Dokumentation von Alexej Kunz

Teamname	SoulThieves
Rolle(n)	UI-Designer und Entwickler

Aufgaben:

- Anpassen der Ressourcen und Erstellen der Animationen
- Implementierung der Power-Up Logik
- Gestalten der Multiplayer-Lobbyliste
- Gestalten und Implementieren des Ingame- und Settings-Menüs

Gliederung:

1. Prozessdokumentation
   1.1 Definitonen der Stadien
   1.2 Verwendete Tools und deren Nutzen
      1.2.1 GitHub als Versionsverwaltungsdienst
      1.2.2 Discord als Kommunikationstool
      1.2.3 Godot als Game engine
      1.2.4 Notion als “Tool für alles”
2. Erstellen der Animationen
   2.1 Anpassen der Ressourcen
   2.2 Erstellen der Laufanimation des Spielers
   2.3 Erstellen der Bombenanimationen
3. Implementierung der Power-Up Logik
   3.1 Aufheben eines Power-Ups
   3.2 Implementierung der Logik
4. Erstellen des Menüs für das Beitreten einer Lobby
   4.1 Struktur des Menüs
   4.2 Struktur einer Reihe
   4.3 Implementierung einer Reihe
   4.4 Auslesen der Sessions
5. Erstellen des Ingame-Menüs und des Settings-Menüs
   5.1 Struktur des Ingame-Menüs
   5.2 Implementierung des Ingame-Menüs
   5.3 Struktur des Settings-Menüs
   5.4 Implementierung des Settings-Menüs

1. Prozessdokumentation

Dieser Abschnitt beschäftigt sich ausschließlich mit der Prozessdokumentation. Darin wird beschrieben, wie bestimmte Stadien in diesem Projekt definiert wurden und wie in diesem Projekt vorgegangen wurde, also beispielsweise **welche** Tools **wofür** verwendet wurden.

1.1 Definitonen der Stadien

Hier beschäftigen wir uns mit der Definition der von uns verwendeten Stadien “not started”, “in progress” und “complete”. Diese wurden von uns in der Notion Roadmap verwendet (für mehr Informationen zu Notion und der dazugehörigen Roadmap siehe Abschnitt 1.2.4), um Kollegen über den aktuellen Status bestimmter Aufgaben zu informieren.

💡 **Definiton “not started”:** Als “not started” wird eine Aufgabe bezeichnet, für die weder der Hauptverantwortliche noch dessen Stellvertreter erste Recherchen oder Vorbereitungen getroffen hat. Diese Aufgabe ist also noch komplett “unangefasst”.

💡 **Definition “in progress”:** Der Status “in progress” darf verwendet werden, sobald entweder der Hauptverantwortliche oder dessen Stellvertreter erste Recherchen oder Vorbereitungen für die Aufgabe getroffen haben und läuft, so lange die Implementierung andauert.

💡 **Definition “complete”:** Eine Aufgabe darf erst dann als “complete” gekennzeichnet werden, sobald der Hauptverantwortliche zufrieden mit der Implementierung ist und auch erste Tests durchgeführt wurden. Sollte der Stellvertreter jedoch etwas zu beanstanden haben, so wird der Status wieder auf “in progress” gesetzt.

1.2 Verwendete Tools und deren Nutzen

Die folgenden Abschnitte nennen und beschreiben die von uns verwendeten Tools und erklären zusätzlich, inwiefern wir als Team, als auch unser Projekt an sich, von den Tools profitieren konnten.

1.2.1 GitHub als Versionsverwaltungsdienst

Jedes größere Softwareprojekt benötigt eine Art Versionsverwaltungsdienst, damit alle Entwickler auf dem gleichen Stand sind und notfalls auch ältere Versionen des Projekts verwendet werden können, falls etwas in einer neueren Auslieferung schief gelaufen sein sollte. Auch wir als Team SoulThieves haben für unseren Granatier-Clon ein Versionsverwaltungsdienst verwendet, nämlich GitHub. GitHub wurde bei uns vor allem genutzt, damit alle Entwickler auf dem gleichen Stand sind, was den Code angeht. Aber auch Dateien, die zwar nicht für das Spiel, wohl aber für das Projekt an sich essentiell sind, wurden dort hochgeladen. Als Beispiel dafür lässt sich unsere Dokumentation nennen. Dadurch bietet GitHub uns zusätzlich einen Schutz vor Dateiverlusten.

1.2.2 Discord als Kommunikationstool

Um sich über den aktuellen Stand des Projekts auszutauschen und auch um die nächsten Schritte zu planen, wurde das kostenlose Tool Discord verwendet. Darin wurde eine Gruppe mit allen Entwicklern erstellt, in der entweder über Themen gechattet wurde, oder in der man sich zu Meetings verabredet hat. In Discord`s Telefonkonferenzen haben wir vor allem von der Möglichkeit gebrauch gemacht, dass ein oder mehrere Personen ihren Bildschirm teilen konnten. So konnte man besser auf Probleme der Teammitglieder eingehen und diese effektiv als Team beheben.

1.2.3 Godot als Game engine

Natürlich ist es auch möglich Spiele ohne eine Game engine zu erstellen. Das würde das Ganze aber unnötig aufwändig machen, weshalb es schön ist, dass es Game engines gibt, die einem unter die Arme greifen. Auch wir verwendeten deshalb eine Game engine, nämlich Godot. Diese erleichterte uns vor allem das Erstellen von Animationen, die Steuerung physischer Prozesse (wie z.B. Bewegung, Hitbox…), sowie die Organisation unserer Dateien. Mit der mitgelieferten Programmiersprache GDScript, die Ähnlichkeiten zu Python aufweist, wurde das

Erstellen eines Spiels nochmals erleichtert, weil diese Programmiersprache speziell für das Entwickeln von Spielen ausgerichtet ist. Dadurch sind bereits eingebaute Methoden und Signale verfügbar, die einem einiges an Arbeit ersparen können.

1.2.4 Notion als “Tool für alles”

Auch wenn “Tool für alles” hier vielleicht etwas zu groß gefasst ist, war Notion trotzdem das Tool, auf dem wir als Team sehr viel Zeit verbracht haben. Das liegt vor allem an dem großen Umfang des Tools. So wurde Notion beispielsweise verwendet, um die Protokolle zu verfassen, gemeinsam an Dokumenten zu arbeiten, die nächsten Schritte festzuhalten, Aufgaben zu verteilen, den Zeitplan zu analysieren oder eben auch um diese Dokumentation zu verfassen. Besonders hervorzuheben war dabei die Roadmap, in der wir unseren Zeitplan festlegen konnten, Sprints planen konnten, die Prioritäten für Aufgaben verteilen konnten, den Status und die Zuständigkeiten der Aufgaben einsehen konnten und noch vieles mehr.

2. Erstellen der Animationen

Eine meiner Aufgaben war das Erstellen grundlegender Animationen für unseren Bomberman Klon. Dafür waren jedoch zu Beginn einige Vorbereitungen zu treffen. So mussten zunächst die Texturen aus dem GitHub unserer „Spielvorlage“ Granatier geholt werden. Da Granatier ein Open Source Projekt mit der GPL 2.0 License ist, können die Texturen ohne weiteres für unser Projekt verwendet werden.

2.1 Anpassen der Ressourcen

Begonnen wurde die Aufgabe damit, dass man zunächst die im Granatier GitHub vorhandenen Texturdateien in eine für Godot akzeptierte Form bringt, um Godot´s Animationstools effektiv nutzen zu können. Die daraus resultierenden Dateien sind in unserem GitHub unter GranatierClient/resources/images einsehbar. Nachdem das richtige Formatieren der Texturen abgeschlossen war, konnten die spannenderen Aufgaben des Sprints angegangen werden, nämlich die Animationen.

2.2 Erstellen der Laufanimation des Spielers

Um die Animationen für den Spieler erstellen zu können, musste zunächst eine Scene erstellt werden. Diese Scene bekam ein “AnimatedSprite” als Kindknoten. Ein “AnimatedSprite” ist die grafisch animierte Repräsentation einer Scene. Godot beinhaltet einen simplen Animationseditor für den “AnimatedSprite”. Hier müssen lediglich die Bilder, die hintereinander abgespielt werden sollen, reingeladen werden, die Abspielgeschwindigkeit in FPS (Frames per second bzw. Bilder pro Sekunde) angegeben werden und es muss festgelegt werden, ob die Animation in einer Schleife abgespielt werden soll oder nach einem durchlauf beendet werden soll. Wir benötigen hier eine Schleife, da der Spieler durchaus länger laufen kann als die Dauer der Animation. In Abbildung 2.1 kann der Animationseditor des “AnimatedSprites” mit den eingefügten Bildern für die Laufanimation des Spielers betrachtet werden. Zu beachten ist hier, dass man auf der linken Seite für jeden der fünf Spieler eine eigene Animation erstellt hat, die im Endeffekt zwar identisch sind, jedoch erstellt werden mussten, da die Spieler sich in ihrer Farbe unterscheiden.

Abbildung 2.1

2.3 Erstellen der Bombenanimationen

Nachdem alle Animationen für die Spieler erstellt wurden, wurde sich der Bombenanimation gewidmet. Diese soll einen pulsierenden Effekt haben, also größer und kleiner werden, nachdem sie gelegt wurde. Dafür musste zunächst, ähnlich wie beim Spieler, eine Scene erstellt werden. Diese Scene bekam jetzt aber keinen “AnimatedSprite” als Kindknoten, sondern lediglich einen “Sprite”. Der Unterschied ist hier, dass ein “Sprite” keine Animation hat. Die Animation soll diesmal nämlich nicht mit einem “AnimatedSprite” abgewickelt werden, sondern mit einem Kindknoten des Sprites namens “AnimationPlayer”. Der “AnimationPlayer” ist eine weitere Möglichkeit für Animationen in Godot. Die Wahl fiel hier auf den “AnimationPlayer”, weil damit der pulsierende Effekt der Bombe leichter dargestellt werden kann. Im “AnimationSprite” bräuchte man für diese Animation mehrere Texturen der Bombe in verschiedenen Größen, um einen Pulseffekt zu erzeugen, während der “AnimationPlayer” eine integrierte Funktion dafür hat und eine einzige Textur der Bombe ausreicht.

Die Bombenanimation wurde mit dem “AnimationPlayer” in folgenden Schritten erreicht:

- Dem Bomb Sprite wurde die Bomben Textur zugewiesen
- Unter dem Button “Add Track“ des “AnimationPlayers” wurde die “scale“ property für den Bomb Sprite ausgewählt, um die Größe der Bombe für die Animation ändern zu können
- Die Länge der Animation wurde auf zwei Sekunden und Loop eingestellt
- In den Sekunden 0, 1 und 2 wurde mit rechtsklick ein Key zugewiesen (werden als Route dargestellt)
- Die Value des mittleren Keys wurde in x und y Richtung auf 1.1 gesetzt -> Die Bombe wird dadurch von Sekunde 0 auf Sekunde 1 um 10 Prozent vergrößert und von Sekunde 1 auf Sekunde 2 wieder auf den Ursprungswert verkleinert

Damit wurde die pulsierende Animation der Bombe erschaffen. Die Abbildung 2.2 gibt einen Einblick in den Animationseditor des “AnimationPlayers”.

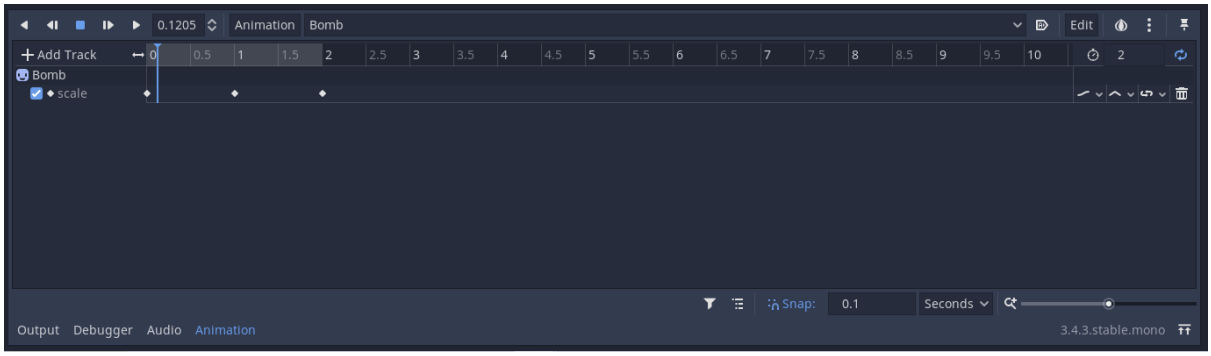

Abbildung 2.2

Eine weitere, wichtige Animation war die Animation der Explosion der Bombe. Diese wurde wiederum mit einem “AnimiatedSprite” erstellt und ist nahezu identisch zur Erstellung der Animation für den Player, weshalb hier nicht so viel dazu erläutert wird. Wichtig zu wissen ist hier lediglich, dass der “ExplosionSpread” (also die vier Explosionsrichtungen) auf vier einzelne “AnimatedSprites” verteilt werden mussten, da es zu Fällen kommen kann, in der beispielsweise nur drei Explosionsrichtungen angezeigt werden sollen, weil die Bombe angrenzend an einer festen Wand der Map ist und die Animation dort nicht durchgeführt werden soll. Einen groben Überblick bietet dabei Abbildung 2.3.

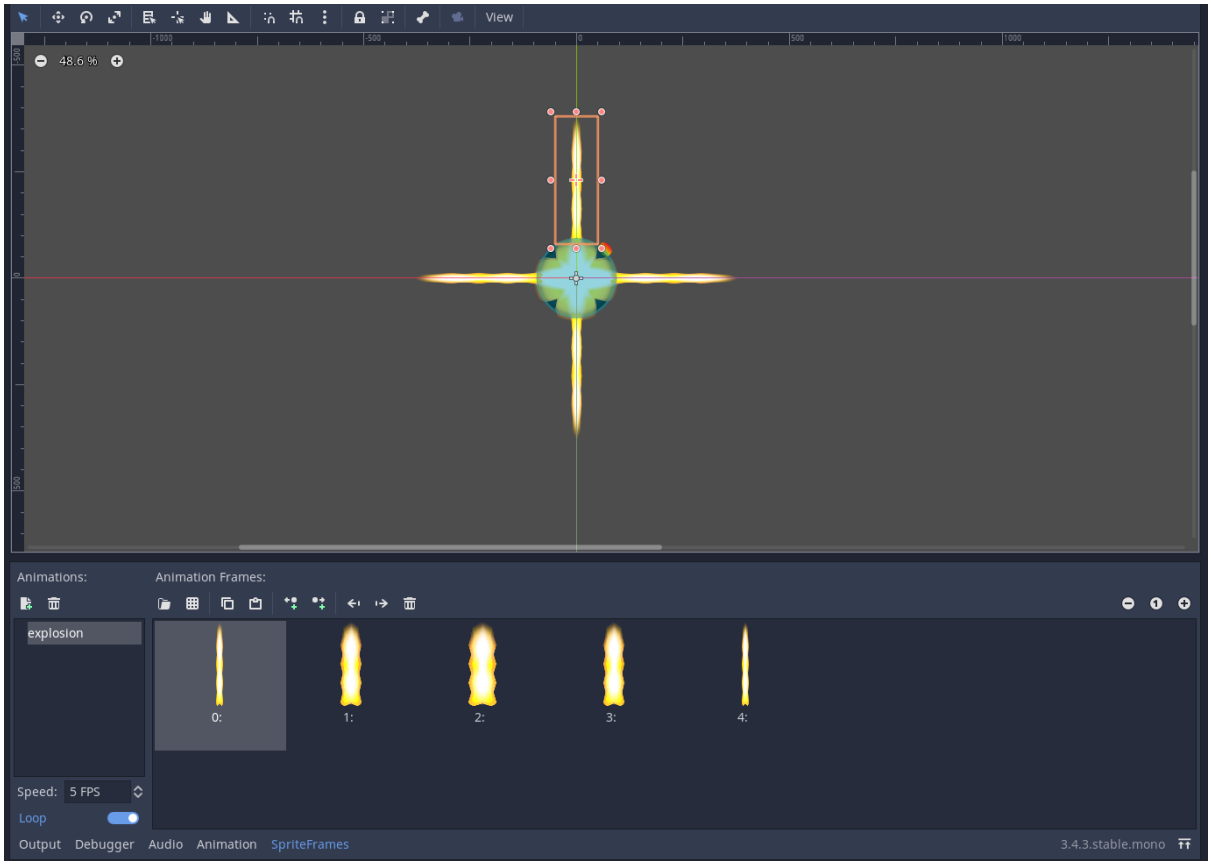

Abbildung 2.3

3. Implementierung der Power-Up Logik

Ein wichtiger Bestandteil unseres Bomberman-Spiels sind Power-Ups. Diese beeinflussen die Fähigkeiten des Spielers, der ein Power-Up aufsammelt. Dabei kann ein Power-Up den Spieler positiv, aber auch negativ beeinflussen. Die genaue Erklärung der Wirkungsweise der Power-Ups ist in der Dokumentation meines Kollegen Frank zu sehen. Im Folgenden wird auf die Implementierung der Power-Up Logik eingegangen, in dem vor allem Codeabschnitte gezeigt und erklärt werden. Die Skripte der Codeabschnitte befinden sich auf dem Serverseitigen Godot Projekt, da der Server dafür verantwortlich sein soll, die Manipulationen der Spielerattribute durchzuführen.

3.1 Aufheben eines Power-Ups

```
# Called when the node enters the scene tree for the first time.
func _ready():
  var randomNumber = RandomNumberGenerator.new()
  randomNumber.randomize()
  type = randomNumber.randi_range(0, Types.size() - 1)


#Signal that is called, when the Player picks up a Powerup
func _on_Powerup_body_entered(body):
  if body.is_in_group("Players"):
    body.pick_up_powerup(type)
    queue_free()
```

Abbildung 3.1

In Abbildung 3.1 ist ein Codeabschnitt für das “Powerup.gd”-Skript zu sehen. Darin ist lediglich die Funktion “_on_Powerup_body_entered(body)” hervorzuheben. Diese Funktion wird durch ein Signal aufgerufen. Das Signal wird dann ausgelöst, wenn sich ein anderes Objekt auf einem Power-Up befindet. Die Funktion überprüft daraufhin, ob sich dieses Objekt in der Gruppe “Players” befindet. In dieser Gruppe sind alle Spieler abgespeichert, die am laufenden Spiel teilnehmen. Ist das Objekt ein Spieler, so wird auf ihm die “pick_up_powerup(type)”-Methode aufgerufen, die in

Abbildung 3.3 näher erläutert wird. Durch die Variable Type wird der “pick_up_powerup”-Methode der Typ des Power-Ups mitgegeben. Am Ende wird mit “queue_free()” das Power-Up Objekt, das gerade vom Spieler aufgenommen wurde, zerstört.

3.2 Implementierung der Logik

Die Codeabschnitte der Abbildungen 3.2, 3.3 und 3.4 befinden sich jeweils im Player.gd Skript auf dem Serverseitigen Godot Projekt.

```
var stats = {
  "speed": 100,
  "can_kick": false,
  "can_throw": false,
  "has_shield": false,
  "layable_bombs": 1,
  "bomb_blast_range": 1,
  "is_scatty": false,
  "is_restrained": false,
  "has_mirror": false,
  "has_teleport": false,
  "hyperactive": false,
  "slow": false,
  "is_dead": false,
  "fallen": false,
  "on_ice": Vector2.ZERO
}
```

Abbildung 3.2

In Abbildung 3.2 sind die Standardattribute des Spielers zu sehen, also die Attribute, die ein Spieler hat, bevor er jemals ein Power-Up eingesammelt hat. Diese werden durch Power-Ups manipuliert.

```
func pick_up_powerup(type):
  match type:
    Powerup.Types.BAD_HYPERACTIVE:
      if not has_bad_powerup:
        stats.hyperactive = true
        has_bad_powerup = true
        $Timer.start()
    Powerup.Types.BAD_MIRROR:
      if not has_bad_powerup:
        stats.has_mirror = true
        has_bad_powerup = true
        $Timer.start()
    Powerup.Types.BAD_RESTRAIN:
      if not has_bad_powerup:
        stats.is_restrained = true
        has_bad_powerup = true
        $Timer.start()
    Powerup.Types.BAD_SCATTY:
      if not has_bad_powerup:
        stats.is_scatty = true
        has_bad_powerup = true
        $Timer.start()
        $ScattyTimer.start()
    Powerup.Types.BAD_SLOW:
      if not has_bad_powerup:
        stats.slow = true
        has_bad_powerup = true
        $Timer.start()
    Powerup.Types.BOMB:
      stats.layable_bombs += 1
    Powerup.Types.KICK:
      stats.can_kick = true
    Powerup.Types.POWER:
      stats.bomb_blast_range += 1
    Powerup.Types.SHIELD:
      if not stats.has_shield:
        stats.has_shield = true
    Powerup.Types.SPEED:
      stats.speed += 20
    Powerup.Types.THROW:
      stats.can_throw = true
    Powerup.Types.NEUTRAL_TELEPORT:
```

```
      stats.has_teleport = true
      get_node("/root/Server").teleport_player(int(name))
    Powerup.Types.NEUTRAL_PANDORA:
      var randomizer = RandomNumberGenerator.new()
      randomizer.randomize()
      var randomNumber = randomizer.randi_range(0, Powerup.Types.size() - 3)
      pick_up_powerup(randomNumber)
```

Abbildung 3.3

Die Abbildung 3.3 zeigt die bereits in Abbildung 3.1 verwendete Funktion “pick_up_powerup(type)”. Der Parameter type gibt dabei den Typen des Power-Ups an, damit unterschieden werden kann, inwiefern die Attribute des Spielers manipuliert werden sollen. Wichtig ist, dass bei den negativen Power-Ups (mit “BAD” im Namen gekennzeichnet) zunächst abgefragt wird, ob der Spieler bereits ein negatives Power-Up hat, da immer nur ein negatives Power-Up gleichzeitig aktiv sein darf. Außerdem wird bei den negativen Power-Ups ein zehnsekündiger Timer gestartet, da ein negatives Power-Up lediglich zehn Sekunden wirken soll. Das Power-Up „NEUTRAL_PANDORA“ soll dem Spieler ein zufälliges Power-Up zuordnen. Deshalb wird zunächst eine Zufallszahl im Bereich 0 bis Powerup.Types.size()-3 ausgewählt. Die -3 ist deshalb da, weil size() die Größe des “Powerup.Types” enum um eins zu Groß ausgeben würde (da man beim Zugriff mit 0 beginnt) und die letzten beiden Power-Ups neutral sind, man dem Spieler aber nur ein positives oder negatives Power-Up zuteilen möchte.

```
func _on_bad_powerup_timer_timeout():
  stats.has_mirror = false
  stats.is_restrained = false
  stats.is_scatty = false
  stats.hyperactive = false
  stats.slow = false
  $ScattyTimer.stop()
  has_bad_powerup = false
```

Abbildung 3.4

Die Funktion “_on_bad_powerup_timer_timeout()” aus Abbildung 3.4 wird dann aufgerufen, wenn der Timer, der aufgrund eines negativen Power-Ups gesetzt wurde, abgelaufen ist. Dies wird durch ein Signal gesteuert. Die Funktion beendet daraufhin den Timer, da er sonst erneut zehn Sekunden herunterzählen würde und setzt alle Attribute des Spielers, die durch ein negatives Power-Up beeinflusst werden können, wieder zurück.

Da die tatsächliche Umsetzung des Spawnens der Power-Ups und das Anzeigen der dazu gehörigen Textur einer Server-Client-Kommunikation bedarf, wurde die Aufgabe von meinem Kollegen Andreas übernommen, da er für die Server-Client-Kommunikation zuständig war.

4. Erstellen des Menüs für das Beitreten einer Lobby

Da unser Bomberman-Clon, im Gegensatz zum originalen Granatier, eine Multiplayer-Funktion bietet, ist es wichtig das Ganze über eine ansprechende grafische Benutzeroberfläche abwickeln zu können. Konkret wurde hier die grafische Benutzeroberfläche für das Beitreten einer Lobby erstellt.

4.1 Struktur des Menüs

Abbildung 4.1 veranschaulicht die Struktur, die hinter der grafischen Oberfläche für das Beitreten einer Lobby steckt. Diese ist in der Szene “JoinExistingLobby“ auffindbar.

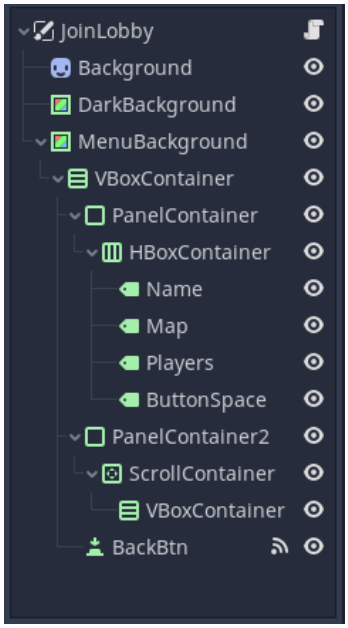

Abbildung 4.1

“JoinLobby” ist ein “CanvasLayer”, was häufig für die Gestaltung von grafischen Oberflächen in Godot Verwendung findet und ist der Elternknoten. “Background” ist dabei der wolkige Hintergrund unseres Spiels. “DarkBackground” sorgt für den abgedunkelten Hintergrund des Menüs, während “MenuBackground” die Größe und Farbe des Menüs angibt. Innerhalb des “MenuBackground” befindet sich ein “VBoxContainer”, um alle Elemente, die sich in dem Menü befinden, untereinander zu platzieren. Darin enthalten ist ein “PanelContainer”, der für den hellgrauen Hintergrund im Kopfbereich sorgt und dem darin enthaltenen “HBoxContainer” etwas Abstand zum Rand gibt. Der “HBoxContainer” ordnet seine Kindknoten (die Labels Name, Map, Players und ButtonSpace) horizontal an. Die Namen der Labels geben bereits an, was in diesem Label steht, mit Ausnahme von “ButtonSpace”. “ButtonSpace” ist lediglich ein leeres Label im Kopfbereich, unter dem in jeder Reihe ein Join-Button zu sehen sein soll (siehe auch Abbildung 4.7). Im “PanelContainer2” ist ein “ScrollContainer” vorhanden, wodurch im darin enthaltenen “VBoxContainer” gescrollt werden kann. Innerhalb dieses “VBoxContainers” werden sich Reihen befinden, die die aktuell laufenden Lobbys anzeigen. Zu guter Letzt ist noch der „BackBtn“ vorhanden, ein Button, mit dem das Menü wieder verlassen werden kann. Das Endresultat dieser Struktur und somit das erschaffene Menü ist in Abbildung 4.2 zu sehen. Das Aussehen der Labels und des Back-Buttons wurden hier noch mit Farben und einer Schriftart verändert.

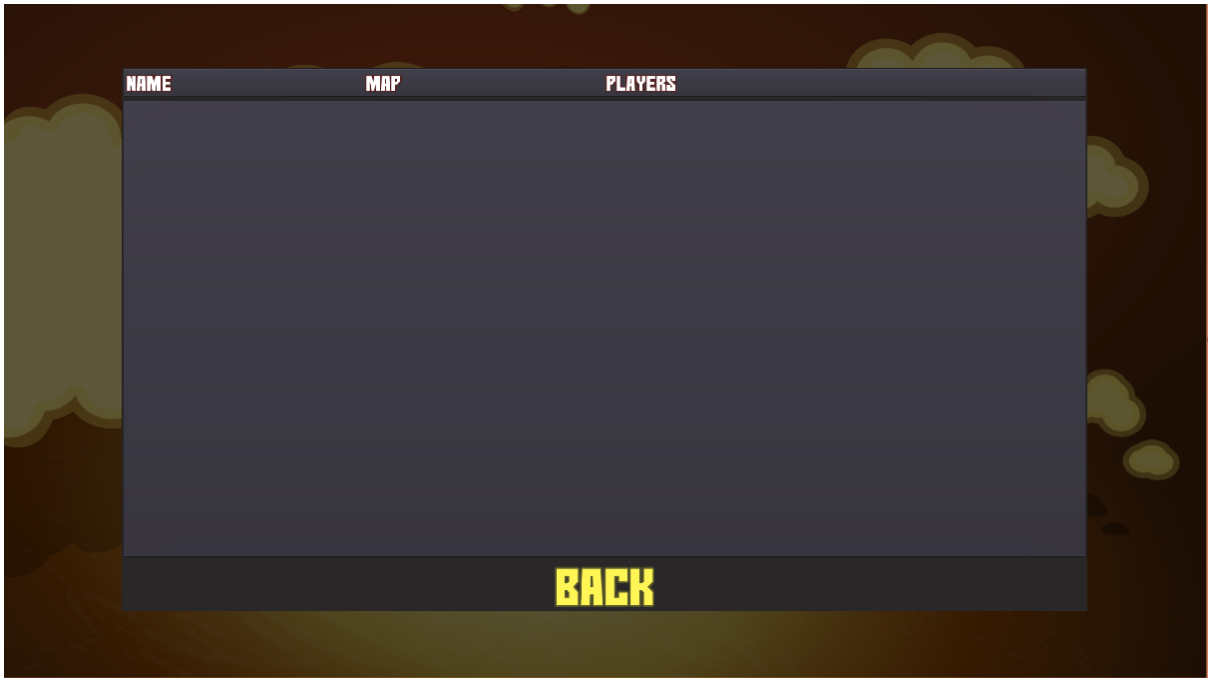

Abbildung 4.2

4.2 Struktur einer Reihe

Das Menü ist so noch nicht vollständig. Da es in Godot nicht so einfach möglich ist, eine Liste bzw. Reihen innerhalb einer Liste zu erstellen, bedarf es einer eigenen Szene, in der diese Funktionalität selbst zusammengebaut wird. Aus diesem Grund ist die Szene “row“ entstanden, die nun näher betrachtet wird.

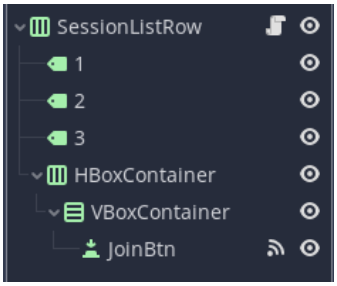

Abbildung 4.3

Abbildung 4.3 zeigt die Struktur hinter der “row”-Szene. Innerhalb des “HBoxContainers” namens “SessionListRow“ befinden sich drei Labels, die mit den Zahlen eins bis drei nummeriert sind. Hier werden später die Informationen zur Lobby dargestellt, nämlich der Lobbyname (1) die Map (2) und die Anzahl der Spieler (3). Diese Informationen werden dynamisch vom Server abgefragt. Ansonsten ist hier nur noch der Button “JoinBtn“ vorhanden, der sich innerhalb eines “HBoxContainers” und eines “VBoxContainers” befindet, damit dieser in der Reihe zentriert werden kann. Der Button hat die Aufschrift “Join“ und wird in jeder Reihe dargestellt. Mit einem Klick tritt der Spieler einer bestehenden Lobby bei.

4.3 Implementierung einer Reihe

```
func load_scene(var path):
  if ResourceLoader.exists(path):
    var ok = get_tree().change_scene(path)
    match ok:
      ERR_CANT_OPEN:
        print("Error: cant open " + path)
      ERR_CANT_CREATE:
        print("Error: cant create scene")

func _on_JoinBtn_pressed():
```

```
  var session_name = get_node("1").get_text()
  Server.join_session(session_name)
  load_scene(worldScene)
  pass # Replace with function body.
```

Abbildung 4.4

In Abbildung 4.4 sind die zwei Funktionen des “row”-Skripts zu sehen. Hier ist lediglich die Funktion “_on_JoinBtn_pressed“ hervorzuheben, die aufgrund eines Signals dann reagiert, wenn auf den Join-Button in einer Reihe geklickt wird. Hier wird zunächst der Session-Name aus dem Label ausgelesen und die Funktion “join_session“ des clientseitigen Server-Skripts aufgerufen. Da die Funktion “join_session“ lediglich die gleichnamige Funktion des serverseitigen Server-Skripts aufruft und ihr den Namen der Session und einen Zeitstempel mitgibt, wird auf die Abbildung verzichtet. Wie die “join_session“ Funktion des serverseitigen Server-Skripts den Spieler tatsächlich beitreten lässt, kann in der Dokumentation meines Kollegen Andreas eingesehen werden, der für die Client-Server-Kommunikation zuständig war.

4.4 Auslesen der Sessions

Die Abbildung 4.5 zeigt einen Ausschnitt aus dem Skript der “JoinExistingLobby“-Szene, deren Struktur bereits in Abbildung 4.1 vorgestellt wurde.

```
var row = preload("res://scenes/row.tscn")
onready var table = get_node("MenuBackground/VBoxContainer/PanelContainer2/ScrollContainer/VBoxContainer")
var row_count = 0

# Called when the node enters the scene tree for the first time.
func _ready():
  #sets icon size of itemList
  #for lobby in openLobbys:
    #itemList.add_item(lobby)
  Server.request_session_list()

func update_list(list):
  for x in range(0, list.size()):
    set_data(list[x])

func set_data(data:Dictionary):
  row_count += 1
  var instance = row.instance()
  instance.name=str(row_count)
  table.add_child(instance)

  #changing data of row
  get_node("MenuBackground/VBoxContainer/PanelContainer2/ScrollContainer/VBoxContainer/"+instance.name+"/1").text=data.name
  get_node("MenuBackground/VBoxContainer/PanelContainer2/ScrollContainer/VBoxContainer/"+instance.name+"/2").text=data.map
  get_node("MenuBackground/VBoxContainer/PanelContainer2/ScrollContainer/VBoxContainer/"+instance.name+"/3").text=data.players
```

Abbildung 4.5

Hier ist der Aufruf auf “request_session_list()“ des clientseitigen Server-Skripts hervorzuheben. In dieser Funktion wird eine Anfrage nach einer Liste gestellt, in der sich alle Sessions, also alle laufenden Spiele, befinden. Diese Liste wird serverseitig aufgebaut. Wie diese Liste aufgebaut wird, ist in Abbildung 4.6 zu sehen.

```
func get_session_list():
  var list = []

  for s in sessions.keys():
    var world = get_node(s + "/World")
    if not world.session_closed:
      max_players = world.get_node("TileMap").spawn_count
      var current_players = sessions[s].size()
      list.push_back({"name": s, "map": world.map ,"players": str(current_players) + "/" + str(max_players)})
  return list

remote func request_session_list():
  var list = get_session_list()
  rpc_id(get_tree().get_rpc_sender_id(), "recieve_session_list", list)
```

Abbildung 4.6

Die Liste wird hier in der Funktion “get_session_list()“ aufgebaut, indem die Informationen, die später in einer Reihe im Menü dargestellt werden sollen, aus den aktuell laufenden Sessions gelesen werden, also der Name, die Map, die Anzahl der aktuell verbundenen Spieler sowie die Anzahl der maximalen Spieler. Diese Liste wird dann an die Funktion “recieve_session_list“ an das clientseitige Server-Skript übergeben. Diese Funktion ruft lediglich die “update_list(list)“ Funktion des “JoinExistingLobby”-Skripts auf. Ausschnitte dieses Skripts konnten bereits in Abbildung 4.5 gesehen werden. Hier wird dann nur noch für jede Session eine Funktion aufgerufen, die dann eine neue “row”-Instanz erstellt und die drei Labels aus Abbildung 4.3 mit den richtigen Werten füllt. Außerdem wird dem “VBoxContainer” aus Abbildung 4.1, der sich in einem “ScrollContainer” befindet, für jede Session ein neuer “row”-Kindknoten erzeugt. Dadurch wird eine Liste mit allen aktuell laufenden Sessions aufgebaut. Das Endresultat des Menüs ist in Abbildung 4.7 zu sehen.

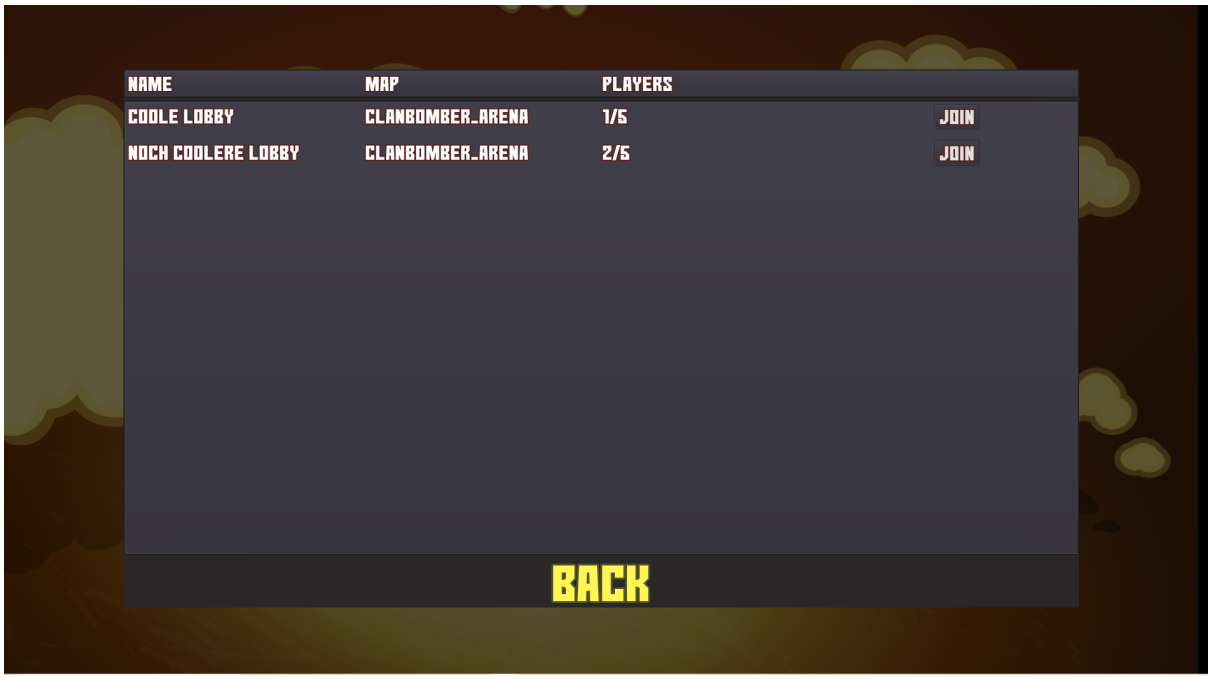

Abbildung 4.7

5. Erstellen des Ingame-Menüs und des Settings-Menüs

Die Spieler unseres Granatier-Clons sollen natürlich in der Lage sein, ein laufendes Spiel einfach verlassen zu können. Aus diesem Grund wurde ein Ingame-Menü benötigt, das der Spieler über die ESC-Taste aufrufen kann. Weiterhin soll der Spieler die Möglichkeit haben, einige Einstellungen zu ändern. Auch dafür wurde ein Menü erstellt. Dieses sogenannte Settings-Menü kann sowohl über das Ingame-Menü, als auch durch das Hauptmenü angesprochen werden.

5.1 Struktur des Ingame-Menüs

Das Ingame-Menü wird innerhalb einer laufenden Session über das Drücken der ESC-Taste aktiviert. Der Aufbau des Menüs ist in Abbildung 5.1 zu sehen, während das Aussehen des Menüs im Spiel in Abbildung 5.2 zu sehen ist.

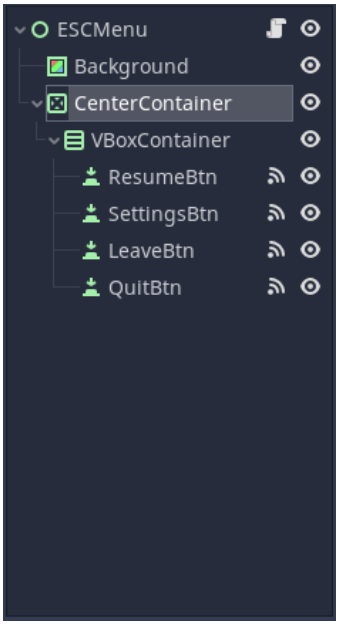

Abbildung 5.1

Der Aufbau des Ingame-Menüs ist recht simpel. Der oberste Knoten ist vom Typ “Control” und ist der Elternknoten des Menüs. Darin enthalten ist der “Background” vom Typ “ColorRect”, der lediglich dafür sorgt, dass das im Hintergrund befindliche Spiel etwas abgedunkelt wird und sich der Spieler auf das Menü konzentrieren kann (siehe auch Abbildung 5.2). Ebenfalls enthalten ist ein “CenterContainer”, der dafür sorgt, dass der sich darin enthaltene “VBoxContainer” mittig am Bildschirm positioniert. Der “VBoxContainer” ist ein Kindknoten des “CenterContainers”. Dadurch werden die Buttons des Menüs (”ResumeBtn”, “SettingsBtn”…) untereinander platziert.

Abbildung 5.2

5.2 Implementierung des Ingame-Menüs

Die Abbildung 5.3 zeigt einen Ausschnitt des Codes des Ingame-Menüs, der sich in der Datei „ESCMenu.gd“ befindet.

```
func _ready():
  visible = false
```

```
func _unhandled_input(_event):
  if Input.is_action_just_pressed("openESCMenu"):
    changeStatus()

func changeStatus():
  visible = !visible
  if get_tree().get_nodes_in_group("LocalPlayers").has(0):
    var local_player = get_tree().get_nodes_in_group("LocalPlayers")[0]
    local_player.settings_open = !local_player.settings_open

func _on_ResumeBtn_pressed():
  changeStatus()

func _on_QuitBtn_pressed():
  get_tree().quit()

func _on_LeaveBtn_pressed():
  Server.leave_session()
  load_scene(mainMenuScene)

func _on_SettingsBtn_pressed():
  $CenterContainer.visible = false
  var settingsInstance = settings.instance()
  $Background.add_child(settingsInstance)
  settingsInstance.visible = true
  settingsInstance.connect("tree_exited", self, "_on_SettingsMenu_tree_exited")

func _on_SettingsMenu_tree_exited():
  $CenterContainer.visible = true
```

Abbildung 5.3

Die “_ready()”-Funktion wird immer dann ausgeführt, wenn die Node die Szene zum ersten Mal betritt, also im Endeffekt dann, wenn der Spieler einem Spiel beitritt. Hier wird das Menü unsichtbar gemacht, da man es zu Beginn nicht sehen soll. Die Funktion “_unhandled_input(event)” ruft die “changeStatus()”-Funktion auf, wenn der Spieler die ESC-Taste drückt und somit das Menü öffnen möchte. Die “changeStatus()”-Funktion ändert die Sichtbarkeit des Ingame-Menüs und gibt den aktuellen Status (also ob das Menü geöffnet ist oder nicht) an den “LocalPlayer” weiter. Die Funktion “_on_ResumeBtn_pressed()” wird durch ein Signal jedes Mal aufgerufen, wenn der Spieler auf den Resume-Button klickt und ruft auch nur “changeStatus()” auf. Die Funktion “_on_QuitBtn_pressed()” wird durch Klicken auf den Quit-Button ausgelöst und schließt das Spiel, in dem “quit()” auf den Szenenbaum aufgerufen wird. Die Funktion “_on_LeaveBtn_pressed()” soll dafür sorgen, dass der Spieler die aktuelle Session verlässt, wenn er auf den Leave-Button klickt. Dafür muss die “leave_session()”-Funktion des clientseitigen Server-Skripts aufgerufen werden. Die “leave_session()”-Funktion leitet die Anfrage lediglich an das serverseitige Server-Skript weiter. Die serverseitige Implementierung wurde von meinem Kollegen Andreas übernommen und kann in dessen Dokumentation nachverfolgt werden. Dann wird nur noch die “load_scene”-Funktion aufgerufen, die das Hauptmenü laden und darstellen soll. Wie die “load_scene”-Funktion aussieht, kann in Abbildung 4.4 gesehen werden. Die Funktion “_on_SettingsBtn_pressed()” wird ausgeführt, wenn der Spieler auf den Settings-Button klickt und muss zunächst das Ingame-Menü unsichtbar machen. Dann wird eine Instanz des Settings-Menü erstellt, als Kindknoten hinzugefügt und sichtbar gemacht. Am Ende wird ein Signal erstellt, das die Funktion “_on_SettingsMenu_tree_exited()” aufruft, wenn das Settings-Menü den Szenenbaum verlässt, also „zerstört“ wird. Das wird dann der Fall sein, wenn der Spieler auf den „Back“-Button des Settings-Menüs klickt. Die “_on_SettingsMenu_tree_exited()”-Funktion macht das Ingame-Menü dann wieder sichtbar.

5.3 Struktur des Settings-Menüs

Die Abbildung 5.4 zeigt den Aufbau des Settings-Menüs.

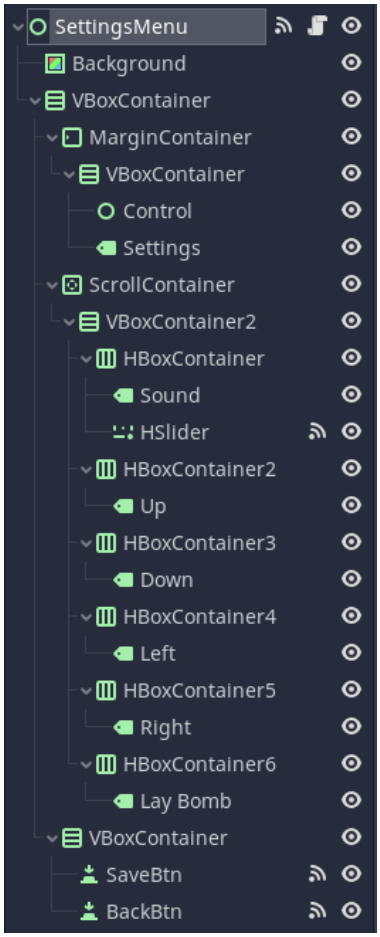

Abbildung 5.4

Auch hier wurde als Elternknoten ein Knoten des Typ “Control” mit dem Namen “SettingsMenu” gewählt. Der “Background” vom Typ “ColorRect” ist auch hier dafür zuständig, den Hintergrund abzudunkeln, damit sich der Spieler besser auf das Menü konzentrieren kann. Der “VBoxContainer” beinhaltet alle restlichen Elemente. Diese werden durch ihn untereinander dargestellt. Der “MarginContainer” zusammen mit dem im “VBoxContainer” befindlichen “Control”-Node werden dafür verwendet, damit das “Settings”-Label, das die Settings-Überschrift darstellt, nicht so sehr am oberen Bildschirmrand „klebt“. Innerhalb eines “ScrollContainer”, der, wie der Name schon sagt, seine Kindknoten „scrollbar“ macht, befindet sich unter anderem der “VBoxContainer2”, der wiederum viele “HBoxContainer” als Kindknoten hat. In den “HBoxContainern” sind nur Labels verpackt. “HBoxContainer” ordnen ihre Kindknoten horizontal nebeneinander an. Dies macht auch bei den einzelnen Labeln in den “HBoxContainern” Sinn, da hier noch Buttons dynamisch aus dem Code geladen und neben den Labels angezeigt werden, wie später zu sehen sein wird. Der erste “HBoxContainer” hat eine Besonderheit, da hier statt einem Button direkt ein “HSlider” angezeigt wird, was ein Slider ist, mit dem die Lautstärke des Spiels geändert werden kann. Der letzte “VBoxContainer” beinhaltet noch die beiden Buttons „Save“ und „Back“.

5.4 Implementierung des Settings-Menüs

In Abbildung 5.5 wird der erste Teil des Skripts des Settings-Menüs dargestellt, welches sich in der Datei „SettingsMenu.gd“ befindet.

```
func _ready():
  visible = false
  keybinds = Keybinds.keybinds.duplicate()
  var counter = 0
  volume = AudioServer.get_bus_volume_db(AudioServer.get_bus_index("Master"))
  soundSlider.value = volume
  for key in keybinds:
    var button = Button.new()

    button.set_h_size_flags(Control.SIZE_EXPAND_FILL)
```

```
    var button_value = keybinds[key]

    if button_value != null:
      button.text = OS.get_scancode_string(button_value)
    else:
      button.text = "Undefined"

    button.set_script(buttonScript)
    button.key = key
    button.value = button_value
    button.menu = self
    button.toggle_mode = true
    button.focus_mode = Control.FOCUS_NONE

    var font = DynamicFont.new()
    font.font_data = load("res://resources/fonts/gomarice_no_continue.ttf")
    font.size = 40
    font.outline_size = 3
    font.outline_color = Color("572222")
    button.set("custom_fonts/font", font)
    button.set("custom_colors/font_color", Color("c8c8c8"))

    var containerNumber = 2+counter
    var container = get_node("VBoxContainer/ScrollContainer/VBoxContainer2/HBoxContainer" + str(containerNumber))
    counter = counter + 1

    container.add_child(button)
    buttons[key] = button
```

Abbildung 5.5

In der “_ready()”-Funktion wird das Settings-Menü zunächst aufgebaut, in dem die aktuellen Keybinds (also die aktuelle Tastenbelegung) und die aktuelle Lautstärkeeinstellung durch den von Godot mitgelieferten “AudioServer” ausgelesen wird. Des Weiteren werden hier die Buttons dynamisch erstellt und dem Szenenbaum hinzugefügt, sowie deren Aussehen z.B. durch Farben und Schriftarten festgelegt. Der Text des Buttons wird aus den Keybinds ausgelesen und eingetragen, da die Buttons die aktuelle Tastaturbelegung anzeigen sollen. In Abbildung 5.8 kann das Keybinds-Skript gesehen werden. Die hier generierten Buttons haben auch ein eigenes Skript, das in Abbildung 5.7 gezeigt und erläutert wird.

```
func change_keyBind(key, value):
  keybinds[key] = value
  for k in keybinds:
    if k != key and value != null and keybinds[k] == value:
      keybinds[k] = null
      buttons[k].value = null
      buttons[k].text = "Undefined"

func _on_BackBtn_pressed():
  queue_free()

func _on_SaveBtn_pressed():
  Keybinds.keybinds = keybinds.duplicate()
  Keybinds.set_movement_keys()
  Keybinds.write_keyBinds_config()
  if volume > soundSlider.min_value:
    AudioServer.set_bus_mute(AudioServer.get_bus_index("Master"), false)
    AudioServer.set_bus_volume_db(AudioServer.get_bus_index("Master"), volume)
  else:
    AudioServer.set_bus_mute(AudioServer.get_bus_index("Master"), true)
  _on_BackBtn_pressed()

func _on_HSlider_value_changed(value):
  volume = value
```

Abbildung 5.6

Die Abbildung 5.6 zeigt den zweiten Teil des Settings-Menü Skripts. Die Funktion “change_keyBind(key, value)” wird aus dem Button-Skript (Abbildung 5.7) aus aufgerufen, sobald der Spieler eine Tastenbelegung ändern möchte. Die erste Zeile der Funktion kümmert sich dabei um die (zumindest temporäre) Neubelegung, während in der Schleife überprüft wird, ob die neu ausgewählte Taste schon für etwas Anderes genutzt wurde. Falls dies der Fall ist, wird die Tastenbelegung der vorherig genutzten Funktionalität entfernt und der dazugehörige Button zeigt den Text „Undefined“ an, damit der Nutzer weiß, dass er diese Tastenbelegung jetzt auch ändern muss. Die Funktion

“_on_BackBtn_pressed()” wird bei einem Klick auf den Back-Button des Settings-Menüs ausgelöst und beendet das Settings-Menü. Mit der Funktion “_on_SaveBtn_pressed()” werden die neuen Tastenbelegungen und Lautstärkeeinstellungen des Spielers endgültig gespeichert, nachdem dieser auf den Save-Button geklickt hat. Die Tastenbelegungen werden dabei in die Konfigurationsdatei geschrieben, die in Abbildung 5.9 zu sehen ist, während die Lautstärkeeinstellungen im “AudioServer” abgespeichert werden. Außerdem wird noch “_on_BackBtn_pressed()” aufgerufen, weil sich das Settings-Menü nach dem Speichern der Einstellungen direkt schließen soll. Die letzte Funktion “_on_HSlider_value_changed(value)” wird jedes Mal aufgerufen, wenn sich der Wert des Sliders ändert und speichert diese Lautstärke vorübergehend ab.

```
extends Button

var key
var value
var menu

var waiting_for_input = false

func _input(event):
  if waiting_for_input:
    if event is InputEventKey:
      value = event.scancode
      text = OS.get_scancode_string(value)
      menu.change_keyBind(key, value)
      pressed = false
      waiting_for_input = false
    if event is InputEventMouseButton:
      if value != null:
        text = OS.get_scancode_string(value)
      else:
        text = "Undefined"
      pressed = false
      waiting_for_input = false

func _toggled(button_pressed):
  if button_pressed:
    waiting_for_input = true
    set_text("Press any key...")
```

Abbildung 5.7

In Abbildung 5.7 ist das Skript für die Buttons des Settings-Menüs zu sehen, das sich in der Datei „KeyButton.gd“ befindet und aus zwei Funktionen besteht. Die Funktion “_input(event)” wird aufgerufen, sobald eine Art von Input des Spielers passiert. Handelt es sich dabei um das Drücken einer Taste, wird diese Taste (zumindest vorübergehend) als neue Tastenbelegung für eine Aktion durch die Funktion “change_keyBind(key, value)” aus Abbildung 5.6 gespeichert und der Text des Buttons dementsprechend aktualisiert. Wird jedoch die Maus geklickt, so wird die Taste im Button angezeigt, die davor schon belegt war, außer es gab keine Belegung dazu, dann wird „Undefined“ im Button angezeigt. Die Funktion “_toggled(button_pressed)” wird ausgeführt, sobald auf einen Button im Settings-Menü geklickt wurde. Die Variable “waiting_for_input” wird dann auf true gesetzt, damit die “_input(event)” Funktion weiß, welcher Button auf Input wartet und somit der richtige Button aktualisiert wird. Außerdem wird der Text auf „Press any key…“ gesetzt.

```
var keybinds = {"move_up": 87, "move_down": 83, "move_left" : 65, "move_right" : 68, "place_bomb" : 81}

func _ready():
  configfile = ConfigFile.new()
  if configfile.load(filepath) == OK:
    for key in configfile.get_section_keys("keybinds"):
      var value = configfile.get_value("keybinds", key)

      if str(value) != "":
        keybinds[key] = value
      else:
        keybinds[key] = null
  else:
    configfile.save(filepath)

  set_movement_keys()

func set_movement_keys():
  for key in keybinds:
    var val = keybinds[key]
```

```
    var currentInputMap = InputMap.get_action_list(key)

    if !currentInputMap.empty():
      InputMap.action_erase_event(key, currentInputMap[0])

    if val != null:
      var new_key = InputEventKey.new()
      new_key.set_scancode(val)
      InputMap.action_add_event(key, new_key)

func write_keyBinds_config():
  for key in keybinds:
    var key_val = keybinds[key]
    if key_val != null:
      configfile.set_value("keybinds", key, key_val)
    else:
      configfile.set_value("keybinds", key, "")
  configfile.save(filepath)
```

Abbildung 5.8

Die Abbildung 5.8 zeigt das “Keybind”-Skript, welches unter „Keybinds.gd“ abgespeichert ist. Dieses Skript kümmert sich hauptsächlich um das Auslesen und das Speichern der Tastenbelegungen. In der “_ready()”-Funktion wird die Tastenbelegung aus der Konfigurationsdatei (Abbildung 5.9) ausgelesen und abgespeichert. Sollte die Datei nicht gefunden werden, wird eine neue Datei erstellt und unter einem festgelegten Pfad abgespeichert. In beiden Fällen wird die “set_movement_keys()”-Funktion aufgerufen. Diese Funktion kümmert sich um das Aktualisieren der Godot “InputMap”. Die “InputMap” ist ein eingebautes Objekt in Godot, in dem die Tastenbelegungen für das Spiel abgespeichert werden können. Nach jedem Aufruf des Settings-Menüs wird also zunächst auch die “InputMap” aktualisiert, sollte sich etwas in der Konfigurationsdatei geändert haben. Die letzte Funktion des “Keybind”-Skripts ist die “write_keyBinds_config()”-Funktion. Diese Funktion kümmert sich um das tatsächliche Abspeichern der Tastenbelegungen in die Konfigurationsdatei. Sollte eine Aktion nicht belegt sein, wird als Taste ein leerer String gespeichert und diese Aktion wird für den Spieler dann nicht ausführbar sein.

```
[keybinds]

move_up=87
move_down=83
move_left=65
move_right=68
place_bomb=81
```

Abbildung 5.9

Die Abbildung 5.9 zeigt, wie die Konfigurationsdatei für die Tastenbelegung aussieht. Dabei handelt es sich um eine .ini-Datei. Das [keybinds] zu Beginn ist sehr wichtig, da man dadurch im Code auf die einzelnen Elemente, also die Tastenbelegungen zugreifen kann. Die Namen, die links vom Gleichheitszeichen stehen, müssen dabei genau gleich definiert sein, wie in der “InputMap” in Godot (siehe Abbildung 5.10), weil diese sonst nicht richtig zugeordnet werden können. Die Zahlen auf der rechten Seite des = sind die sogenannten Scancodes der Tasten. Diese werden im Code des Öfteren in deren String-Repräsentation durch “OS.get_scancode_string()” umgewandelt.

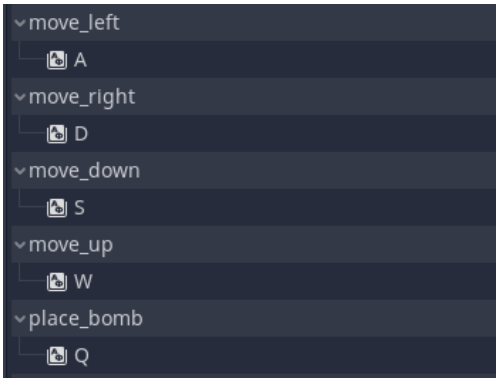

Abbildung 5.10

Abbildung 5.10 zeigt die Tastenbelegung in Godots “InputMap”. Man kann also sehen, dass die Namensgebung hier mit der Namensgebung in der Konfigurationsdatei übereinstimmt.

Abbildung 5.11

In Abbildung 5.11 ist das Aussehen des Settings-Menüs im Spiel zu sehen.