generiert und an alle anderen Partner gesendet. In der Regel reicht dies aus, um eine Verbindung zu imitieren. [26,27]

In besonders geschützten Netzwerken kann es vorkommen, dass zwar ausgehende Verbindungen zur Außenwelt möglich sind, eingehende jedoch blockiert werden. Aufgrund der Peer-to-Peer Architektur kann dadurch keine direkte Verbindung initiiert werden. Um dennoch eine WebRTC Datenaustausch zu erreichen, kommen TURN (Traversal Using Relays around NAT, zu Deutsch: *Durchquerung von NATs unter Verwendung von Relaisservern*)-Server zum Einsatz. Jeder Teilnehmer knüpft einen Kontakt zum TURN-Server. Dadurch muss keine eingehende Verbindung zu dem Gerät im geschützten Netzwerk aufgebaut werden. Der gesamte Datenverkehr fließt dann über den TURN-Server, was je nach dessen Auslastung und geografischen Standort zu einer höheren Latenz führen kann. Die über den Host erreichbaren Teilnehmer (*Relayed Candidates*) bilden ebenfalls ICE-Kandidaten.[26,27]

Jetzt besitzt jeder Kommunikationspartner eine Liste von ICE-Kandidaten, über die er die anderen erreichen kann. Bei einem Informationsaustausch wird stets die schnellste Route gewählt. Sollte diese unterbrochen werden, kann auf andere angegebene Kandidaten ausgewichen werden. So kann beispielsweise ein zuvor nicht genutzter TURN-Server mitsamt der damit verbundenen *Relayed Candidates* zum Einsatz kommen.[26,27]

Eine WebRTC-Verbindung gilt in der Regel als sicher. Der WebRTC-Standard sieht vor, dass jede Komponente verschlüsselt ist. Dies führt dazu, dass der Datenverkehr von sich aus geschützt ist. Verwendet werden Verfahren wie DTLS (Datagram Transport Layer Security) und SRTP (Secure Real-time Transport Protocol). Durch das Peer-to-Peer-Modell besteht in der Regel nur ein Informationsfluss zwischen den Teilnehmern. Nachrichten können nur im Falle eines TURN-Servers mitgeschnitten werden. Da diese Technologie im Browser läuft und somit eine Art virtuellen Maschine darstellt, hat es ein Virus auf dem Rechner schwer, die Verbindung zu stören oder mitzuschneiden. Der RTC-Standard lässt allerdings die Implementierung des Signalisierungsprozesses offen, wodurch dieser Prozess sehr flexibel gestaltet werden kann. Dabei sollte beachtet werden, dass ein eventueller Signalisierungsserver alle wichtigen Verbindungsinformationen mitlesen kann, wodurch er sich als der Gegenüber ausgeben könnte. Dagegen helfen jedoch die Verschlüsselungsverfahren. Bei DTLS muss sich jeder Partner von einer Certificate Authority (Zertifizierungsstelle) zertifizieren lassen, um sicherzustellen, dass er tatsächlich der Gesprächspartner ist.

[26] online: Chris, 2020 (05.11.2024)
[27] online: CURRIER, 2022 (05.11.2024)

In der Vergangenheit gab es einen sogenannten IP-Leak bei RTC-Verbindungen. Trotz einer VPN-Verbindung war es durch STUN-Server möglich, die echte IP-Adresse des Beteiligten zu ermitteln.[28,29,30,31]

2.4 Remote Procedure Call

Die Bezeichnung RPC[32] bedeutet übersetzt „Aufruf einer entfernten Prozedur“. Es ist ein Konzept, welches den Informationsaustausch zwischen Systemprozessen regelt und arbeitsteilige Strukturen in Client-Server-Architekturen realisiert. Definiert wurde es 1984 als ein synchroner Mechanismus, welcher Daten als Prozeduraufruf zwischen zwei Adressräumen über ein schmalbandiges Netz transferiert. Ein solcher RPC-Aufruf folgt immer einem bestimmten Ablauf, dabei sind auf Empfänger- und Senderseite spezielle Instanzen beteiligt, die Stubs (Stummel) genannt werden. Stubs dienen als Prozedur-Schnittstellen und verbergen die Komplexität der Netzwerkkommunikation für den Entwickler, indem sie lokale Einheiten simulieren. Der Ablauf eines Aufrufs erfolgt folgendermaßen:

Der Client-Code ruft eine Stub-Prozedur auf, woraufhin der Client Stub gemäß dem RPC-Protokoll aus den übergebenen Parametern eine sendefähige Nachricht generiert. Dies wird auch *Marshalling* genannt. Dann kontaktiert der Client-Stub das Kommunikationssystem des lokalen Rechners, welches für den Nachrichtenaustausch zwischen Client und Server sorgt und die Nachricht wahlweise über TCP/IP oder UDP/IP versendet. Nach Ankunft der Nachricht entpackt der Server-Stub die Parameter der empfangenen Nachricht auf Basis des RPC-Protokolls (*Unmarshalling*). Anschließend werden die Parameter übergeben und sorgen für den Aufruf einer Server-Prozedur. Der resultierende Funktionswert wird wieder an den Stub kommuniziert. Der Prozess läuft jetzt umgekehrt wieder ab: Der Rückgabewert wird mittels Marshalling verpackt, an den Client gesendet und dort erneut entpackt.

RPCs werden in vielen Bereichen eingesetzt, z.B. als Baustein für Webservices, verteilte Anwendungen, dezentrale Peer-to-Peer-Netzwerke oder Blockchains. Zu ihren Vorteilen gehören geringe Verarbeitungszeiten, die Modularisierung bzw. Auslagerung von Prozessen sowie die Skalierbarkeit der Client-Server-Architekturen. Jedoch gibt es auch einige Nachteile, wie zum Beispiel das Fehlen von einheitlichen Standards, da die Umsetzungen meist firmenspezifisch sind. Außerdem kann die Aufteilung der Prozesse die Fehleranfälligkeit erhöhen, was zu Verzögerungen, Ausfällen und einer aufwendigeren Fehlerbehandlung führt.[33]

---

[28] online: GISBERTZ, 2023 (17.10.2024)
[29] online: ComputerWeekly.de, 2024 (05.11.2024)
[30] online: READ-MCFARLAND, 2023 (05.11.2024)
[31] online: LUBER; tutanch, 2020 (05.11.2024)
[32] Abkürzung für Remote Procedure Call
[33] Ionos Redaktion, 2020

3 Konzeption

3.1 Geplanter Aufbau der Anwendung

Bevor die Umsetzung der Anwendung erfolgt, werden zuerst die Ideen und Konzepte für die einzelnen Funktionen und Bestandteile vorgestellt. Dies umfasst auch den Aufbau der Anwendung. Er soll in klar strukturierte Bereiche unterteilt sein, diese werden im Folgenden kurz vorgestellt.

• Startseite

Startet der Spieler die Anwendung, ist dies die erste Seite, die ihm angezeigt wird. Auf der Startseite soll ausgewählt werden können, ob das Spiel offline (lokaler Einzel- und Mehrspieler) oder online (nur Mehrspieler) gespielt werden möchte.

• Szene zum Verbindungsaufbau

Hat sich der Spieler für eine Online-Partie entschieden, wird er zum Verbindungsaufbau weitergeleitet. Hier wird mithilfe von Netzwerktechnologien, wie WebSockets oder WebRTC versucht, eine stabile Verbindung mit einem anderen Spieler herzustellen. Wichtig ist hierbei die Synchronisierung zwischen beiden Parteien.

• Levelauswahl-Szene

Diese Szene spielt sowohl beim offline als auch beim online spielen eine Rolle. Hier werden verschiedene Einstellungen für das bevorstehende Spiel vorgenommen. Diese beeinflussen verschiedene Parameter, wie die Schwierigkeit, das gewünschte Spielfeld sowie den Spielmodus (kompetitiv oder kooperativ). Im Mehrspieler sollen beide Spieler über ihre Wünsche abstimmen können.

• Level-Szene

Nach dem Festlegen der Parameter startet die Level-Szene. In dieser Szene findet das eigentliche Spiel statt. Dabei unterscheidet sich das Spielerlebnis durch die vorher eingestellten Parameter im Hinblick auf die Geschwindigkeit, das Aussehen des Spielfelds und den Spielmodus. Nach dem Ende einer Runde soll die Auswahl bestehen, mit den gleichen Einstellungen ein neues Spiel zu starten oder zurück zur Levelauswahl-Szene zu gelangen.

Zur Veranschaulichung des geplanten Aufbaus der Anwendung werden die einzelnen Szenen, deren Funktionen und Verbindungen in der folgenden Abbildung dargestellt.

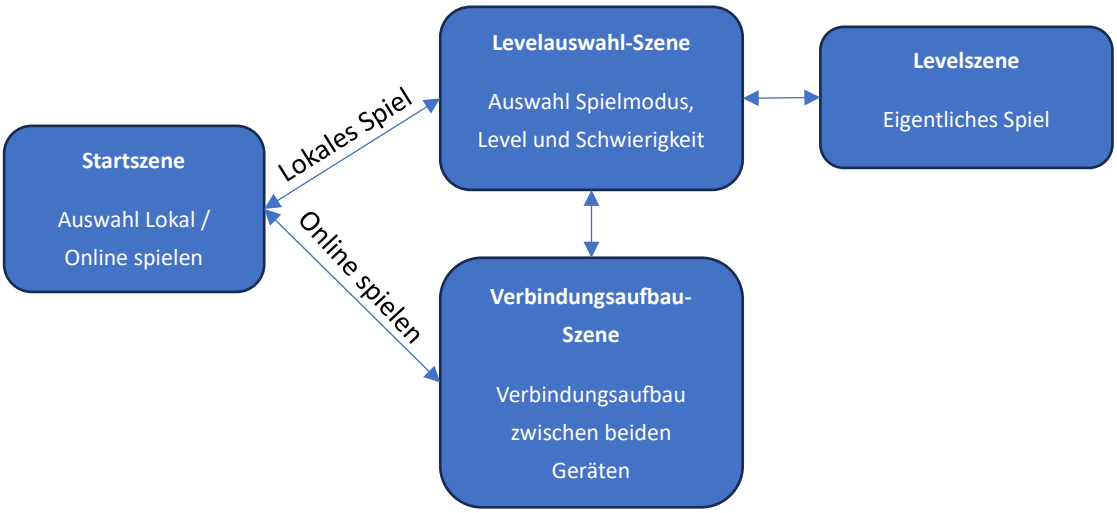


Abbildung 4: Skizze zum geplanten Aufbau der Anwendung

Es ist vorgesehen das, dass eigentliche Spielgeschehen über eine WebRTC-Verbindung abgewickelt wird. Wie in Kapitel 2.3 beschrieben, müssen hierfür erst SDP-Daten und anschließend alle ICE-Kandidaten ausgetauscht werden. Dies soll über ein WebSocket System geschehen.

3.2 Konzept zur Umsetzung der Spiellogik

Im Spiel soll die Schlange aus mehreren Segmenten bestehen. In Godot kann hierfür der Node *Line2D* genutzt werden. Dieser zeichnet eine 2D-Polylinie aus einzelnen Punkten bzw. Segmenten, welche miteinander verbunden sind. Dabei repräsentiert das erste Segment den Kopf, die restlichen bilden den Körper der Schlange. Weiterhin muss die Schlange sich fortbewegen können, die Richtungssteuerung soll dabei über Tasteneingaben erfolgen. Die Positionen der Segmente werden dabei nacheinander in Bewegungsrichtung verändert, sodass der Kopf immer an der Spitze bleibt und die restlichen Körpersegmente ihm folgen.

Das Spielfeld selbst soll aus einem Raster bestehen, wobei jedes Feld eine bestimmte Größe hat. Für die Bewegung und Kollisionsüberprüfung ist es am besten, wenn sich jedes Segment in genau einem Feld dieses Rasters befindet. Die Positionen des Spielfeldrandes oder zusätzlichen Hindernissen auf dem Spielfeld sollen dabei in einem zusätzlichen Array oder einer Liste gespeichert werden, um sie für die Kollisionsüberprüfung zu verwenden. Um das Wachstum der Schlange zu ermöglichen, wird überprüft, ob der Kopf der Schlange auf der gleichen Position wie eine Frucht liegt. Dies wird ebenfalls immer auf einem festen Rasterfeld platziert, sodass die Positionen leicht miteinander verglichen werden können. Wenn die Schlange eine Frucht frisst, wird der Datenstruktur, welche die Segmente der Schlange speichert, am Ende ein weiteres Körperteil hinzugefügt.

Während des Spielgeschehens müssen zwei Arten von Kollisionen geprüft werden. Erstens muss sichergestellt werden, dass die Schlange nicht den Spielfeldrand oder andere Hindernisse berührt. Dazu wird die Position des Kopfes der Schlange mit den gespeicherten Positionen in der Hindernisliste verglichen. Wenn eine Übereinstimmung gefunden wird, endet das Spiel. Zweitens muss die Schlange auf eine Kollision mit sich selbst überprüft werden. Hierzu wird die aktuelle Position des Kopfes mit den Positionen der restlichen Segmente der Schlange verglichen. Falls der Kopf eine Position einnimmt, die bereits von einem anderen Segment belegt ist, kommt es zu einer Kollision, und das Spiel endet.

Zur Veranschaulichung ist der gesamte Spielablauf sowie die zu berücksichtigenden Ereignisse in einem Flussdiagramm (Abbildung 5) dargestellt. Dabei befindet sich das Spiel nach dem Start in einer Schleife, bei welcher zuerst die Tasteneingaben für die Bewegungsrichtung überprüft werden. Anschließend wird die Schlange um ein Feld in die entsprechende Richtung bewegt. Nach Abschluss eines Bewegungszyklus finden die Überprüfungen auf Kollision und auf das Einsammeln einer Frucht statt. Hat eine Kollision mit einem Hindernis oder dem eigenen Körper stattgefunden, führt dies zum Ende der aktuellen Spielrunde. Wurde eine Frucht eingesammelt, wird die Schlange um ein Segment erweitert und die Frucht wird neu auf dem Spielfeld platziert. Im Anschluss oder falls keine Frucht eingesammelt wurde, führt die Schleife zurück zur Überprüfung der Tasteneingabe.

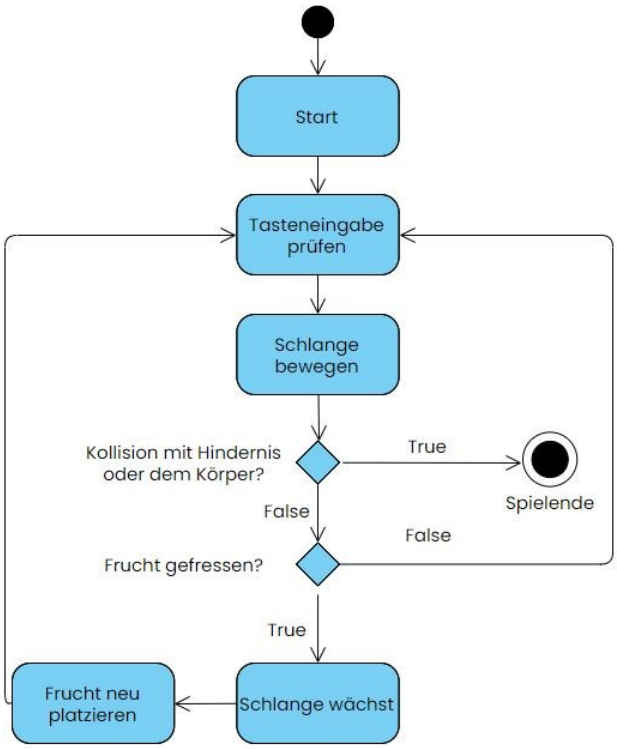


Abbildung 5: Flussdiagramm zum Spielablauf

3.3 Geplante Funktionsweise des Mehrspielers

Bei einer Vielzahl von modernen, mehrspielerfähigen Anwendungen übernimmt ein Server die Simulation der gesamten Spielwelt. Die eigentlichen Spieler bekommen jene über das Internet mitgeteilt. So verlagert sich die Hauptrechenleistung auf den Host. Das Senden der gesamten Welt nach jedem Serverzyklus würde zu einer deutlichen Netzwerküberlastung führen. Stattdessen bekommen Clients in regelmäßigen Abständen nur die Änderungen ihres letzten Standes, auch genannt Deltakompression, mitgeteilt. Dadurch kann der Netzwerkverkehr deutlich reduziert werden. In der entgegengesetzten Richtung übermitteln die Clients alle getätigten Spielereingaben an den Server. Das Spielgeschehen erscheint allerdings noch sehr ruckartig. Es wird nur alle 30 bis 100 Millisekunden (ms) eine Veränderung auf Basis der Serverupdates angezeigt. Dies stellt eine Bildfrequenz von ca. 10 bis 30 Hertz dar, welche je nach Netzwerkverbindung schwanken kann, indem beispielsweise Pakete verloren gehen oder zu spät ankommen. Um eine flüssige Bewegung darzustellen, werden zwischen den Host-Updates kleinere Veränderungen in kurzen Zeitabständen berechnet und abgebildet. Der Client misst die Zeitspanne zwischen zwei empfangenen Updates und erzeugt basierend auf diesen Zeitabständen und den beiden Weltzuständen eine flüssige Bewegung. Verändert ein Spieler beispielsweise seinen Standort, wird in jedem Programmdurchlauf eine kleine Wegdifferenz von seiner Startposition in Richtung Zielposition addiert. So entstehen genügend Bildänderungen und das menschliche Auge kann die einzelnen Bilder nicht mehr wahrnehmen. Eine flüssige Bewegung entsteht. Bei dieser Technik ist der Server dem Client immer voraus. Beispielsweise wird in einem Update die Position einer Spielfigur gesendet. Während sich die Figur auf dem Server bereits an der Position befindet, muss sie sich auf der Clientseite noch hinbewegen. Sollte der Server selbst aktiv am Spiel beteiligt sein, besitzt er dadurch einen Vorteil.[34]

Der Mehrspielermodus soll mithilfe der im vorhergehenden Absatz erwähnten Methoden umgesetzt werden. Dabei ist zu beachten, dass die Verbindung zwischen den beiden Geräten nach wie vor auf einer Web-RTC Verbindung, also einem Peer-to-Peer System, beruhen soll. Die Spiellogik folgt jedoch einem Client-Server-Ansatz. Es ist vorgesehen, dass einer der beiden Spieler die Rolle des Servers übernimmt. Er ist für die Bewegung der Schlangen und die Interaktionen mit der Spielwelt verantwortlich. Der andere Spieler ist der Client. Dieser soll nur die neuen Schlangenpositionen in regelmäßigen Abständen bekommen.

In Abbildung 6 ist die geplante Funktionsweise des Servers vereinfacht dargestellt. Die Schlangenbewegung ähnelt der im Offline-Modus. Nachdem ein Zyklus abgeschlossen ist, wird zuerst geprüft, ob eine Frucht gegessen wurde. Ist dies der

[34] online: Source Multiplayer Networking - Valve Developer Community, 2024 (01.11.2024)

Fall, muss die Schlange wachsen. Da während des Fressvorgangs Geräusche abgespielt werden sollen, muss der Client auch darüber informiert werden. Gleiches geschieht, wenn die Schlange gestorben ist. Danach beginnt der Zyklus von vorn. Parallel und unabhängig davon existiert ein zweiter Programmablaufpfad. Er soll prüfen, ob der Client ein Update benötigt und bei Bedarf die neuen Schlangenpositionen an ihn senden.

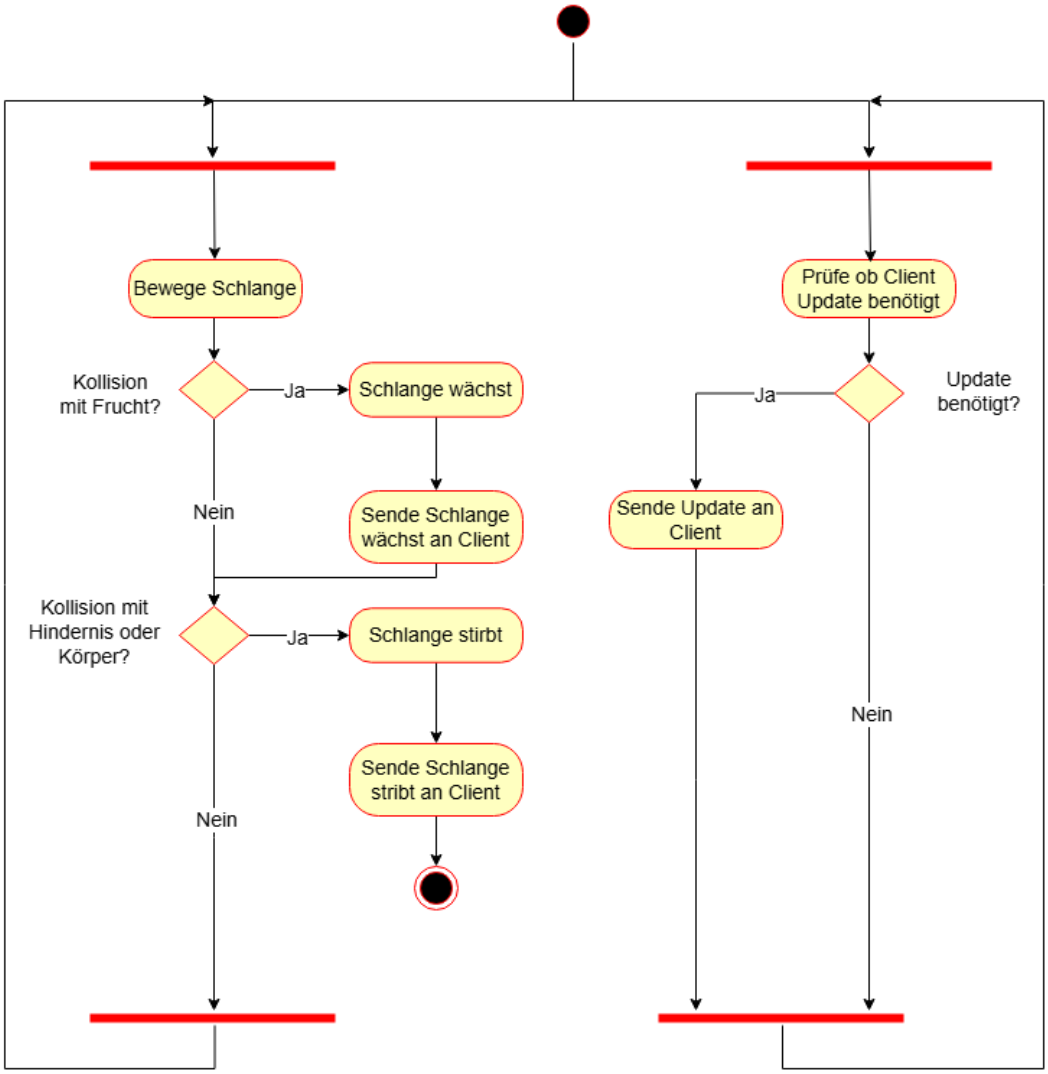


Abbildung 6: Aktivitätsdiagramm Spielserver

Auf Basis der regelmäßigen Positionsupdates vom Server soll die Schlange auf dem Client bewegt werden. Dadurch wird die Synchronität beider Geräte gewährleistet. Abbildung 7 zeigt, wie der Client geplant ist. Wie beim Host sollen beide Programmstränge unabhängig voneinander ablaufen. Der Linke ist für die Interpretation der Nachrichten zuständig. So sollen bei einem Positionsupdate alle gesendeten Punkte gespeichert werden. Zu diesen Positionen soll sich die Schlange bewegen. Hier sieht man, dass der Client zeitlich nachläuft. Während sich auf dem Server die Schlange schon an den gesendeten Punkten befindet, muss sie sich auf der Clientseite noch dahin bewegen. Um eine möglichst flüssige Bewegung

herzustellen, ist es vorgesehen, die Zeitspanne zwischen den letzten beiden Updates zu messen. Daraus soll ein Faktor ermittelt werden, um den die Schlange in jedem rechten Programmstrangzyklus in Richtung der Zielpositionen gesetzt wird.

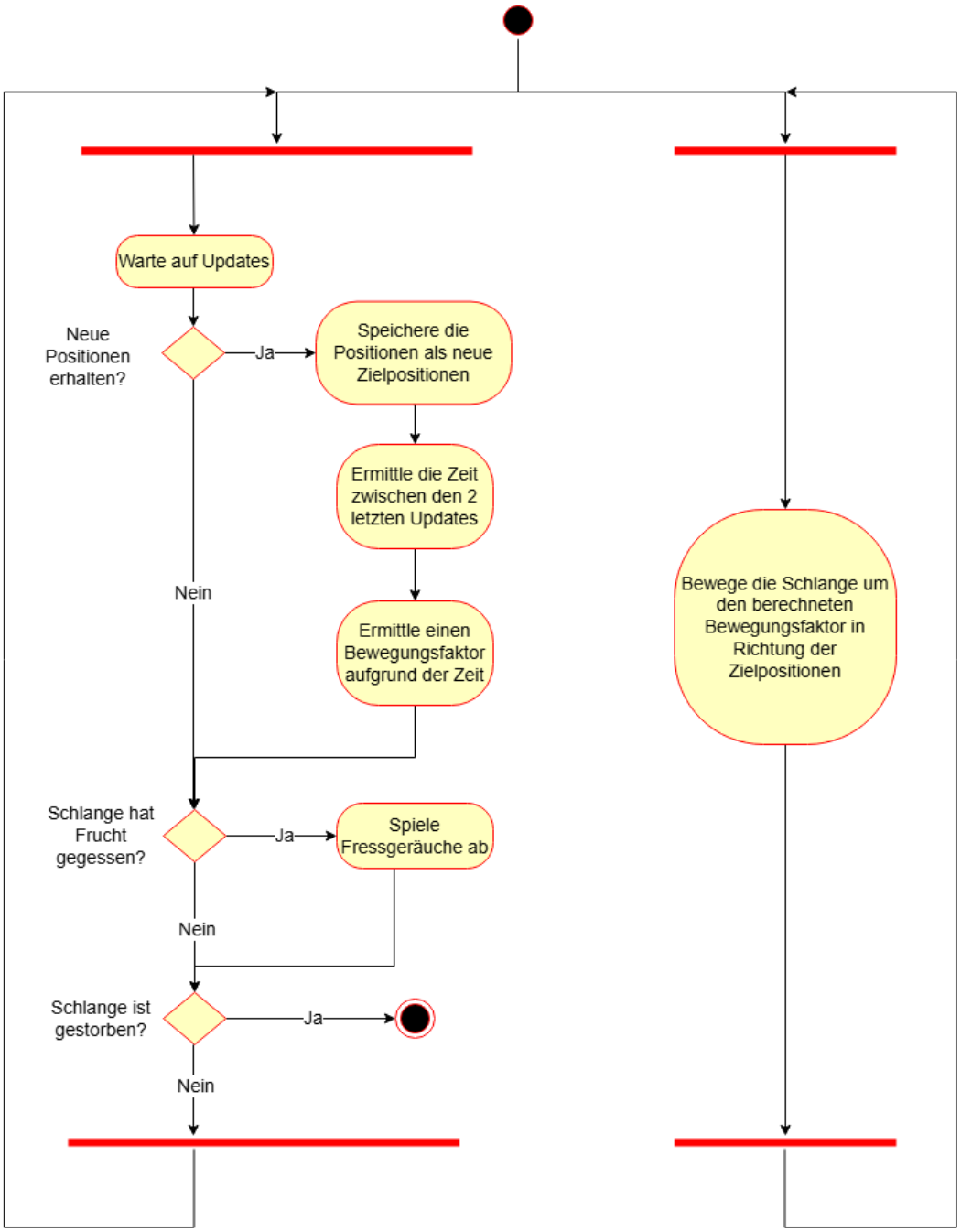


Abbildung 7: Aktivitätsdiagramm Spielclient

Bekommt der Client die Nachricht, dass eine Frucht gegessen wurde, sollen Fressgeräusche abgespielt werden. Sollte die Schlange gestorben sein, endet der Zyklus. Die beiden Spieler bekommen zeitgleich eine Nachricht, welche bestätigt, dass das Spiel vorbei ist.

4 Umsetzung

4.1 Aufbau und Gestaltung der Anwendung

Auf der Startseite der Anwendung (siehe Anhang 1) hat der Spieler die Auswahl, ob er lokal oder online spielen möchte. Dazu wurden zwei Buttons hinzugefügt und die *pressed*-Signale mit Methoden verknüpft, welche ihn jeweils zu einer anderen Szene weiterleiten. Die Methoden sind in Listing 1 abgebildet. Der Szenenwechsel erfolgt dabei mittels der Methode *GetTree().ChangeScene(),* bei welcher der Pfad der zu ladenden Szene angegeben werden muss. Dieser Pfad ist der Speicherort der Szene im Projektverzeichnis, wobei das *res://*-Verzeichnis der Basisordner des Projektes ist.

```
public void _on_Einzelspieler_pressed()
	{
		GetTree().ChangeScene("res://Szenen/Einstellungen.tscn");
		GlobalVariables.Instance.OnlineGame = false;
	}

public void _on_Verbindung_erstellen_pressed()
	{
		GetTree().ChangeScene("res://Szenen/Verbindungseinstellungen.tscn");
		GlobalVariables.Instance.ConfirmationDialog =
		ResourceLoader.Load("res://Szenen/ConfirmationDialog.tscn");
	}
```

Listing 1: Verknüpfte Methoden zum Szenenwechsel

Hat sich der Spieler für das lokale Spiel entschieden, wechselt die Szene zur Levelauswahl (Anhang 2), wo die Schwierigkeit, das zu spielende Level sowie der gewünschte Spielmodus ausgewählt werden können. Die Funktionalität dahinter wird genauer in Abschnitt 4.2.5 erläutert. Sobald eine Auswahl getroffen wurde, kann das Spiel über einen Wechsel zur jeweiligen Level-Szene gestartet werden (Anhang 3).

Wenn der Spieler online spielen möchte, wechselt die Szene zu einem Menü, in welchem zwischen verschiedenen Verbindungsarten gewählt werden kann (Anhang 4). Diese umfassen einerseits den manuellen WebRTC Verbindungsdatenaustausch und andererseits das Beitreten einer Lobby über eine WebSocket-Verbindung. In der Lobby können beigetretene Spieler sich über einen Chat austauschen, Räume erstellen oder diesen beitreten. Beim manuellen Datenaustausch müssen die Spieler die benötigten Informationen über ein externes Medium, wie Messenger Dienste austauschen. Wurde eine Verbindung hergestellt, wird zur bereits bekannten Levelauswahl-Szene weitergeleitet.

Für die Gestaltung der Anwendung wurden frei verwendbare Designelemente verwendet, wobei die Quellen sämtlicher verwendeter Ressourcen von Drittanbietern ordnungsgemäß angegeben wurden. Hierfür gibt es auf der Startseite einen zusätzlichen Button, welcher die Quellen anzeigt. Für die Benutzeroberfläche bzw. die Hintergrundgestaltung kam das Paket *World of Solaria* zum Einsatz. Dieses enthält verschiedene Kacheln (*Tiles*) und Grafiken, darunter Gebäude, Werkzeuge, Grasböden und Pflanzen, welche für die Gestaltung der einzelnen Level verwendet wurden. Als alternative Schriftart wurde *RO Spritendo* gewählt, welche sich stilistisch an dem Schriftzug des bekannten japanischen Videospiel- und Konsolenherstellers Nintendo orientiert. Um das Spielerlebnis zusätzlich zu verschönern und lebendiger zu gestalten, wurde passende Musik integriert. Dabei wurden Titel von Kevin MacLeod verwendet, die unter der *Creative Commons-Lizenz* veröffentlicht wurden.

Für die Steuerung der Lautstärke wurde in der Anwendung ein Lautstärkeregler in Form eines horizontalen *Sliders* implementiert, sodass die Spieler die Musik nach ihren Wünschen anpassen können. Listing 2 zeigt die dafür verantwortliche Klasse. Die Lautstärke wird den *ProjectSettings,* welche auch nach Schließen der Anwendung gespeichert bleiben, entnommen und dem *Slider* als Wert zugewiesen. Die Methode *SetVolume()* sorgt bei Änderung des Wertes für eine Umwandlung in Dezibel und wendet die Änderung auf den Audiobus *Master* an. Anschließend wird der Wert wieder in den *ProjectSettings* gespeichert.

```
public class VolumeSlider : HSlider
{
    private string settingKey = "audio/volume";
    private HSlider _volumeSlider= GetParent().GetNode<HSlider>("VolumeSlider");

    public override void _Ready()
    {
        float savedVolume = ProjectSettings.HasSetting(settingKey) ?
                            (float)ProjectSettings.GetSetting(settingKey) :
                              50.0f;
        _volumeSlider.Value = savedVolume;
        SetVolume((float)_volumeSlider.Value);
    }

    private void OnVolumeChanged(float value)
    {
        SetVolume(value);
        ProjectSettings.SetSetting(settingKey, value);
        ProjectSettings.Save();
    }

    private void SetVolume(float value)
    {
        float db = Mathf.Lerp(-40, 0, value / 100.0f);
        AudioServer.SetBusVolumeDb(AudioServer.GetBusIndex("Master"), db);
    }
}
```

Listing 2: Klasse zum Einstellen der Lautstärke

4.2 Umsetzung der Spiellogik von Snake

4.2.1 Aufbau des Spielfelds

Das Spielfeld wurde, wie bereits in der Konzeption beschrieben, als ein Raster aufgebaut. Hierfür wurde das gesamte Fenster der Anwendung genutzt, welches eine Auflösung von 1280x720 Pixeln besitzt. Aufgrund dieser Maße wurde sich entschieden, die Größe der einzelnen Felder des Rasters auf 32x32 Pixel festzulegen. Dies führt dazu, dass das Spielfeld insgesamt aus 40x22,5 Rasterfeldern besteht. Die unterste Reihe wird durch die Ränder des Fensters abgeschnitten. Da dieser Bereich ohnehin nicht mehr zum spielbaren Teil gehört, stellt dies jedoch kein Problem dar. Für das eigentliche Spielfeld, auf welchem sich die Schlange bewegen soll, wurden 20x17 Felder in der Mitte des Rasters genutzt. Die restlichen Felder bilden die Spielfeldbegrenzung bzw. den nicht passierbaren Bereich. Auch wenn dieser Teil des Spielfelds nicht von der Schlange betreten werden kann, bietet er Platz für zusätzliche Elemente, die in der Szene angezeigt werden können, z.B. einen Punktestand.

Um ein Feld, auf welchem sich die Schlange bewegen kann, von einem Hindernisfeld, welches bei Kollision zum Spielende führen würde, zu unterscheiden, wurde ein zweidimensionales Array erstellt. Dieses hat die gleiche Größe wie die Maße des Rasters (40x23 Elemente) und speichert für jedes Feld entweder eine 0 oder eine 1. Eine 0 steht für ein freies Feld, eine 1 für ein Hindernisfeld. Um die Positionen dieser Hindernisse für die Kollisionsprüfung zu verwalten, wird zusätzlich eine separate Liste geführt. Diese Liste enthält die genauen Positionen aller Hindernisse. Dafür wird in einer verschachtelten Schleife das Spielfeld-Array durchlaufen. Falls der Wert des aktuellen Feldes 1 beträgt, wird der Index mit 32, der Rasterfeldgröße multipliziert. Zusätzlich wird ein Offset von 16 addiert, damit die gespeicherte Position genau in der Mitte des jeweiligen Feldes liegt. In Listing 3 ist der Quellcode für diesen Vorgang zu sehen.

```
private void SaveObstaclePositions()
{
	for (int y = 0; y < _gameField.GetLength(1); y++)
	{
		for (int x = 0; x < _gameField.GetLength(0); x++)
		{
			if (_gameField[x, y] == 1)
			{
				var obstaclePosition = new Vector2((y*32)+16,
                                                   (x*32)+16);
				_obstacles.Add(obstaclePosition);
			}
		}
	}
}
```

Listing 3: Speichern der Hindernispositionen

Die Methode *SaveObstaclePositions()* befindet sich in der Datei *GameController.cs*. Diese ist unter anderem mit der Level-Szene verknüpft und verwaltet die Hauptlogik und zentrale Steuerung des Spiels. In der folgenden Abbildung ist die Benutzeroberfläche des Godot-Editors mit der Level-Szene abgebildet. Links im Bild sind die einzelnen Nodes in Baumstruktur angeordnet, rechts ist eine Ansicht der Szene zu sehen.

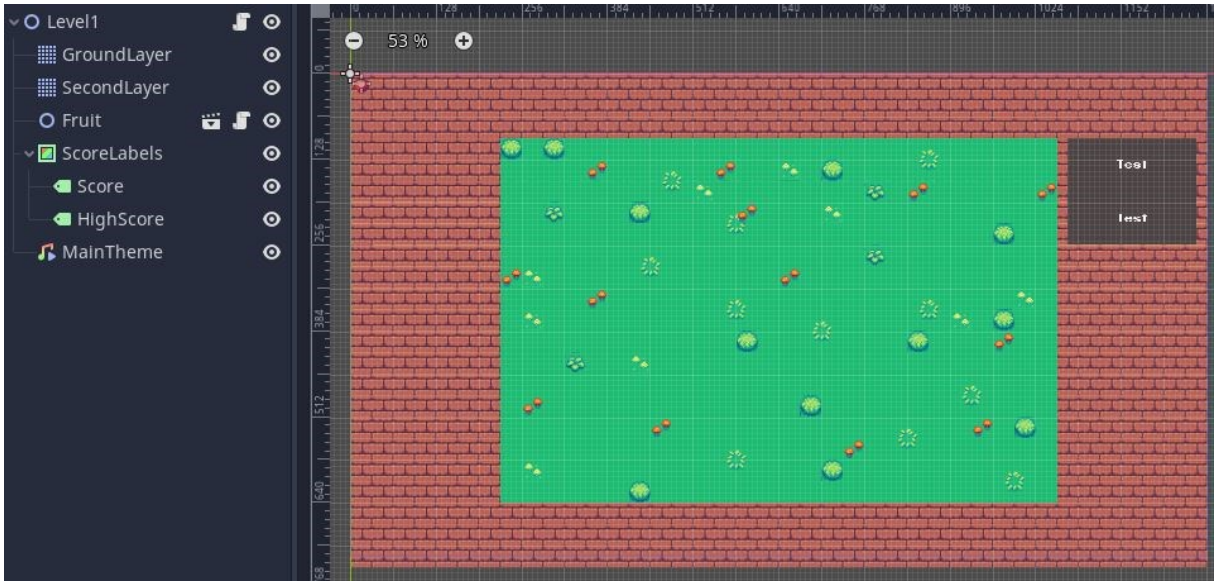


Abbildung 8: Level-Szene im Editor

Die Nodes *GroundLayer* und *SecondLayer* sind vom Typ *TileMap* und bestimmen die grafische Gestaltung der Szene. Dabei besteht *GroundLayer* aus der Grasfläche und den Mauerkacheln, *SecondLayer* aus den Pflanzen auf dem Gras. Weiterhin enthält die Szene einen den Knoten *Fruit* von *Node2D* abgeleitet ist. Dieser stellt die Frucht dar, welche von der Schlange im Spiel eingesammelt werden muss. Die Logik dahinter wird in Abschnitt 4.2.3 genauer erklärt. Darunter befindet sich ein *ColorRect*, welchem zwei *Labels* untergeordnet sind. Diese stellen einen Punktezähler dar, auf welchem der aktuelle Punktestand (*Score*) sowie die höchste Punktzahl für das Level (*HighScore*) angezeigt werden. In Kapitel 4.2.4 wird dies näher erläutert. Zuletzt besteht die Szene aus einem *AudioStreamPlayer2D* mit dem Namen *MainTheme*. Dieser ist für die Musik während des Spiels verantwortlich.

4.2.2 Implementierung der Schlangenlogik

Um eine Struktur für die Umsetzung der Schlange zu erstellen und um auf aufkommende Unterschiede zwischen Offline- und Online-Funktionalitäten einzugehen, wurde eine Vererbungshierarchie aus mehreren Klassen geschaffen. Dies vereinfacht die Anpassung bzw. Erweiterung der Spiellogik für mehrere Modi. Die Klassenstruktur für die Offline-Funktionalität ist in Anhang 5 abgebildet.

Bevor jedoch darauf eingegangen wird, müssen der Aufbau und die Bestandteile der Schlangen-Szene erläutert werden. Diese ist in der folgenden Abbildung zu sehen.

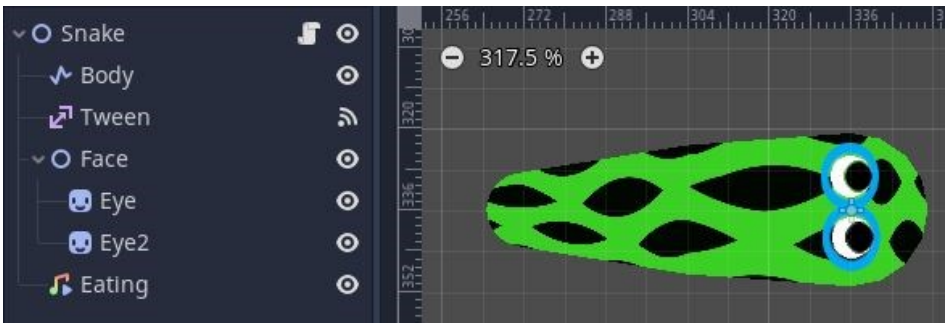


Abbildung 9: Szene der Schlange im Editor

Die einzelnen Körpersegmente der Schlange werden durch den Knoten *Body* vom Typ *Line2D* umgesetzt. Die Segmente sind über die Eigenschaft *Points*, einem Array der Klasse *Vector2,* abruf- und veränderbar. Um dafür zu sorgen, dass die Schlange nicht intervallweise und abrupt bewegt, sondern flüssig animiert wird, wurde ein *Tween*-Node hinzugefügt. Die Bezeichnung *Tween* leitet sich von *Tweening* oder *Inbetweening* ab. Dabei handelt es sich um einen Begriff aus der Animation, „bei dem Einzelbilder zwischen zwei *Keyframes*, auch Schlüsselbilder genannt, eingefügt werden. Ziel ist ein fließender Übergang zwischen den Keyframes herzustellen“[35]. Zusätzlich wurden, um die Schlange äußerlich ansprechender zu gestalten, zwei Augen, welche einem Knoten untergeordnet sind, hinzugefügt. Diese beiden Augen sind jeweils vom Typ *Sprite.* Wie bei der Level-Szene besitzt die Schlangen-Szene einen *AudioStreamPlayer2D*, welcher einen Soundeffekt abspielt, wenn die Schlange eine Frucht einsammelt.

Die zentrale Grundlage für die Logik bildet die Klasse *BaseSnake*. Sie enthält wesentliche Variablen und Methoden, welche für die Schlange allgemein benötigt werden. Dazu gehören Funktionen wie Bewegung, Wachstum und Kollisionsabfragen. Die spezifischen Anforderungen der Offline- und Online-Spielmodi werden in den Kindklassen berücksichtigt. Zu den wichtigsten Attributen gehören unter anderem die Variablen *_direction* und *_directionCache.* Diese sind Objekte der Klasse *Vector2* und speichern die aktuelle bzw. geplante Bewegungsrichtung der Schlange. Sie werden über Tasteneingaben festgelegt. Dabei wird geprüft, dass der eingegebene Richtungsvektor nicht entgegengesetzt der aktuellen Richtung verläuft. Die Schlange würde eine 180 Grad Drehung vollführen und sich selbst fressen. Der entsprechende Quellcode ist in Listing 4 zu sehen.

[35] online: Tweening: Definition, Geschichte und Anwendung. | Adobe, 2024 (28.10.2024)

```
if (Input.IsActionPressed("ui_up") && _direction != Vector2.Down) _directionCache
    = Vector2.Up;
if (Input.IsActionPressed("ui_right") && _direction != Vector2.Left)
    _directionCache = Vector2.Right;
if (Input.IsActionPressed("ui_left") && _direction != Vector2.Right)
    _directionCache = Vector2.Left;
if (Input.IsActionPressed("ui_down") && _direction != Vector2.Up)
    _directionCache = Vector2.Down;
```

Listing 4: Bestimmung des Richtungsvektors über Tasteneingaben

Für die eigentliche Bewegung der Schlange wurden dem Code Verweise auf den *Line2D* und *Tween*-Node in der Szene hinzugefügt. Diese werden in den Methoden *MoveSnake()* und *MoveTween()* benötigt (siehe Anhang 6). Erstere aktualisiert die Bewegungsrichtung der Schlange und definiert eine Interpolation mithilfe des *Tween* Knotens, um die Bewegung flüssig zu gestalten. Die interpolierte Methode ist *MoveTween()* und bewegt die Schlange schrittweise entlang der festgelegten Richtung. Sie berechnet bei jedem Aufruf eine neue Position für die Schlangensegmente, abhängig vom übergebenen Wert (einer Zahl zwischen 0 und 1). Dadurch wird die Bewegung in kleinen Schritten über den festgelegten Zeitraum verteilt, was eine kontinuierliche Animation ergibt. Nachdem ein Bewegungsvorgang abgeschlossen ist, muss überprüft werden, ob die Schlange mit einem Hindernis oder mit sich selbst kollidiert ist. Hierfür wurde die Methode *IsGameOver()* erstellt, welche zwei Abfragen durchführt: Den Abgleich der Position des Schlangenkopfes mit der Liste *Obstacles*, welche in Abschnitt 4.2.1 erläutert wurde und die Überprüfung, ob die Position des Kopfes der Position eines anderen Schlangensegmentes entspricht. Zum besseren Verständnis ist die Methode in Listing 5 abgebildet.

```
protected virtual bool IsGameOver()
{
    foreach (var obstacle in _controller.Obstacles)
    {
        if (_body.Points[0] == obstacle)
        {
            _controller.LoseMessage = ($"{Name}hat ein Hindernis getroffen!");
            return true;
        }
    }

    if (_points.Length >= 3)
    {
        for (int i = 1; i < _points.Length; i++)
        {
            if (_points[0] == _points[i])
            {
                _controller.LoseMessage = $"{Name} hat sich selbst gefressen!";
                return true;
            }
        }
    }
    return false;
}
```

Listing 5: Methode zur Kollisionsüberprüfung

4.2.3 Funktionsweise der Frucht

Die Szene der Frucht zum Wachstum der Schlange ist in Abbildung 10 zu sehen. Sie ist einfach gehalten und besteht lediglich aus einem *Sprite*-Node, welchem ein *AnimationPlayer* untergeordnet ist. Der Sprite enthält das Bild eines Apfels, um diesen im Spiel ansprechender zu gestalten, wird über den *AnimationPlayer* eine pulsartige Animation abgespielt.

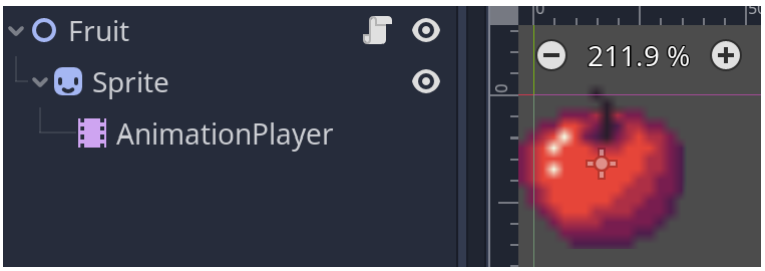


Abbildung 10: Frucht-Szene im Editor

Zu Beginn einer Runde wird eine zufällige Position für die Frucht festgelegt. Hierfür wird über die *GameController* Klasse, welche für die Initialisierung der Frucht verantwortlich ist, die Methode *RandomizePosition()* der Frucht-Klasse aufgerufen (siehe Anhang 7). Diese erzeugt mithilfe der Methode *GetRandomPos()* eine zufällige Position, welche im Rahmen des Spielfelds liegt. Anschließend muss noch überprüft werden, ob diese Position nicht durch andere Hindernisse auf dem Spielfeld oder durch ein Segment des Schlangenkörpers bereits besetzt ist. Für den Abgleich sorgt die Methode *IsPositionOccupied()*. Hierfür wird in einer *do-while* Schleife so lange eine neue Position generiert, bis die Methode *false* zurückgibt, also die Position nicht besetzt ist. Im Spiel überprüft die Logik der Schlange, ob eine Kollision mit einer Frucht stattgefunden hat. Dies ist in Listing 6 zu sehen.

```
protected virtual void CheckFruitCollision()
{
    if (_body.Points[0] == _fruit.Position)
    {
        _tween.StopAll();
        _audioPlayer.Play();
        _eating = true;
        _fruit.SetNewPosition(_fruit.RandomizePosition());
        MoveSnake();
    }
}
```

Listing 6: Überprüfung auf Fruchtkollision

Dabei wird überprüft, ob die Position des Schlangenkopfes mit der Position der Frucht übereinstimmt. Ist dies der Fall, wird die Variable *_eating* auf *true* gesetzt, was dafür sorgt, dass beim nächsten Bewegungsvorgang dem Körper ein Segment hinzugefügt wird. Anschließend wird eine neue Position für die Frucht über das bereits erklärte Schema festgelegt.

4.2.4 Hinzufügen eines Punktezählers

Der Punktezähler ist eine zusätzliche Funktion, welche während der Entwicklung des Spiels aufkam. Er dient als Messlatte für das Können des Spielers sowie als Motivation, so lange wie möglich im Spiel durchzuhalten und so viele Früchte wie möglich zu fressen. In der Level-Szene wird er rechts neben dem Spielfeld dargestellt und zeigt den aktuellen Punktestand sowie die höchste Punktzahl für das Level an (siehe Abbildung 11).

Abbildung 11: Punktezähler in einem Spiel

Für die Umsetzung des Punktezählers wurden zwei neue Klassen erstellt, *HighScores* und *HighScoreManager* (siehe Anhang 8). Erstere enthält ein *Dictionary,* bei welchem der Schlüssel ein *string* und der dazugehörige Wert vom Typ *int* ist. Der Schlüssel stellt dabei den Namen des jeweiligen Levels und der Wert den Punktestand dar. Die Klasse *HighScoreManager* regelt das Abrufen, Ändern sowie Speichern des Punktestands. Zum Speichern wird dabei eine JSON[36]-Datei genutzt, welche im Godot-spezifischen Benutzerverzeichnis des Spiels gespeichert wird.

Zu Beginn einer Runde wird durch den Konstruktor der Klasse die Methode *LoadHighScores()* aufgerufen. Diese überprüft, ob die Datei *highscores.json* vorhanden ist, also ob schon einmal Punktestände erfasst und gespeichert wurden. Ist dies der Fall, wird der Inhalt der Datei abgerufen und in einem Objekt der *HighScores*-Klasse gespeichert. Ist die Datei nicht vorhanden, wird sie mit einem leeren Dictionary erstellt. Anschließend wird mit der Methode *GetHighScore()* die gespeicherte Höchstpunktzahl für das jeweilige Level aus dem Dictionary abgerufen und in der Level-Szene angezeigt. Wenn im Verlaufe des Spiels eine Frucht eingesammelt wird, wird dies in der *GameController.cs*-Datei, welche sämtliche Abläufe im Spiel steuert, über die Variable *_score* festgehalten. Ist das Spiel vorbei, wird die Methode *SetHighScore()* mit dem Levelnamen und Punktestand als Parameter aufgerufen und falls die Punktzahl höher als das bisherige Maximum ist, in der JSON-Datei gespeichert.

[36] Abkürzung für JavaScript Object Notation

4.2.5 Variierung der Schwierigkeit, Level und Spielmodi

Um mehr Abwechslung für den Spieler zu schaffen, wurden wie bereits erwähnt, mehrere Schwierigkeitsgrade, Level und Spielmodi in die Anwendung eingebaut. Die unterschiedlichen Schwierigkeiten sind „Leicht“, „Mittel“, „Schwer“ und „Profi“. Diese unterscheiden sich durch die Geschwindigkeit, mit welcher sich die Schlange fortbewegt. Genauer wird die Zeitspanne variiert, in welcher die *MoveTween()*-Methode ausgeführt wird. Höhere Werte führen dazu, dass die Schleife länger benötigt, um einen Bewegungszyklus abzuschließen, womit sich die Schlange langsamer bewegt. Niedrigere Werte hingegen bewirken, dass die Methode schneller und somit insgesamt häufiger ausgeführt wird. Die Auswahl des Schwierigkeitsgrads erfolgt in der Anwendung über einen *OptionSelection*-Node (siehe Anhang 2), bei welchem eine der vier Möglichkeiten ausgewählt werden kann. Der entsprechende Index wird dann im *GameController* abgefragt und der Wert für die Dauer der *MoveTween* Methode gesetzt. Das folgende Listing zeigt Ausschnitte aus den Dateien *Einstellungen.cs* und *GameController.cs*, welche die Logik für die Levelauswahl-Szene bzw. die eigentliche Level-Szene beinhalten.

```
//Ausschnitt aus Einstellungen.cs
private OptionSelection _SelectDifficulty = new OptionSelection
         (4, new string[] { "Leicht", "Mittel", "Schwer", "Profi" });
GlobalVariables.Instance.LevelDifficulty = _SelectDifficulty.SelectedOption;

//Ausschnitt aus GameController.cs
switch (GlobalVariables.Instance.LevelDifficulty)
        {
            case 0:
            {
                // einfach
                _snake1.moveDelay = _snake2.moveDelay = 0.3f;
                _multiplayerSnake.moveDelay = 0.4f;
                break;
            }
            case 1:
            {
                // mittel
                _snake1.moveDelay = _snake2.moveDelay =  0.2f;
                    _multiplayerSnake.moveDelay = 0.35f;
                break;
            }
       …
```

Listing 7: Code zum Einstellen der Schwierigkeit

Zur Variierung der spielbaren Level wurden neben dem bereits existierenden Level ohne Hindernisse noch zwei weitere erstellt. Diese besitzen zusätzliche Objekte auf dem Spielfeld, um welche der Spieler die Schlange navigieren muss (siehe Anhang 9). Umgesetzt wird dies durch Abfragen des ausgewählten Levels in der *GameController*-Klasse, ähnlich wie beim Schwierigkeitsgrad.

Abhängig davon wird dem Spielfeld-Array ein anderer Wert zugewiesen, wobei an Stellen mit zusätzlichen Hindernissen eine 1 anstatt einer 0 gesetzt wurde. So wurden insgesamt drei verschiedene Szenen für die unterschiedlichen Spielfelder erstellt, wobei das Aussehen jeweils durch das Hinzufügen von Mauer-Kacheln an den Stellen der Hindernisse angepasst wurde, sodass es mit den Einstellungen im Code übereinstimmt. Die Überprüfung auf Kollisionen mit den Hindernissen musste dabei nicht angepasst werden, da diese schon die Felder überprüft, welche den Wert 1 haben.

Neben verschiedenen Schwierigkeitsgraden und Spielfeldern stand die Einführung mehrerer Spielmodi im Vordergrund der Entwicklung. Eine Idee für den Mehrspieler war ein kompetitiver Modus, bei welchem zwei Spieler gegeneinander antreten, Früchte aufnehmen und vermeiden, die gegnerische Schlange zu berühren. Hierzu musste die Logik der Schlange angepasst werden, sodass sie in der Lage ist, auf eine zweite Schlange im Spiel zu reagieren um z.B. Eingaben für die Bewegung zu unterscheiden. Hierfür wurde eine zusätzliche boolesche Variable *_isPlayerOne* und eine zweite Schlangeninstanz *_otherSnake* eingeführt. Beide Werte werden mithilfe der Methode *SetPlayerSettings()* initialisiert, welche den Wert für *_isPlayerOne* übergibt und abhängig davon die entsprechende Schlangen-Szene referenziert. Die Methode wird im folgenden Listing dargestellt.

```
public override void SetPlayerSettings(bool isPlayerOne)
    {
        _isPlayerOne = isPlayerOne;
        BaseSnake s = null;
        if (!isPlayerOne)
        {
            if (GetParent() != null)
                s = GetParent().GetNodeOrNull<BaseSnake>("Snake1");
            if (s != null)
                _otherSnake = s;
            else
                _otherSnake = null;

            _body.DefaultColor = new Color(255, 255, 0, 1);
            for (int i = 0; i < _points.Count; i++)
            {
                _points[i] += new Vector2(0, 2 * _gridSize);
                _body.SetPointPosition(i, _points[i]);
            }
        }
        else
        {
            _otherSnake = GetParent().GetNodeOrNull<BaseSnake>("Snake2");
        }
    }
```

Listing 8: Methode zum Bestimmen der Spielerreihenfolge

Darüber hinaus erhält die Schlange von Spieler 2 eine andere Farbe und wird zwei Felder unter Spieler 1 platziert, damit die beiden Schlangen nicht zu Beginn miteinander kollidieren.

Die Boolesche Variable steuert zudem noch, ob die Bewegung der Schlange über die Tasten W, A, S, D oder über die Pfeiltasten erfolgt. Die Referenzierung der zweiten Schlange ist essenziell für die gegenseitige Kollisionserkennung. Dabei wird überprüft, ob die Schlangen mit ihren beiden Köpfen zusammengestoßen sind oder ob eine Schlange mit dem Körper der anderen Schlange kollidiert. Stoßen beide Schlangen mit ihren Köpfen zusammen, wird dies als Unentschieden gewertet, ansonsten verliert der Spieler, welcher mit der anderen Schlange kollidiert. Listing 5 zeigt einen Ausschnitt aus der Methode *IsGameOver()*, welche die Kollisionen überprüft.

```
if (_otherSnake != null && IsInstanceValid(_otherSnake))
        {
            if (_otherSnake.Points.Contains(_body.Points[0]))
            {
                if (_body.Points[0] == _otherSnake.Points[0])
                {
                    _controller.LoseMessage = ($"Unentschieden.\n{Name} und
                    {_otherSnake.Name} sind kollidiert.");
                    return true;
                }
                else
                {
                    _controller.LoseMessage = ($"Game Over fuer {Name}.\nIst mit
                    {_otherSnake.Name} kollidiert!");
                    return true;
                }
            }
        }
```

Listing 9: Überprüfung der Kollision mit einer anderen Schlange

Zusätzlich zum kompetitiven Spielmodus wurde ein kooperativer Spielmodus entwickelt. Dieser basiert auf dem Prinzip, dass eine Schlange abwechselnd von beiden Spielern gesteuert wird (siehe Anhang 10). Die Steuerung wechselt, wenn eine Frucht eingesammelt wird. Zusätzlich bewegt sich die Schlange beim Wechsel in die entgegengesetzte Richtung. Dies soll die Herausforderung beim Spielen erhöhen und ist besonders auf Spielfeldern mit zusätzlichen Hindernissen knifflig.

Um diese Idee umzusetzen, wurde eine neue Kindklasse, welche vom Basistypen *BaseSnake* abgeleitet wird, erstellt. Diese trägt den Namen *OfflineMultiplayerSnake.* Für die zweiköpfige Schlange musste die Logik zur Bewegung, Kollisionsüberprüfung sowie Richtungsbestimmung Anpassungen unterzogen werden. Hierzu wurde zuerst eine zusätzliche Variable der Klasse *Vector2* erstellt, welche die vorgesehene Bewegungsrichtung der Schlange für den zweiten Spieler speichert, sowie die boolesche Variable *_isPlayerOneTurn*, welche aussagt, ob Spieler 1 oder 2 am Zug ist. Die Anpassung der Bewegung ist im folgenden Listing dargestellt. Dafür wurde die Methode *MoveTween()* der Basisklasse überschrieben.

```
_currentDirection = _isPlayerOneTurn ? _directionCachePlayer1 :
_directionCachePlayer2;

int headIndex = _isPlayerOneTurn ? 0 : _body.Points.Count() - 1;
int direction = _isPlayerOneTurn ? 1 : -1;

for (int i = _isPlayerOneTurn ? 0 : _body.Points.Count() - 1;
     _isPlayerOneTurn ? i < _body.Points.Count() : i >= 0;
     i += direction)
{
    Vector2 newPos;

    if (i == headIndex)
    {
        newPos = _points[i] + _currentDirection * _gridSize * argv;
    }
    else
    {
        int prevIndex = i - direction;
        Vector2 diff = (_points[prevIndex] - _points[i]) / _gridSize;
        newPos = _points[i] + diff * _gridSize * argv;
    }

    _body.SetPointPosition(i, newPos);
}
```

Listing 10: Fortbewegung der Mehrspieler-Schlange

Hier werden zuerst die aktuelle Bewegungsrichtung der Schlange sowie der Index der Kopfposition im Array der Körperteile ermittelt. Anschließend werden mittels einer *for-Schleife*, welche durch die Punkte des Körpers iteriert, die neuen Positionen für jeden Körperteil bestimmt und zugewiesen. Für die Überprüfung einer Fruchtkollision wird ebenfalls abhängig davon, ob Spieler 1 oder 2 an der Reihe ist, der jeweilige Index im Körperteil-Array mit der Fruchtposition verglichen.

4.3 Umsetzung der Lobby als WebSocket System

Um eine WebRTC-Sitzung aufbauen zu können, müssen die dazugehörigen Verbindungsdaten ausgetauscht werden. Hierfür werden WebSockets verwendet. Da externe Server laufende Kosten verursachen, muss einer der Spieler einen eigenen WebSocket-Server einrichten. Alle anderen können sich dann als Clients über dessen IP-Adresse und den zugehörigen Port auf ihn verbinden. Das Gerät, auf dem der Server läuft, sollte nach Möglichkeit eine öffentliche IP-Adresse besitzen. Im Gegensatz zu WebRTC bieten WebSockets keine eingebaute Möglichkeit, Verbindungsblockaden durch unterschiedliche NAT-Netzwerke oder Firewalls zu umgehen. Durch beispielsweise eine Portfreigabe am Router kann festgelegt werden, über welche Ports ein Gerät im privaten Netzwerk für externe Teilnehmer erreichbar ist. Sollte dies nicht möglich sein, bietet das Spiel eine Option, die WebRTC-Daten über ein anderes Medium, wie z. B. E-Mails, auszutauschen. Hierfür werden die Verbindungsinformationen ausgegeben. Der Benutzer muss sie an seinen Partner übermitteln. Dieser speichert die Daten und generiert daraufhin seine eigenen Verbindungsinformationen, welche er wiederum über ein externes Medium an seinen

Mitspieler weitergeben muss. Geräte im gleichen Netzwerk können den Server auch ohne spezielle Freigaben erreichen. Um die Last des Geräts, welches den WebSocket-Server betreibt, zu minimieren, werden nur die SDP-Verbindungsdaten und ICE-Kandidaten über den Server geleitet. Nachdem eine WebRTC-Sitzung erfolgreich initiiert wurde, trennen sich beide Spieler von diesem. Die eigentlichen Spielübertragungsinformationen werden dann nur zwischen den zwei Spielteilnehmern ausgetauscht.

Der Server kann gestartet werden, indem der Benutzer im Mehrspielermenü entweder *Server starten* oder *Server und Client starten* auswählt. In der ersten Variante wird nur ein WebSocket-Server unter einem angegebenen Port gestartet. Das bedeutet, andere Spieler können über diesen Server Spielsitzungen starten, man selbst jedoch nicht. Dies ist in der zweiten Variante möglich, bei der zusätzlich ein Client erzeugt wird, der sich mit dem Server verbindet. Dadurch hat man selbst die Möglichkeit, am Spielgeschehen teilzunehmen. Der Server bleibt im Hintergrund geöffnet. Nach einem erfolgreichen Verbindungsaufbau gelangt man in eine *Lobby*. Hier gibt es einen Textchat, über den alle Spieler miteinander kommunizieren können. Zudem besteht die Möglichkeit, Räume zu erstellen oder ihnen beizutreten. Wenn zwei Spieler einem Raum beigetreten sind, werden die WebRTC-Verbindungsdaten zwischen ihnen ausgetauscht. Sobald diese Verbindung erfolgreich initiiert wurde, wird der WebSocket-Client geschlossen und das Spiel startet.

Anhang 11 veranschaulicht die Kommunikation zwischen den Clients und dem Host. Zuerst sendet der Client eine Verbindungsanfrage. Ist diese erfolgreich, erzeugt der Server eine eindeutige Zahl (ID) und sendet sie an den Client zurück. Daraufhin antwortet der Client mit dem Spielernamen, unter dem der Benutzer im Textchat Nachrichten veröffentlichen kann. Der Server speichert die ID und den Namen des Clients. Alle Nachrichten folgen dem in Listing 11 gezeigten Aufbau.

```
class msg
{
   public Nachricht state;
   public int publisher;
   public int target;
   public string data;
   public msg(Nachricht state,int publisherid, int target, string data)
   {
       this.state = state;
       publisher = publisherid;
       this.target = target;
       this.data = data;
    }
}
```

Listing 11: Aufbau einer WebSocket Nachricht

Die Eigenschaft *state* ist eine Enumeration, die festlegt, wie die nachfolgenden Daten in *data* zu interpretieren sind. Informationen mit dem Status *chatMSG* werden als Textnachrichten interpretiert und an alle Empfänger weitergeleitet. Die Variable *target* spezifiziert, an welche Clients die Informationen gesendet werden. So tauschen sich beispielsweise nur die Raumpartner ihre SDP-Daten untereinander aus. In den meisten Fällen ergibt sich bereits aus dem *state,* an wen die Nachricht gerichtet werden muss.

Der Großteil des Kommunikationsablaufs ähnelt dem Austausch von Textnachrichten. Ein Client sendet eine Nachricht an den Server, dieser interpretiert sie und leitet die Information an den Zielteilnehmer weiter. Möchte beispielsweise ein Spieler einen Raum eröffnen, sendet er eine Anfrage an den Server. Dieser fügt den Raum zu den aktuell vorhandenen hinzu und sendet eine Liste aller Räume an alle Clients. Jetzt sieht jeder den neuen Raum, dem beigetreten werden kann. Dabei ist zu beachten, dass der Kommunikationsverlauf nicht blockierend ist. Das bedeutet, während eine Raumanfrage bearbeitet wird, können andere Teilnehmer ebenfalls Anfragen stellen und verschiedene Nachrichtenarten austauschen. Keiner der Teilnehmer muss aktiv auf eine Antwort warten.

Eine Besonderheit ist der Austausch der WebRTC-Verbindungsdaten. Wenn zwei Spieler in einem Raum sind, erhält derjenige, der den Raum ursprünglich erstellt hat, die Initiative, ein SDP-Angebot zu erstellen. Sendet er dieses, übermittelt er die SDP-Informationen mit der *target-ID* seines Raumpartners an den Server. Dieser fungiert nun als Relay-Server und leitet die Informationen an den Client mit der entsprechenden *target-ID* weiter. Dieser erstellt eine Antwort und gibt als Ziel ebenfalls seinen Raumpartner an. Der Server erhält diese Nachricht und leitet sie weiter. Ähnlich funktioniert es mit den ICE-Kandidaten. Diese können jedoch nach dem SDP-Austausch zeitgleich von beiden Teilnehmern erstellt und an den Host gesendet werden. Nachdem die RTC-Verbindung hergestellt ist, trennen sich beide Spieler vom Server. Dieser wartet, bis beide den erfolgreichen Verbindungsaufbau bestätigen. Anschließend werden beide aus der Clientliste entfernt. Den Nachrichtenaustausch kann man im Anhang 11 gezeigten Sequenzdiagramm nochmal nachvollziehen.

4.4 Umsetzung der Remote Procedure Calls

In der Anwendung werden RPC ausschließlich über die WebRTC Verbindung ausgeführt und über die Klasse *NetworkManager* geregelt. Eine Nachricht ist aus einem Status und JSON-String zusammengesetzt. Die Daten werden abhängig vom Status interpretiert. In Listing 12 ist die Funktion abgebildet. Sie wird auf beiden Geräten aufgerufen. Um RPC auszuführen, wird auf die Methoden *GetNode()* und *Call()* zurückgegriffen. *GetNode()* ermittelt die Referenz eines Knotens aufgrund seines Pfades. Dieser kann mit *GetPath()* bestimmt werden. Weiterhin legt *Call()* die

Methode, welche von dem Knoten aufgerufen werden soll, fest. *remoterpc* sollte bei dem Aufrufer immer *false* sein. Der Empfänger extrahiert die Nachricht und ruft dann die Funktion mit *remoterpc = false* auf. Würde er es nicht tun, sendet dieser wiederum an den eigentlichen Sender einen *Remote Procedure Call* und eine Endlosschleife entsteht. Durch das Setzen von *dolocal* auf *false* kann erreicht werden, dass der Funktionsaufruf nur auf dem entfernten Gerät ausgeführt wird.

```
public void rpc(string NodePath, string Method, bool remoterpc = false,
bool dolocal = true, bool relaible = true, params object[] Args)
{
  if(dolocal == true)
  {
    // lokal den Rpc aufühen
    try
    {
      GetNode(NodePath).Call(Method,Args);
    }
    catch(Exception e)
    {
      GD.Print("Der Pfad: " + NodePath + " oder die Methode: " + Method +
               " existiert nicht!",e);
    }
  }
  if(remoterpc == false && _multiplayerIsActive == true)
  {
    var settings = new JsonSerializerSettings
    {
      ReferenceLoopHandling = ReferenceLoopHandling.Ignore
    };
    if(relaible == true)
      _multiplayer.TransferMode =
      WebRTCMultiplayer.TransferModeEnum.Reliable;
    else
      _multiplayer.TransferMode =
      WebRTCMultiplayer.TransferModeEnum.UnreliableOrdered;
    _RtcMsg msg = new _RtcMsg(_RtcMsgState.RPC,NodePath + "|" + Method +
                  "|" + JsonConvert.SerializeObject(Args,settings));
    SendRawMessage(_RtcMsg.ConvertToJson(msg).ToUTF8());
  }
}
```

Listing 12: RPC Funktion

Die WebRTC Nachrichten können bestätigt oder unbestätigt versendet werden. Um zu garantieren, dass eine Nachricht ankommt, ist *reliable* auf *true* zu setzen. Diese Informationen benötigen durch eine Bestätigung eine längere Übertragungszeit. Sollte eine Methode Übergabeparameter benötigen, können diese durch *params* angegeben werden.

Ein Beispiel für einen Funktionsaufruf ist in Listing 15 zu sehen. Hier sendet der Server ein Update der Schlangenpositionen an den Client. Die Methode soll nicht lokal ausgeführt werden, da der Server kein Update benötigt. Da die Nachrichten in kurzen Zeitabständen gesendet werden, können sie unbestätigt gesendet werden. Die Positionen werden als JSON-Strings konvertiert und an den Client gesendet.

Listing 13 zeigt einen Ausschnitt der Interpretation von WebRTC-Nachrichten. Die *_Process* Funktion wird innerhalb eines Programmzyklus aufgerufen. Darin wird geprüft, ob eine neue Nachricht empfangen wurde. Besitzt sie den Status *RPC* müssen die Daten extrahiert werden. Hier ist zu sehen, dass *remoterpc*, der dritte Übergabeparameter, immer *true* ist. So wird an den anderen Teilnehmer kein neuer Aufruf versendet.

```
public override void _Process(float delta)
{
  if(_multiplayerIsActive == true)
  {
    // Auf Nachrichten hören und diese interpretieren!
    _multiplayer.Poll();
    if(_multiplayer.GetAvailablePacketCount() > 0)
    {
      string strMsg = _multiplayer.GetPacket().GetStringFromUTF8();
      _RtcMsg data = _RtcMsg.ConvertTo_RtcMsg(strMsg);
      if(data != null)
      {
        switch(data.MsgState)
        {
          case _RtcMsgState.RPC:
          {
            //0. = NodePath, 1. = Method, wenn mehr: 3. = Args
            string[] msg = data.Data.Split("|");
            if (msg.Length >= 3)
            {
              rpc(msg[0], msg[1], true, true, true,
                        JsonConvert.DeserializeObject<object[]>(msg[2]));
            }
            else
            {
              rpc(msg[0], msg[1], true);
            }
            break;
        }
```

Listing 13: Empfangen von WebRTC Nachrichten

4.5 Das Peer-To-Peer System mit WebRTC

Die Spiellogik für den internetbasierten Mehrspielermodus verfolgt ein Server-Client Modell. Nachdem zwischen den zwei Teilnehmern eine WebRTC Verbindung aufgebaut wurde, gelangen sie in ein Einstellungsmenü. Hier können, wie im Einzelspielermodus, Schwierigkeitsgrad, Level und Spielmodi festgelegt werden. Das Menü ist so gestaltet, dass jeder Spieler die Entscheidungen des anderen durch einen grünen Haken neben den Optionen sehen kann. Sobald Beide ihre Einstellungen vorgenommen haben, bestätigen sie mit *Bereit*. Jetzt wartet der Spieler, bis sein Mitstreiter ebenfalls bereit ist. Wenn beide Spieler unterschiedliche Einstellungen

gewählt haben, entscheidet das System zufällig und startet das Spiel basierend auf dieser Auswahl.

Der Benutzer, welcher zuerst fertig ist, wird zu Spieler 1 und steuert die grüne Schlange. Er ist zugleich der Server und übernimmt die Spielverwaltung. Wie in Kapitel 3.3 beschrieben, muss er die Schlange bewegen. Den in Abbildung 6 gezeigten, linken Programmstrang übernimmt die Tween Schleife. Sie wird wie im Einzelspieler durchlaufen. Nachdem die Schlange ein Feld weitergerückt ist, überprüft das System, ob eine Frucht gefressen wurde oder eine tödliche Kollision aufgetreten ist. Hier existiert ein Unterschied zum Einzelspieler. Der Client wird über die Ereignisse mittels der in Kapitel 4.4 besprochenen RPC-Aufrufe benachrichtigt. So bekommt er mit, wenn eine Frucht gegessen wurde. Jetzt spielt das System Fressgeräusche ab und platziert die Frucht an einer neuen, vom Server bestimmten Position. Die Funktion in Listing setzt den rechten Programmablauf um. Die *_Process* Methode wird circa alle 100 ms aufgerufen und prüft, ob der Client ein Update benötigt. Die Funktion *Time.GetTicksUsec()* gibt die Zeit in Mikrosekunden zurück, die seit dem Anwendungsbeginn vergangen ist. Mit einem festgelegten *_updateInterval* von 100.000 Mikrosekunden sendet Spieler 1 alle 100 ms ein Update an Spieler 2.

```
public override void _Process(float delta)
{
    if (_isServer)
    {
        if (Time.GetTicksUsec() - _TimeSinceLastUpdate > _updateInterval)
        {
            _TimeSinceLastUpdate = Time.GetTicksUsec();
            TimeToSynchBodyPoints();
        }
    }
}
```

Listing 14: Methode zum zyklischen Senden der Weltveränderungen

Änderungen in der Spielewelt beschränken sich in diesem Anwendungsfall auf die einzelnen Positionen der Schlangensegmente. Wie in Listing 15 gezeigt, werden zuerst die Positionen der einzelnen Segmente in Arrays gespeichert. Danach werden sie als JSON-Objekte mit einem Zeitstempel über einen RPC-Aufruf an den Client gesendet. Am zweiten *false*-Parameter erkennt man, dass dieser Aufruf nicht lokal ausgeführt wird. Nur beim Spieler 2 wird die Methode *SyncBodyPointsOnClient()*, welche in Listing 16 dargestellt ist, aufgerufen und ausgeführt.

```
protected void TimeToSynchBodyPoints()
{
    float[] x = new float[_body.GetPointCount()];
    float[] y = new float[_body.GetPointCount()];
    for (int j = 0; j < _body.GetPointCount(); j++)
    {
        x[j] = _body.GetPointPosition(j).x;
        y[j] = _body.GetPointPosition(j).y;
    }
    NetworkManager.NetMan.rpc(GetPath(), nameof(SynchBodyPointsOnClient), false,
    false, false, JsonConvert.SerializeObject(x), JsonConvert.SerializeObject(y),
    Time.GetTicksUsec());
}
```

Listing 15: Server sendet Weltenupdate an Client

```
protected void SynchBodyPointsOnClient(string Xjson, string Yjson, UInt64 SendTime)
{
    float[] x = JsonConvert.DeserializeObject<float[]>(Xjson);
    float[] y = JsonConvert.DeserializeObject<float[]>(Yjson);
    _TargetPoints.Clear();

    for (int i = 0; i < x.Length; i++)
    {
        _TargetPoints.Add(new Vector2(x[i], y[i]));
    }
    _ClientTimeDiffBodyUpdate = SendTime - _ClientTimeAtBodyPointUpdate;
    _ClientTimeAtBodyPointUpdate = SendTime;
    if (_ClientTimeDiffBodyUpdate < _updateInterval)
        _ClientTimeDiffBodyUpdate = _updateInterval;

    // Umwandlung von Microsekunden in Millisekunden
    _ClientTimeDiffBodyUpdate /= 1000;
    _CalculateNewLatenyFactor = true;
}
```

Listing 16: Client empfängt Positionsupdate

Der Client erhält durch die Interpretation der JSON-Objekte alle neuen Punkte der Schlange. Die Konzeption sieht vor, dass der Übergang zwischen den angezeigten und den neu empfangenen Schlangenpositionen fließend sein muss. Dafür wird die Zeit zwischen den Updates durch den gesendeten Zeitstempel berechnet. Die Variable *ClientTimeAtBodyPointUpdate* enthält den letzten empfangenen Zeitwert. Durch eine Differenzbildung der beiden Parameter kann die Zeit, welche zwischen zwei Updates vergangen ist, ermittelt und unter *_ClientTimeDiffBodyUpdate* gespeichert werden. Um einen flüssigen Übergang zwischen den Positionen zu erzeugen, wird die *_PhysicsProcess*-Methode (Anhang 12) angepasst. Zuerst wird die Wegdifferenz zwischen den zwei Welten berechnet. Es reicht, die Positionsdifferenz für das erste Segment (Index 0) zu berechnen. Alle anderen Segmente besitzen einen konstanten Abstand zueinander und müssen dadurch denselben Weg überwinden.

Die Distanzberechnung kann auf eine Achse reduziert werden, da die Schlange nie schräg übers Spielfeld laufen kann. Nachdem die Distanz bekannt ist, kann berechnet werden, wie viel Prozent von dieser in einem Schleifenzyklus zurückzulegen ist. Hierfür wird die Zeit zwischen den Schleifenaufrufen, genannt *delta,* mit der Zeit zwischen zwei Updates (*_ClientTimeDiffBodyUpdate*) dividiert und unter der Variable

latencyFactor gespeichert. Diese Prozedur wird nur einmal pro Updatezyklus ausgeführt. Würde sie in jedem Zyklus berechnet, würde die Position sich dem Ziel nur annähern, es aber nie erreichen, da *diff* und der damit zusammenhängende latencyFactor gegen Null tendieren. Nachdem der *latencyFactor* berechnet ist, findet die Positionsannäherung statt. Hierfür wird in einer for Schleife die Distanz zwischen den neuen und den alten Punkten erneut ermittelt und in *DiffVec* gespeichert. Bei dieser Differenzberechnung bleibt die Richtung im Gegensatz zur vorherigen Berechnung erhalten. Anschließend wird diese Differenz mit dem *latencyFactor* multipliziert und auf die aktuelle Segmentposition addiert. So erhält man eine flüssige Bewegung ohne stehende Bilder. Um Netzwerkschwankungen auszugleichen, berechnet die Methode *MakeAverageLatenyFactor()* einen Durchschnittswert der Variable *latencyFactor*. Dabei werden die letzten 45 berechneten Werte einbezogen.

Bei dem Spielmodus *Gegeneinander* ist zu beachten, dass der erläuterte Mechanismus bei beiden Schlangen parallel abläuft. Das bedeutet, beide werden vom Server simuliert, aber der Serverspieler kann nur seine Schlange steuern. Der Client sendet seine Tasteneingaben an ihn. Dieser berücksichtigt die Richtungsänderungen bei seinem nächsten Tweendurchlauf und setzt diese dann in die gewünschte Richtung. Die Steuerung der Schlangen im Onlinemodus entspricht der des Einzelspielers. Das heißt beide Spieler können mit den Tasten W, A, S und D die jeweilige Richtung bestimmen.

5 Fazit

Zusammenfassend lässt sich sagen, dass das Ziel der Anwendung erfolgreich umgesetzt wurde. Mit der Videospielentwicklungsumgebung Godot wurde eine Anwendung erstellt, mit welcher das klassische Videospiel Snake sowohl im Offline- als auch Online-Modus gespielt werden kann. Dabei wurden zur Abwechslung und Herausforderung für den Spieler mehrere Schwierigkeitsgrade, Spielfelder und Spielmodi eingebaut.

Bei der Entwicklung des Online-Mehrspielers traten mehrere Probleme auf, welche den Fortschritt behinderten und zeitaufwendige Lösungen benötigten. So verursachten die eingebauten *Remote Procedure Calls*, über welche Godot verfügt, Komplikationen, da diese nicht ordnungsgemäß funktionierten. Dies führte dazu, dass eigene RPC-Methoden programmiert werden mussten, welche diese Funktionalitäten ersetzten. Trotz dessen konnte der Online-Mehrspieler-Modus über zwei verschiedene Funktionsweisen umgesetzt werden.

Zum einen wurde eine Variante mit Websockets erstellt, bei welcher ein Teilnehmer eine Lobby erstellt, der andere Spieler beitreten können. Dort können jeweils zwei Spieler ein gemeinsames Spiel starten und Textnachrichten austauschen. Zudem gibt es die Möglichkeit, dass sich beide Spieler über einen Sprachchat unterhalten können. Befinden sich die Teilnehmer in verschiedenen Netzwerken, muss der Ersteller der Lobby eine Portfreigabe vornehmen, damit andere Personen beitreten können. Die zweite Möglichkeit für den Online-Mehrspieler-Modus wurde mittels eines Peer-To-Peer System unter Verwendung von WebRTC erstellt. Bei diesem müssen die beiden Teilnehmer ihre Verbindungsdaten untereinander austauschen. Wie bei sämtlichen anderen Online-Videospielen musste hier stets auf die Häufigkeit und Menge der übertragenen Daten geachtet werden, damit es nicht zur Überlastung der Verbindung kommt und es zu keiner Verzögerung im Spiel kommt.

Literaturverzeichnis

ATAUL, Mukit: WebRTC SDP (Session Description Protocol) Details, 2018. In: https://dev.to/lucpattyn/webrtc-sdp-session-description-protocol-details--5593#RFC4566 (05.11.2024)

CHRIS: WebRTC - eine Übersicht - Chris Blog, 2020. In: https://chris-blog.com/2020/08/22/webrtc-eine-uebersicht/ (05.11.2024)

COMPUTERWEEKLY.DE: Was ist WebRTC (Web Real-Time Communications)? - Definition von Computer Weekly, 2024. In: https://www.computerweekly.com/de/definition/WebRTC-Web-Real-Time-Communications (05.11.2024)

CURRIER, Nathaniel: ICE and WebRTC: What Is This Sorcery? We Explain…, 2022. In: https://temasys.io/guides/developers/webrtc-ice-sorcery/ (05.11.2024)

DIGITAL SAMBA: Deciphering SDP: An In-Depth Exploration of WebRTC's Session Description Protocol, 2023. In: https://medium.com/@digital_samba/deciphering-sdp-an-in-depth-exploration-of-webrtcs-session-description-protocol-b5dc0fca71a9 (05.11.2024)

GISBERTZ, Jan: Was ist WebRTC und wie funktioniert die Echtzeitkommunikation über den Browser?, 2023. In: https://www.tk-gisbertz.de/news/digitalisierung-loesungen/was-ist-webrtc-und-wie-funktioniert-die-echtzeitkommunikation-ueber-den-browser/ (17.10.2024)

GODOT ENGINE DOCUMENTATION: Introduction to Godot, 2024a. In: https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html (30.09.2024)

GODOT ENGINE DOCUMENTATION: Overview of Godot's key concepts, 2024b. In: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html (30.09.2024)

GORSKI, Peter L.: WebSockets – Moderne HTML5-Echtzeitanwendungen entwickeln: Hanser eLibrary. 1. Aufl. München, 2015

ICHI.PRO: Herstellen einer WebRTC-Verbindung: Videoanruf mit WebRTC Schritt 3, 2024. In: https://ichi.pro/de/herstellen-einer-webrtc-verbindung-videoanruf-mit-webrtc-schritt-3-80078290460324 (05.11.2024)

IONOS DIGITAL GUIDE: Was ist WebSocket?, 2024. In: https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-websocket/ (01.10.2024)

IONOS REDAKTION: Remote Procedure Call (RPC) – effiziente Kommunikation in Client-Server-Architekturen. In: IONOS, 10.03.2020. In: https://www.ionos.de/digitalguide/server/knowhow/was-ist-ein-remote-procedure-call/

LUBER, Stefan; TUTANCH: Was ist DTLS (Datagram Transport Layer Security)?, 2020. In: https://www.ip-insider.de/was-ist-dtls-datagram-transport-layer-security-a-903b5df4f4c79da30d226dbf1b1feb67/ (05.11.2024)

LUECKE, David: HTTP vs Websockets: A performance comparison - The Feathers Flightpath. In: The Feathers Flightpath, 27.01.2018. In: https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77

MASCELLINO, Alessandro: 70+ Videospielstatistiken 2024: Marktwachstum, aktuelle Trends und mehr. In: Techopedia, 02.04.2024. In: https://www.techopedia.com/de/videospielstatistiken

READ-MCFARLAND, Austen: Sicherheit in WebRTC: Sichere Kommunikation im Browser, 2023. In: https://blog.wildix.com/de/sicherheit-in-webtc/ (05.11.2024)

Source Multiplayer Networking - Valve Developer Community, 2024. In: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (01.11.2024)

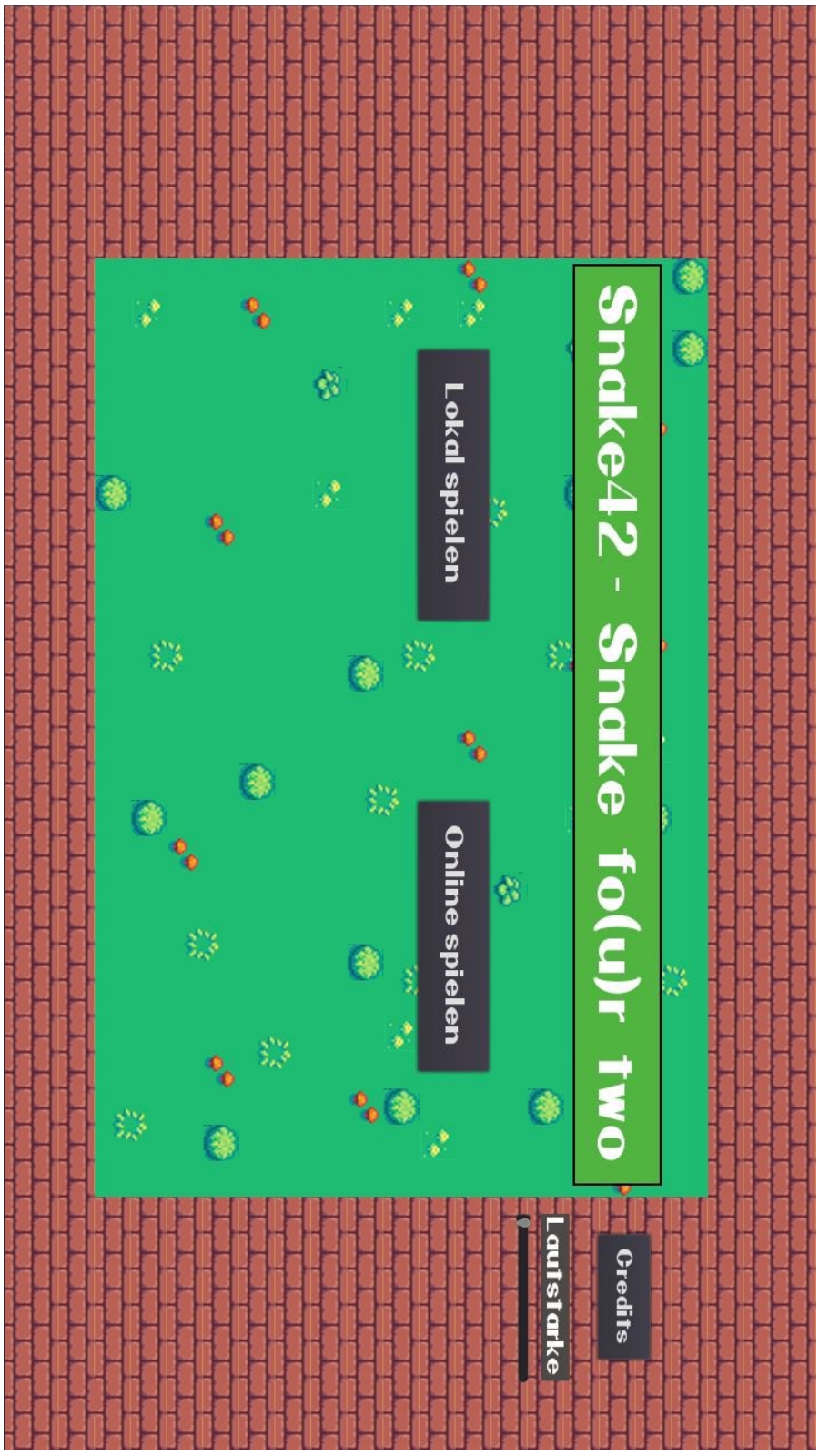
The history of Snake: How the Nokia game defined a new era for the mobile industry, 2024. In: https://www.itsnicethat.com/features/taneli-armanto-the-history-of-snake-design-legacies-230221 (26.09.2024)

Tweening: Definition, Geschichte und Anwendung. | Adobe, 2024. In: https://www.adobe.com/de/creativecloud/animation/discover/tweening.html?msockid=2c3da908e79668833a27bb54e61d692e (28.10.2024)

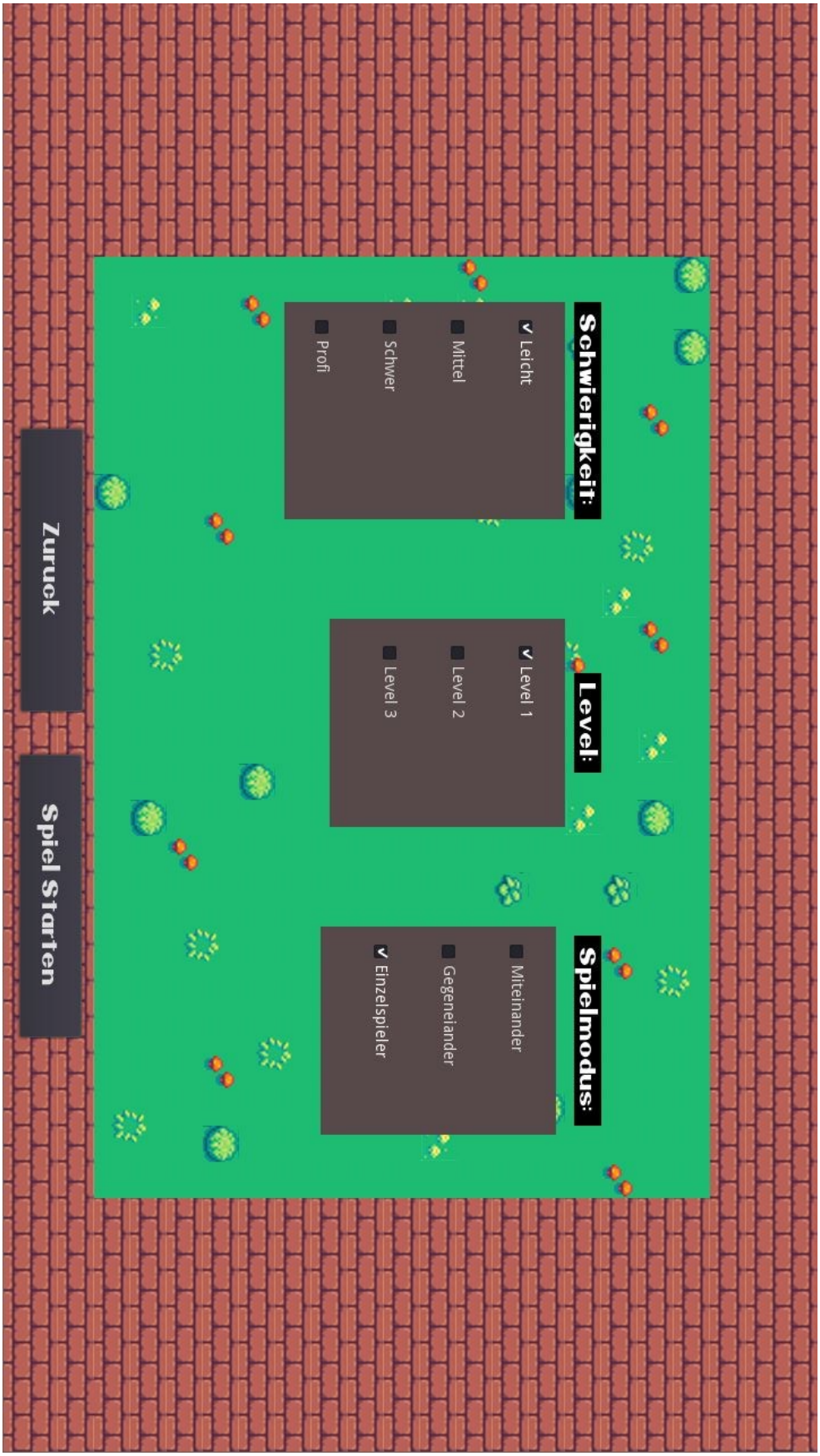
WIKIPEDIA: Snake (Computerspiel), 2024. In: https://de.wikipedia.org/w/index.php?title=Snake_(Computerspiel)&oldid=243810583 (26.09.2024)

Anhangsverzeichnis

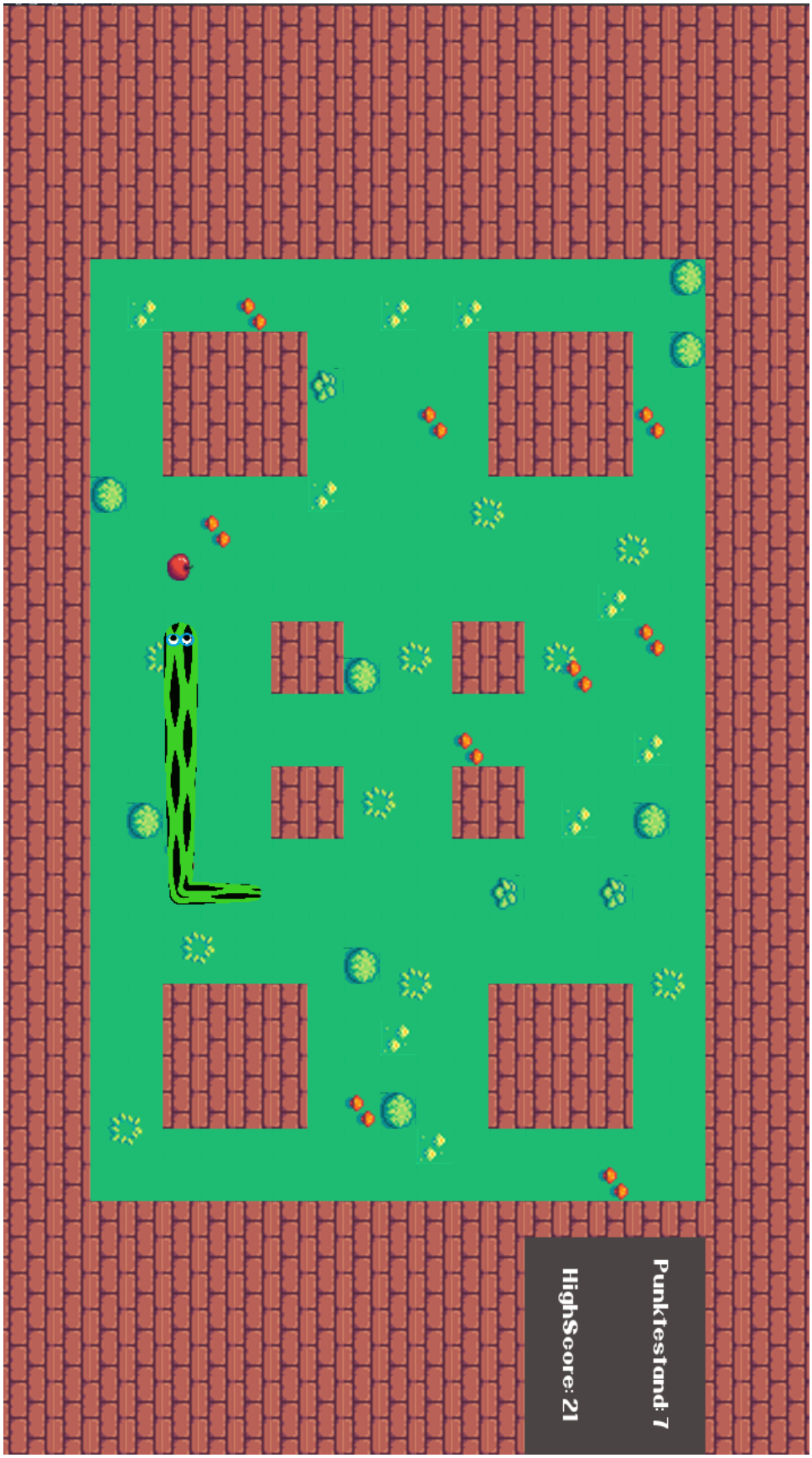
Anhang 1: Startseite der Anwendung



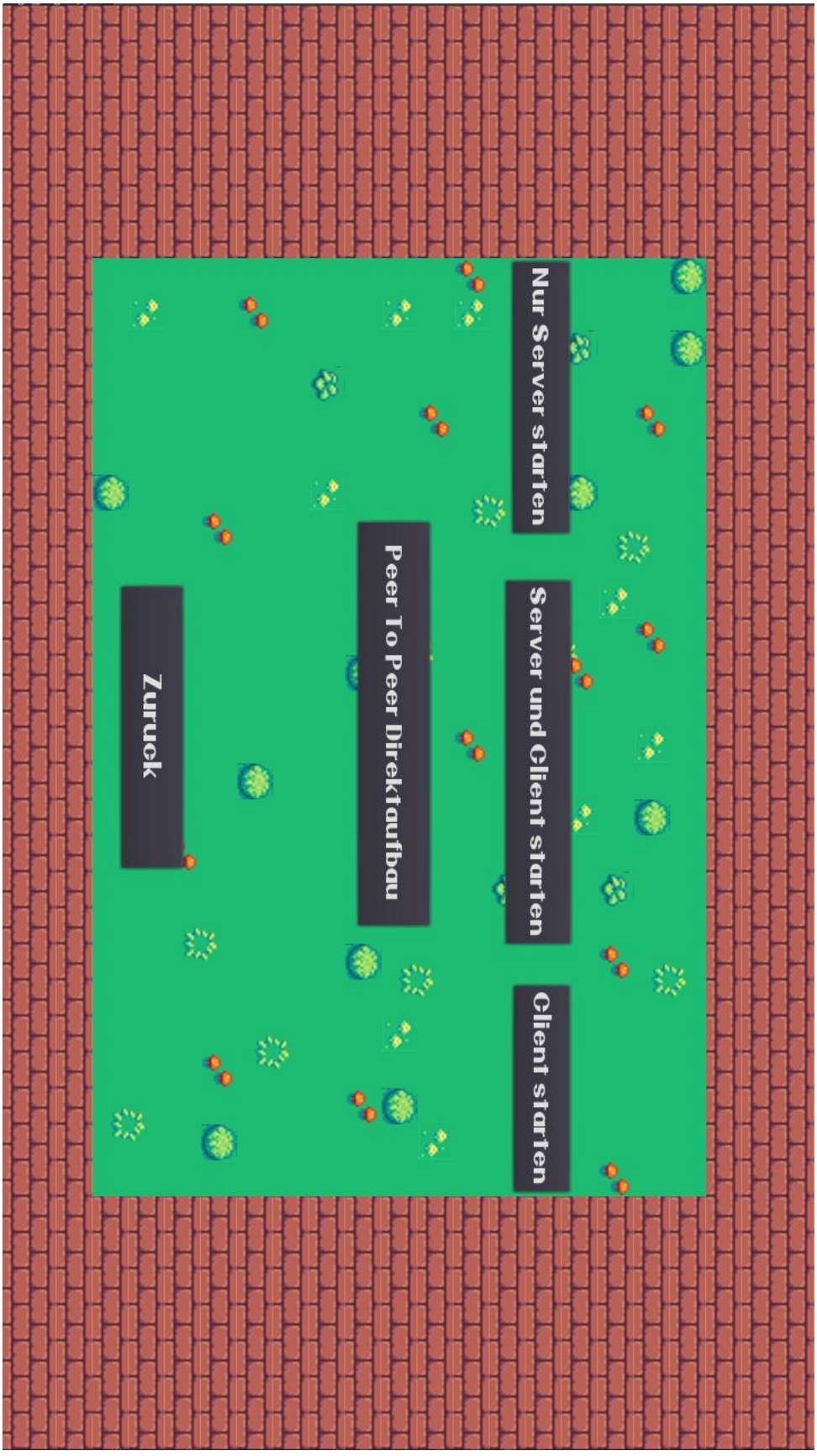
Anhang 2: Levelauswahl-Szene



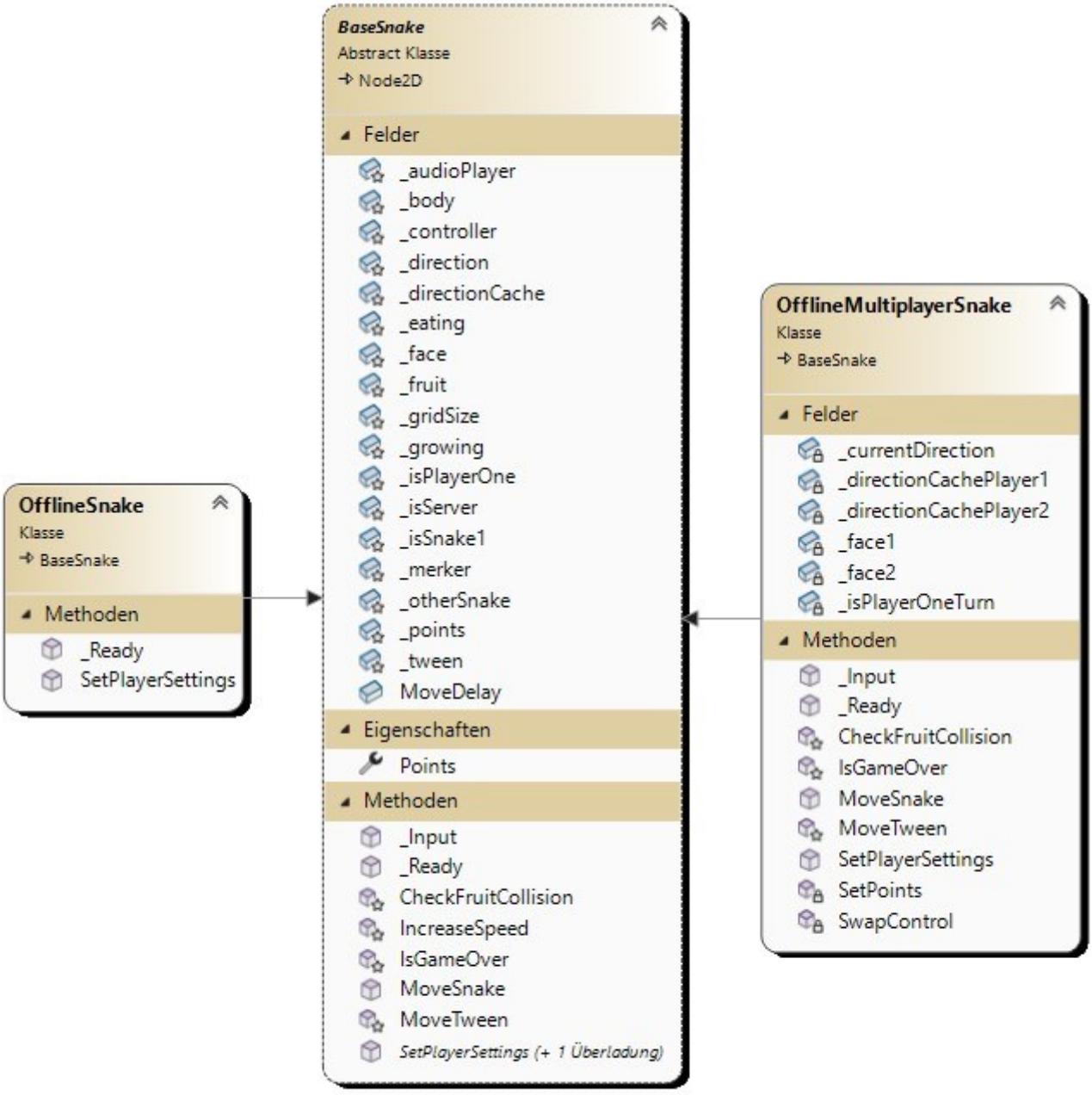
Anhang 3: Level-Szene



Anhang 4: Auswahl der Verbindungsart



Anhang 5: Klassendiagramm für die Offline-Logik



Anhang 6: Methoden zur Bewegung der Schlange

```
public virtual void MoveSnake()
{
    _direction = _directionCache;
    _tween.InterpolateMethod(this, "MoveTween", 0, 1, MoveDelay,
     Tween.TransitionType.Linear, Tween.EaseType.InOut);
    _tween.Start();
}

protected virtual void MoveTween(float argv)
{
    int i = 0;
    foreach (Vector2 pos in _body.Points)
    {
        Vector2 newPos, diff = Vector2.Zero;
        if (i == 0)
            newPos = _points[i] + _direction * new Vector2(_gridSize * argv,
             _gridSize * argv);
        else
        {
            if (!(_growing == true && i == _body.Points.Count() - 1))
            {
                diff = Vector2.Zero;
                if (_points[i - 1].x - _points[i].x != 0)
                    diff.x = (_points[i - 1].x - _points[i].x) / _gridSize;
                if (_points[i - 1].y - _points[i].y != 0)
                    diff.y = (_points[i - 1].y - _points[i].y) / _gridSize;

                newPos = _points[i] + diff * new Vector2(_gridSize * argv,
                _gridSize * argv);
            }
            else
            {
                newPos = _body.GetPointPosition(i);
            }
        }
        _body.SetPointPosition(i, newPos);
        i++;
    }
  …
}
```

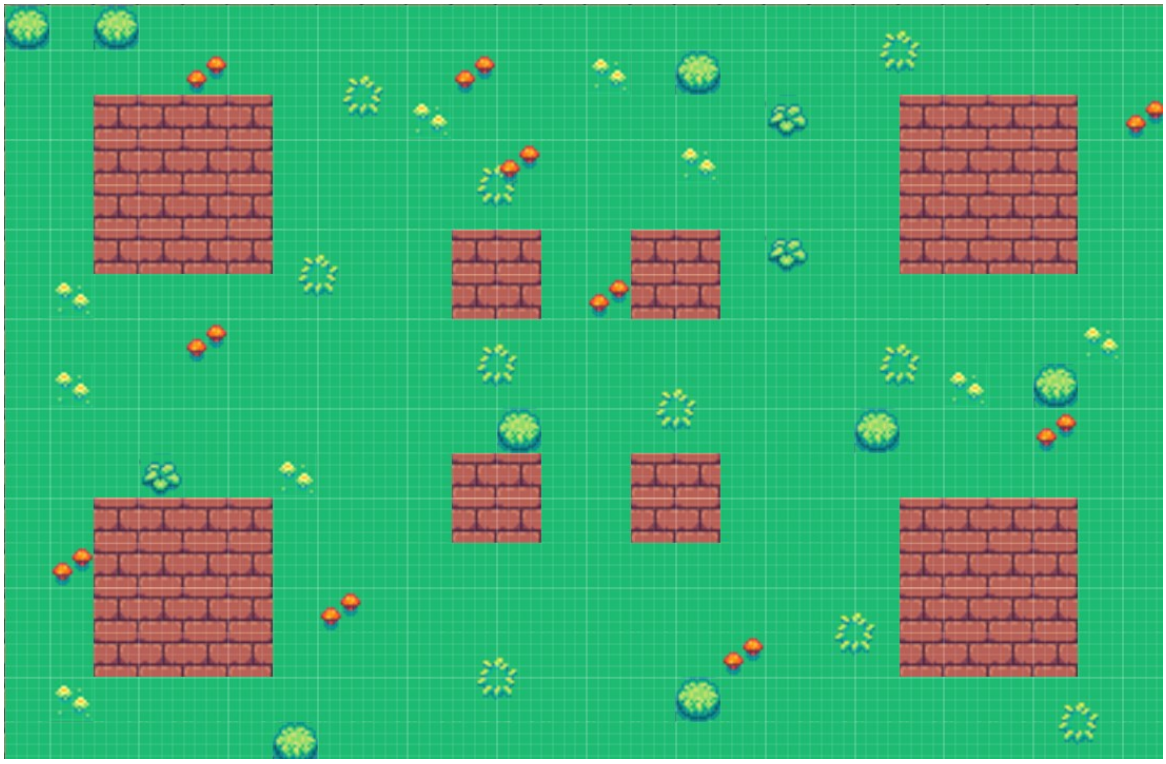
Anhang 7: Methoden zur Positionsbestimmung der Frucht

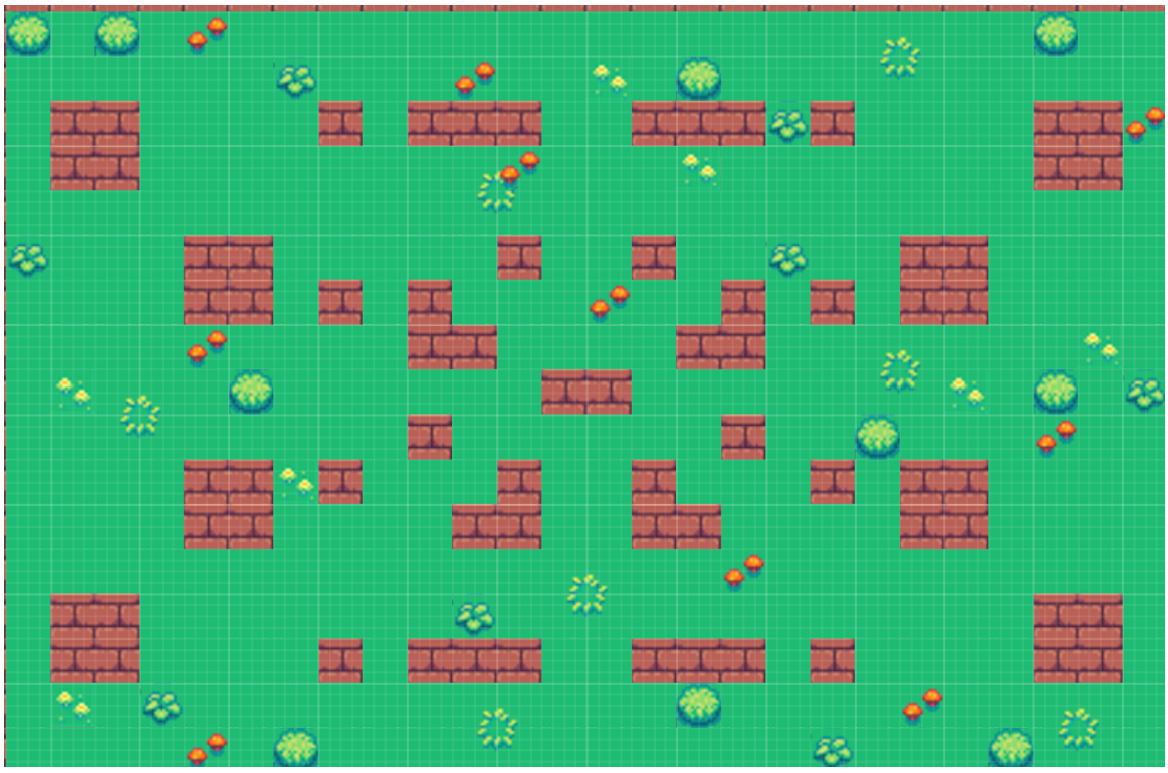
```
public Vector2 RandomizePosition()
{
    Vector2 position;
    Random random = new Random();

    do
    {
        position = GetRandomPos(random);
    }
    while (IsPositionOccupied(position));

    GD.Print("Frucht Position: " + position);
    return position + new Vector2(16,16);
}

private Vector2 GetRandomPos(Random random)
{
    int xMax = _controller.GameField.GetLength(1) - 1;
    int yMax = _controller.GameField.GetLength(0) - 1;

    float xPos = random.Next(7, xMax) * _cellSize;
    float yPos = random.Next(3, yMax) * _cellSize;

    return new Vector2(xPos, yPos);
}

public bool IsPositionOccupied(Vector2 position)
{
    int x = (int)(position.x / _cellSize);
    int y = (int)(position.y / _cellSize);

    if (x < 0 || x >= _controller.GameField.GetLength(1) || y < 0 || y >=
    _controller.GameField.GetLength(0))
    {
        return true;
    }

    if (_controller.GameField[y, x] == 1)
    {
        return true;
    }

    if (_snake1.Points.Contains(position))
    {
        return true;
    }

    if (_snake2 != null && _snake2.Points.Contains(position))
    {
        return true;
    }

    if (_snake3 != null && _snake3.Points.Contains(position))
    {
        return true;
    }

    return false;
}
```

Anhang 8: Klasse und Methoden zur Verwaltung des Punktestands

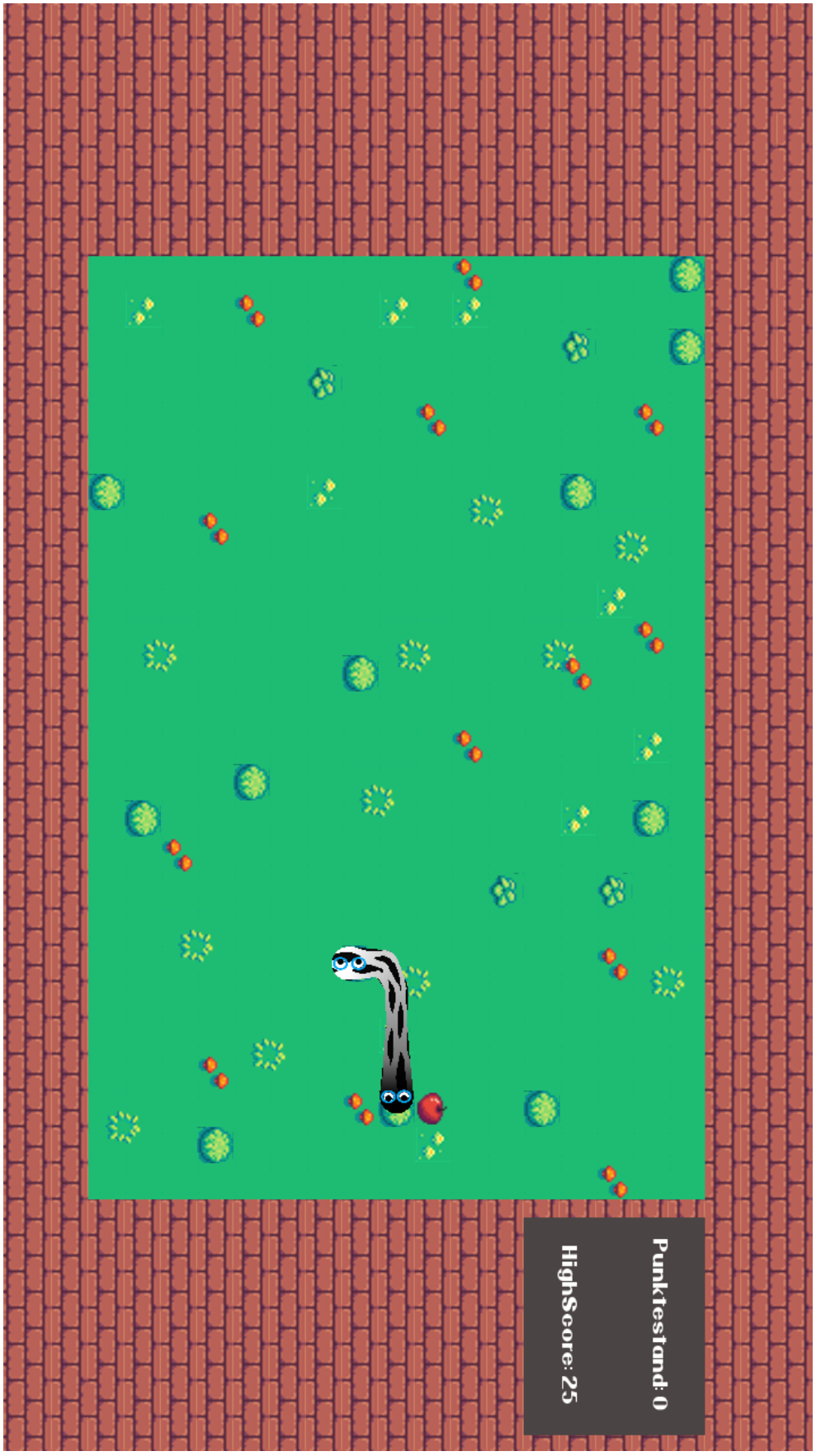
```
public class HighScores
{
    public Dictionary<string, int> HighScoreDict = new Dictionary<string, int>();
}

public class HighScoreManager
{
    private readonly string _filePath =
    ProjectSettings.GlobalizePath("user://highscores.json");
    private HighScores _highscores;

    public HighScoreManager()
    {
        LoadHighScores();
    }

    private void LoadHighScores()
    {
        if (System.IO.File.Exists(_filePath))
        {
            var json = System.IO.File.ReadAllText(_filePath);
            _highscores = JsonConvert.DeserializeObject<HighScores>(json);
            GD.Print("HighScores geladen");
        }
        else
        {
            _highscores= new HighScores();
            SaveHighScores();
            GD.Print("HighScore Datei neu erstellt");
        }
    }

    private void SaveHighScores()
    {
        var json = JsonConvert.SerializeObject(_highscores);
        System.IO.File.WriteAllText(_filePath, json);
        GD.Print("HighScores gespeichert");
    }

    public void SetHighScore(string level, int score)
    {
        if(_highscores.HighScoreDict.ContainsKey(level))
        {
            if(score > _highscores.HighScoreDict[level])
            {
                _highscores.HighScoreDict[level] = score;
                SaveHighScores();
                GD.Print("Neuer HighScore gesetzt");
            }
        }
        else
        {
            _highscores.HighScoreDict[level] = score;
            SaveHighScores();
        }
    }

    public int GetHighScore(string level)
    {
        if (_highscores.HighScoreDict.ContainsKey(level))
        {
            return _highscores.HighScoreDict[level];
        }
        else
        {
            return 0;
        }
    }
}
```

Anhang 9: Spielfelder für das zweite und dritte Level

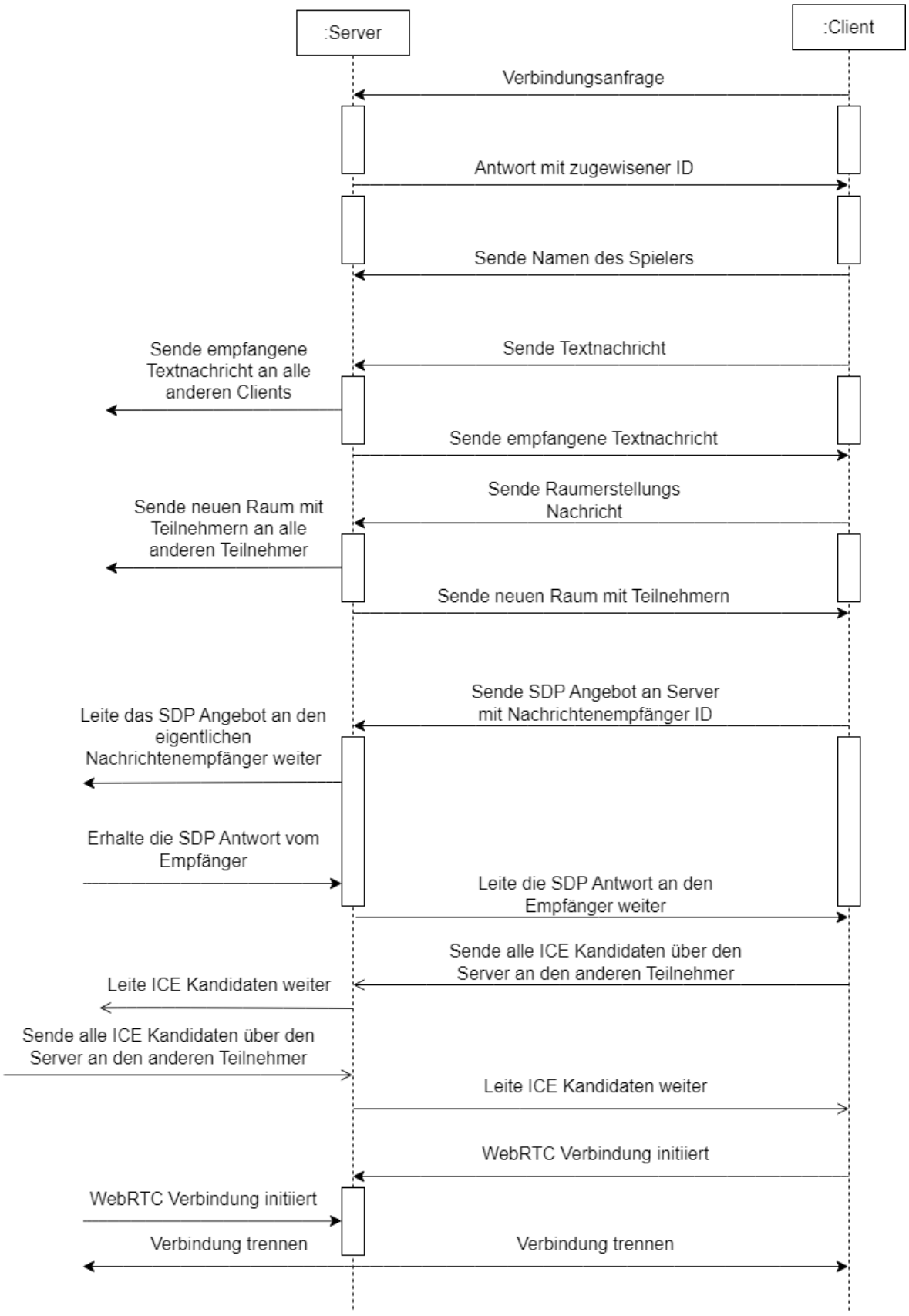




Anhang 10: Kooperativer Spielmodus



Anhang 11: Kommunikation zwischen dem Websocket Client und Server



Anhang 12: Erzeugen einer flüssigen Schlangenbewegung auf dem Client

```
public override void _PhysicsProcess(float delta)
{
  if(!_isServer)
  {
    float diff = 0f;
    if(_CalculateNewLatenyFactor)
    {
      // _TargetPoints ist ein Array von vector2 (2-dimensionaler Vektor)
      // _body ist der angezeigte Schlangenkörper, ebenfalls ein Array
      // aus Vector. GetPoints liefert indexbezogen die Positionen der
      // einzelnen Schlangensegmente
      if(_TargetPoints[0].x - _body.GetPointPosition(0).x != 0f)
        diff = _TargetPoints[0].x - _body.GetPointPosition(0).x;
      else
        diff = _TargetPoints[0].y - _body.GetPointPosition(0).y;
      latencyFactor = delta / (float)(_ClientTimeDiffBodyUpdate);

      // Berechnung eines Durchschnittwertes um kurzzeitige
      // Netzwerkschwankungen auszugleichen
      MakeAverageLatencyFactor();
      _CalculateNewLatenyFactor = false;
      // Die aktuellen TargetPoints speichern, da sie asynchron
      // aktualisiert werden können, was zu Rucklern führt.
      _SavedTargetPoints.Clear();
      for(int i = 0; i < _TargetPoints.Count(); i++)
        _SavedTargetPoints.Add(new Vector2(_TargetPoints[i]));
      _points = _body.Points;
      // eigentliches weitersetzen der Positionen:
      for(int i = 0; i < _SavedTargetPoints.Count(); i++)
      {
        Vector2 DiffVec = _SavedTargetPoints[i] - _points[i];
        DiffVec *= latencyFactor;
        Vector2 newPos = _body.Points[i] + DiffVec;
        // Schlangenkörper (_body) auf neu berechnete Position setzen
        // i = Index des Schlangensegments, newPos die neue Position
        _body.SetPointPosition(i, newPos);
      }
      // Gesicht nachsetzen
      _face.Position = _body.Points[0];
      _face.RotationDegrees=-
      Mathf.Rad2Deg(_direction.AngleTo(Vector2.Right));
  }
}
```

Eidesstattliche Erklärung

Ich erkläre an Eides statt,

dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Zustimmung des/der beteiligten Unternehmen/s zur Verwendung betrieblicher Unterlagen habe ich eingeholt.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht noch einer anderen Prüfungsbehörde/-stelle vorgelegt.

Grünitz, Lukas

Name, Vorname Verfassender

Glauchau, 25.11.2024

Ort, Datum Abgabetermin

Unterschrift Verfassender

Eidesstattliche Erklärung

Ich erkläre an Eides statt,

dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Zustimmung des/der beteiligten Unternehmen/s zur Verwendung betrieblicher Unterlagen habe ich eingeholt.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht noch einer anderen Prüfungsbehörde/-stelle vorgelegt.

Roscher, Maurice

Name, Vorname Verfassender

Glauchau, 25.11.2024

Ort, Datum Abgabetermin

Unterschrift Verfassender

BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
GLAUCHAU
UNIVERSITY OF COOPERATIVE EDUCATION

Erklärung zur Prüfung wissenschaftlicher Arbeiten

Die Bewertung wissenschaftlicher Arbeiten erfordert die Prüfung auf Plagiate. Die hierzu von der Staatlichen Studienakademie Glauchau eingesetzte Prüfungskommission nutzt sowohl eigene Software als auch diesbezügliche Leistungen von Drittanbietern. Dies erfolgt gemäß § 7 des Gesetzes zum Schutz der informationellen Selbstbestimmung im Freistaat Sachsen (Sächsisches Datenschutzgesetz – SächsDSG) vom 25. August 2003 (Rechtsbereinigt mit Stand vom 31. Juli 2011) im Sinne einer Datenverarbeitung im Auftrag.

Der Studierende bevollmächtigt die Mitglieder der Prüfungskommission hiermit zur Inanspruchnahme o.g. Dienste. In begründeten Ausnahmefällen kann der Datenschutzbeauftragte der Staatlichen Studienakademie Glauchau sowohl vom Verfasser der wissenschaftlichen Arbeit als auch von der Prüfungskommission in den Entscheidungsprozess einbezogen werden.

Name:	Grünitz
Vorname:	Lukas
Matrikelnummer:	4004987
Studiengang:	Technische Informatik
Titel der Arbeit:	
Datum:	
Unterschrift:	

BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
GLAUCHAU
UNIVERSITY OF COOPERATIVE EDUCATION

Erklärung zur Prüfung wissenschaftlicher Arbeiten

Die Bewertung wissenschaftlicher Arbeiten erfordert die Prüfung auf Plagiate. Die hierzu von der Staatlichen Studienakademie Glauchau eingesetzte Prüfungskommission nutzt sowohl eigene Software als auch diesbezügliche Leistungen von Drittanbietern. Dies erfolgt gemäß § 7 des Gesetzes zum Schutz der informationellen Selbstbestimmung im Freistaat Sachsen (Sächsisches Datenschutzgesetz – SächsDSG) vom 25. August 2003 (Rechtsbereinigt mit Stand vom 31. Juli 2011) im Sinne einer Datenverarbeitung im Auftrag.

Der Studierende bevollmächtigt die Mitglieder der Prüfungskommission hiermit zur Inanspruchnahme o.g. Dienste. In begründeten Ausnahmefällen kann der Datenschutzbeauftragte der Staatlichen Studienakademie Glauchau sowohl vom Verfasser der wissenschaftlichen Arbeit als auch von der Prüfungskommission in den Entscheidungsprozess einbezogen werden.

Name:	Roscher
Vorname:	Maurice
Matrikelnummer:	4004838
Studiengang:	Technische Informatik
Titel der Arbeit:	
Datum:	
Unterschrift:	