Dokumentation von Andreas Riegg

Teamname	SoulThieves
Rolle(n)	Entwickler

Aufgaben:

- Grundgerüst für Player
- Client/Server Implementierung

Inhaltsverzeichnis:

1 Systemdokumentation

1.1 Allgemeines

Das Spiel Granatier Online basiert auf dem Spiel Granatier von KDE.
(https://apps.kde.org/de/granatier/)
Alle Ressourcen, wie Texturen und Sounds, stammen von dort.

Es werden sowohl Windows als Mac OSX und Linux unterstützt.

Der Multiplayer unterstützt auch das Zusammenspielen von Nutzern unterschiedlicher Betriebssysteme.

Der Server unterstützt allerdings ausschließlich Linux, bis auf Weiteres steht ein öffentlicher Server unter *granatier.soulthieves.tk* zur Verfügung, die Verfügbarkeit dieses Servers wird allerdings nicht garantiert.

1.2 Engine

Die Basis für das Spiel bildet die Godot-Engine.

Godot ist eine Game-Engine, mit der sowohl 2D- als auch 3D-Spiele entwickelt werden können.

Sie bietet einen umfassenden Satz von Werkzeugen zu Spielentwicklung, die es Entwicklern ermöglichen, sich auf das Spiel zu konzentrieren, ohne die Grundlagen alle selbst entwickeln zu müssen.

Das Exportieren für verschiedenste Betriebssysteme ist sehr einfach möglich.
Es werden natürlich Mac OS, Windows und Linux unterstütz, aber auch Apps für Android und IOS sowie HTML5-Browsergames können entwickelt werden.

Dabei ist die Engine völlig kostenlos und quelloffen, lizenziert ist sie unter der MIT-Lizenz.

Strukturiert werden Spiele hier in sogenannte Szenen, welche aus mitgelieferten Nodes zusammengebaut und ineinander verschachtelt werden können.

Das Verhalten dieser Szenen wird mittels Scripts gesteuert.

Diese Scripts werden in der, der Engine eigenen, Sprache GDscript geschrieben, die syntaktisch an Python angelehnt ist.

Optional gibt es auch eine Godot-Version, die C# als Programmiersprache unterstützt, diese wurde allerdings nicht verwendet, da die dies die Kompatibilität mit allen Betriebssystemen außer Windows beeinträchtigt hätte.

1.3 Netzwerkkommunikation

Es handelt sich um ein Multiplayer-Spiel.
Die Netzwerkkommunikation findet über die Godot-eigene Multiplayer-API statt.

Diese sendet die Daten per UDP über einen festgelegten Port, hier wurde der Port 50000 gewählt.

Für weitere Informationen zu Netzwerkkommunikation siehe 4 Multiplayer-Implementierung.

2 Client-Server-Architektur

2.1 Grundlage der Architektur

Um die Multiplayer-Implementierung dieses Spiels zu erklären, muss zunächst das theoretische Konzept beschrieben werden, dass dieser zugrunde liegt.

Die Implementierung des Multiplayers und der Kommunikation zwischen Server und Client ist stark von der Artikel-Serie Fast-Paced Multiplayer von Gabriel Gambetta inspiriert.

Wie in dem Artikel beschrieben, wurde ein autoritativer Server entwickelt, der die Aktionen jedes Spielers durchführt und das korrekte Ergebnis an alle Clients sendet. Die Clients korrigieren gegebenenfalls den Zustand der Spieler und der Map, sollten sie von den Informationen des Servers abweichen.

2.2 Client-Side Prediction

Das Prinzip eines autoritativen Servers setzt voraus, das jede Aktion und generell jede Änderung des Spiel-Zustands vom Server durchgeführt wird.

So wird verhindert, dass ein Spieler mit einem Cheat das tatsächliche Spiel beeinflussen kann.
Da jede Aktion vom Server ausgeführt wird, kann ein betrügerischer Spieler höchstens das ändern, was er selbst sieht, nicht aber das Spiel der anderen Spieler.

Diese Vorsichtsmaßnahme führt allerdings auch zu einem Problem. Wenn alle Aktionen vom Server ausgeführt werden müssen, dann entsteht zwangsweise eine gewisse Verzögerung zwischen dem

Tastendruck des Spielers und der tatsächlichen Reaktion im Spiel. Dies kommt zustande, weil es immer etwas Zeit in Anspruch nimmt, die Aufforderung an den Server zu senden und dessen Antwort zu empfangen. (Siehe Abbildung 3, rechts)

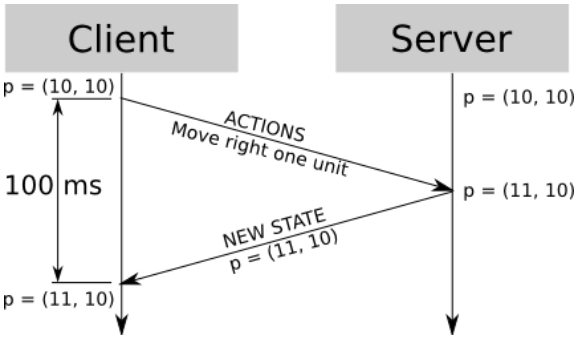

Abb. 1: Schaubild Netzwerklatenz

Um diese Verzögerung zu verhindern und ein flüssiges Spielerlebnis zu ermöglichen, muss der Client also vorhersehen, welche Änderungen am Server durchgeführt werden und sie anwenden, bevor ihn die Antwort des Servers erreicht.

Dies wird erreicht, indem die Aktion tatsächlich zunächst am Client durchgeführt wird.
Der Server sendet dann regelmäßig den aktuellen Zustand der gesamten Welt an die Clients, die sich bei unterschieden immer an den Server anpassen.

2.3 Server Reconciliation

Der im letzten Abschnitt beschriebene Vorgang führt allerdings zu einem anderen Problem.
Wenn der Client nämlich die Aktion zum Beispiel eine Bewegung des Spielers ausführt, bevor der Server dies tut, kann es dazu kommen, dass er zu dem Zeitpunkt, zu dem die Antwort des Servers mit dem aktuellen Zustand der Welt erhält, bereits eine weitere Bewegung vollzogen hat.

In diesem Fall würde der Spieler sich also zunächst bewegen, wenn aber die Antwort des Servers eintrifft, würde er zu der Stelle zurückgesetzt werden, an der er sich vor der letzten Bewegung befunden hat. Erst wenn der Server diese neue Bewegung erhalten hat, würde der Spieler wieder auf die Position nach der Bewegung gesetzt werden.

Nebenstehende Abbildung veranschaulicht dieses Problem. Nachdem der Client bereits zwei Bewegungen durchgeführt hat und sich an Position (12, 10) befindet, erhält er die Information von Server, dass die korrekte Position (11, 10) ist. Dies führt

dazu, dass der Spieler zunächst auf Position (11, 10) zurückgesetzt würde, bevor das nächste Update von Server eintrifft und der Spieler wieder auf (12,10) platziert wird.

Dieses Synchronisationsproblem lässt sich durch die sogenannte *“Server Reconciliation”*, also die Versöhnung des Clients mit dem Server lösen.

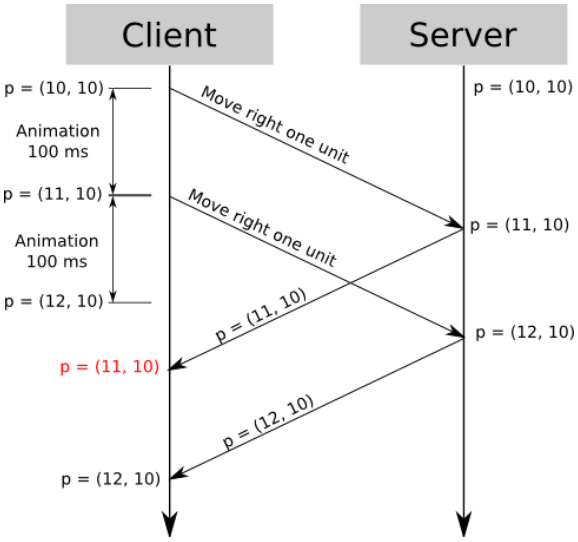

Abb. 2: Synchronisationsproblem

Hier wurde eine etwas andere Implementierung gewählt als die, die der Artikel vorschlägt.
Der Grundgedanke besteht darin, dass man bei jedem Update durch den Server wissen muss, welche Aktionen am Server bereits durchgeführt wurden und welche bisher nur der Client ausführen konnte, da der Server sie erst nach versenden des Updates erhalten hat.

IIn dem Artikel wird vorgeschlagen, sämtliche Aktionen mit einem Index zu versehen. Der Server fügt dann mit dem Update den Index der letzten ausgeführten Aktion bei.

Der Client muss dann alle ausgeführten Aktionen speichern. Wenn ein Update des Servers eintrifft, löscht er die Aktionen, die am Server bereits durchgeführt wurden, und führt die übrigen Aktionen auf Basis der vom Server übermittelten Daten erneut durch. Die Daten vom Server befinden sich in der Vergangenheit und können so rückwirkend anstelle der Daten eingesetzt werden, die der Client vorhergesehen hat.

Da dieses Vorgehen in Godot aber schwer umzusetzen ist, wurde hier ein etwas anderes Vorgehen gewählt.

Statt jedes Mal die Position des Spielers auf Basis der Serverdaten neu zu berechnen, wird die Position des Spielers nur dann an das Update des Servers angepasst, wenn es neuer ist als die Daten des Clients. Das bedeutet, dass die Position des Spielers nur dann mit dem Server abgeglichen wird, wenn er kurz stehen bleibt, der Nutzer also für mindestens einen Frame (ein Sechzigstel einer Sekunde) keine Taste drückt.

Auch wenn dieses Vorgehen dazu führt, dass der Client nicht ganz so exakt mit den Daten des Servers abgeglichen wird, trifft diese Situation häufig genug ein, um eine ausreichende Synchronizität mit dem Server zu gewährleisten, solange der Spieler nicht versucht zu betrügen. Auf diese Weise ist es möglich, ein flüssiges Spiel zu garantieren, ohne die aufwendigen Berechnungen nachzuvollziehen und nachzuahmen, die bei Godot im Hintergrund durchgeführt werden.

2.4 Entity Interpolation

Der Server kann sehr viele Anweisungen von vielen Spielern gleichzeitig erhalten, um den Server nicht zu überlasten, wird allerdings nicht jedes Mal ein Update an alle Clients gesendet. Stattdessen sendet der Server periodische Updates, die für gewöhnlich die Ergebnisse mehrerer Aktionen enthalten. In unserem Fall sendet der Server dreißigmal die Sekunde ein Update.

Das Problem, dass dadurch entsteht, betrifft die anderen Spieler, gegen die man spielt.
Diese Spieler senden ihre Aktionen an den Server, der sie wiederum an alle anderen Clients weitergeben muss.

Da der Server diese Daten allerdings nur Periodisch versendet würde dies dazu führen, dass sich alle anderen Spieler “ruckelig” bewegen.

Wie Abbildung 5 veranschaulicht würde der Spieler, der durch Client 1 gesteuert wird, aus Sicht von Client 2 stillstehen, bis ein Update von Server eintrifft.

Bei Eintreffen dieses Updates würde der Spieler einen abrupten Sprung zur neuen Position vollziehen und im Anschluss wieder bis zum nächsten Update stillstehen.

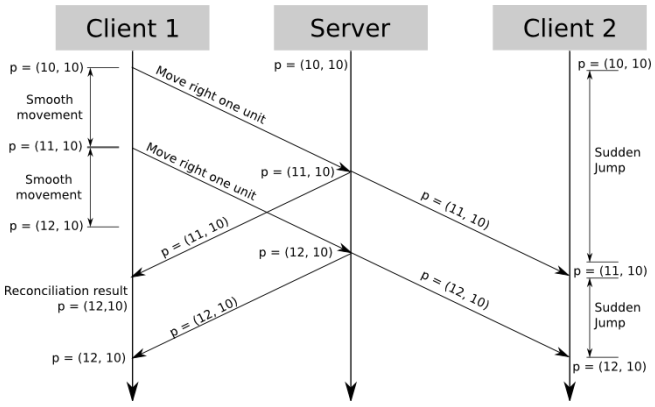

Abb. 3 Update-Problem: Client 2 aus sicht von Client 1

Dieses Problem lässt sich, wie im Artikel beschrieben auf verschiedene Weise lösen. Da sich die Bewegung eines Spielers allerdings unmöglich voraussagen lassen, ist

die einzige hier anwendbare Lösung die sogenannte “Entity Interpolation”.

Der Grundgedanke besteht hier darin, dass die Position der anderen Spieler nicht in der Gegenwart angezeigt werden, sondern minimal verzögert, sodass man die Spieler an der Position sieht, wo sie vor einigen Millisekunden waren.

Wenn der Client ein Update von Server erhält, wird die Position der Mitspieler in einer Warteschlange gespeichert. Nach Ablauf einer geringen Verzögerung wird der jeweilige Spieler dann in Richtung der ersten Position in dieser Warteschlange beweg. Ist diese Position erreicht, bewegt sich der Spieler in Richtung der nächsten Position und so weiter.

Durch die Verzögerung wird gewährleistet, dass trotz der periodischen Updates eine flüssige Bewegung zustande kommen kann. Diese Verzögerung ist dabei so gering, dass sie das Spiel nicht nennenswert beeinträchtigt und nur im direkten Vergleich wahrnehmbar ist. Details zur Umsetzung werden in 3.3 OnlinePlayer erklärt.

3 Grundaufbau Player

3.1 Vererbungsstruktur

Es wurden zwei separate Szenen für Spieler erstellt.
Die Szene LocalPlayer stellt den lokalen Spieler dar, der vom Nutzer über die Tastatur gesteuert wird.
Alle anderen Mitspieler, die von Nutzern an anderen Geräten gesteuert und über den Server synchronisiert werden, werden durch die OnlinePlayer-Szene dargestellt.

Diese beiden Szenen sind dabei vollkommen identisch und nur das jeweils hinterlegte Script zu Steuerung ist ein anderes.

Da aber einige Gemeinsamkeiten zwischen diesen beiden Scripts bestehen, wurde ein Basis-Script erstellt, von dem die beiden Spieler-Scripts erben.

Dieses Player-Script enthält im Wesentlichen die Statuswerte des Spielers und wird in der Dokumentation eines anderen Team-Mitglieds genauer erwähnt werden.

Die beiden abgeleiteten Scripts sind für alle Aktionen der Spieler wie Laufen oder eine Bombe setzen zuständig und werden im Folgenden genauer erklärt.

3.2 LocalPlayer

Die wichtigste Funktion in Player ist _physics_process.

Alle übrigen Methoden des Players sind für diese Dokumentation nicht relevant, da sie von anderen Gruppenmitgliedern entwickelt wurden.

Es handelt sich hierbei um eine vordefinierte Funktion von Godot, die jeden Frame einmal aufgerufen wird.

In dieser Funktion werden die Eingaben des Spielers verarbeitet und an den Server gesendet.

Zunächst wird überprüft, ob der Spieler bereits gestorben ist, die Einstellungen offen sind oder der Spieler gerade herunterfällt.
In all diesen Fällen wird die Funktion sofort verlassen, da der Spieler sich dann nicht bewegen und auch sonst keine Aktionen tätigen kann.

Wie in der folgenden Abbildung zu sehen, wird danach jeweils überprüft, ob einen der relevanten Tasten gedrückt wurde.
Die Funktion *is_acion_just_pressed* wird von der Engine bereitgestellt. Sie gibt dann “true” zurück, wenn die jeweilige Taste in diesem Frame neu gedrückt wurde, also im vorherigen Frame noch nicht gedrückt war.

Dabei wird nicht explizit die Taste angegeben, die überprüft werden soll, sonder eine “Aktion”, der über eine Konfigurationsdatei eine Taste zugewiesen wird.
Auf diese Weise lässt sich die Tastenbelegung leicht ändern.

Wenn eine der überprüften Tasten gedrückt wurde, dann werden jeweils bestimmten Variablen Werte zugewiesen.

So wird beispielsweise die Variable *motion* gesetzt.
Bei *motion* handelt es sich um einen Vektor, der die derzeitige Bewegungsrichtung des Spielers angibt.

Ein anderer wichtiger Wert, der hier gesetzt wird, ist die *rotation*, diese gibt an, wie die Textur des Spielers ausgerichtet ist. Dies sorgt dafür, dass der Spieler auch optisch in die Richtung schaut, in die er läuft.

Die anderen Variablen, die hier gesetzt werden, sind für die Anwendung von Powerups notwendig und werden vermutlich in der Dokumentation eines anderen Teammitglieds weiter ausgeführt.

Eine Ausnahme bietet hierbei die Variable *last_pressed*, die jeweils die zuletzt gedrückte Aktion speichert.
Dies ist entscheidend, um überprüfen zu können, ob diese Taste losgelassen wurde und die Bewegung somit aufhören muss. Solch eine separate Behandlung des Stehenbleibens ist deshalb notwendig, weil es vorkommen kann, dass eine weitere Taste gedrückt wird, bevor die zuerst gedrückte Taste losgelassen wird.

Hier wird überprüft, ob der Spieler sich auf Eis befindet und dementsprechend weiterrutschen muss, dies wurde allerdings von einem anderen Teammitglied implementiert.

Der einzige Zweig der If-Verzweigung, der bisher noch nicht erwähnt wurde, ist die Überprüfung der “bomb_placed” Aktion.
Wenn diese ausgelöst wird, wird sie einfach an den Server weitergegeben, ob eine Bombe gesetzt wird, entscheidet der Server (siehe:4.2 Client) .

```
45 func _physics_process(_delta):
46     #if stats.is_dead or can_move:
47     if stats.is_dead or settings_open or stats.fallen:
48         return
49     if Input.is_action_just_pressed("move_right"):
50         last_pressed = "move_right"
51         motion = Vector2.RIGHT
52         viewing_direction = motion
53         rotation = 0
54         invert_rot = fmod(rotation + PI, 2 * PI)
55     elif Input.is_action_just_pressed("move_down"):
56         last_pressed = "move_down"
57         motion = Vector2.DOWN
58         viewing_direction = motion
59         rotation = PI/2
60         invert_rot = fmod(rotation + PI, 2 * PI)
61     elif Input.is_action_just_pressed("move_left"):
62         last_pressed = "move_left"
63         motion = Vector2.LEFT
64         viewing_direction = motion
65         rotation = PI
66         invert_rot = fmod(rotation + PI, 2 * PI)
67     elif Input.is_action_just_pressed("move_up"):
68         last_pressed = "move_up"
69         motion = Vector2.UP
70         viewing_direction = motion
71         rotation = 1.5*PI
72         invert_rot = fmod(rotation + PI, 2 * PI)
73     elif Input.is_action_just_pressed("place_bomb"):
74         Server.place_bomb()
75     elif not last_pressed == null and Input.is_action_just_released(last_pressed):
76         if not stats.on_ice:
77             motion = Vector2.ZERO
78         else:
79             motion /= 2
```

Abb. 4 Ausschnitt aus LocalPlayer.gd: *_physics_process*-Funkiton

Nach einigen Überprüfungen, die ebenfalls das bereits erwähnte Eis betreffen, wird die eigentliche Bewegung durchgeführt.

```
114  if motion.length() > 0:
115      var amplifier = 1
116      if stats.hyperactive:
117          amplifier = 4
118      elif stats.slow:
119          amplifier = 0.5
120      elif stats.has_mirror:
121          amplifier = -1
122          rotation = invert_rot
123      if stats.on_ice != Vector2.ZERO:
124          if motion == Vector2.ZERO:
125              move_and_slide(stats.on_ice * stats.speed * amplifier)
126          else:
127              amplifier *= 2
128
129      var _v = move_and_slide(motion * stats.speed * amplifier)
130      Server.move_player(motion)
131      $AnimatedSprite.play()
132  else:
133      $AnimatedSprite.stop()
134      $AnimatedSprite.frame = 0
```

Abb. 5 Ausschnitt aus LocalPlayer.gd: *_physics_process*-Funkiton

Hier wird überprüft, ob der *motion*-Vektor länger als null ist, der Spieler sich also bewegt.

Ist das der Fall, so wird ein *amplifier* abhängig davon gesetzt, ob der Spieler von einem temporären Effekt betroffen ist, den die Bewegung beeinflusst.

Schließlich wird die eigentliche Bewegung mittels *move_and_slide* durchgeführt. Übergeben wird ein Vektor, der aus dem vorher bestimmten *motion*-Vektor, der derzeitigen Geschwindigkeit des Spielers und dem *amplifier* errechnet wird.

Der Server wird daraufhin aufgefordert, diese Bewegung ebenfalls durchzuführen, er wird das Ergebnis ggf. korrigieren, um Cheats zu vermeiden. (Siehe 2.2 Client-Side Prediction)

Die *play*-Methode des *AnimatedSprite* führt die Laufanimation des Spielers aus, wenn der Spieler sich nicht bewegt, wird die Animation im Else-Zweig wieder

angehalten.

3.3 OnlinePlayer

Ein OnlinePlayer ist ein Spieler, der dem Spiel von einem anderen Client aus beigetreten ist, der also nicht direkt über die Tastatur gesteuert wird, sondern vom Server Anweisungen erhält.

Auch hier gibt es eine *_physics_process*-Funktion, die von Godot automatisch sechzigmal pro Sekunde aufgerufen wird.

```
21 func _physics_process(_delta):
22     if new_stats.size() > 0  and new_stats[0].timestamp <= Server.get_time() - 50 and pos_reached:
23         stats = new_stats.pop_front().stats
24 
25     if stats.is_dead and not died:
26         if not stats.fallen:
27             $AnimatedSprite.animation = $AnimatedSprite.animation + "_dead"
28             $DieSound.play()
29             died = true
30 
31     if stats.fallen and not died:
32         $AnimatedSprite.get_node("FallAnimation").play("fall")
33         died = true
34 
35     if next_positions.size() > 0 and next_positions[0].time <= Server.get_time() - 50:
36         pos_reached = false
37         $AnimatedSprite.play()
38         var motion = (next_positions[0].position - position).normalized()
39 
40         if motion != Vector2.ZERO:
41             if motion.y == 0:
42                 rotation = motion.angle_to(Vector2.RIGHT)
43             else:
44                 rotation = -motion.angle_to(Vector2.RIGHT)
45             var amplifier = 1
46             if stats.hyperactive:
47                 amplifier = 4
48             elif stats.slow:
49                 amplifier = 0.5
50             var _v = move_and_slide(motion * stats.speed * amplifier)
51         if position.distance_squared_to(next_positions[0].position) < 16:
52             position = next_positions[0].position
53             next_positions.pop_front()
54             pos_reached = true
55     else:
56         $AnimatedSprite.stop()
57         $AnimatedSprite.frame = 0
```

Abb. 6 Ausschnitt aus LocalPlayer.gd: *_physics_process*-Funktion

Anders als beim *LocalPlayer* werden hier keine Eingaben verarbeitet.

Stattdessen werden zweit Listen verarbeitet. Die Liste new_stats enthält alle Zustände, die die Statuswerte am Server angenommen haben und am Client noch nicht angewendet wurden.
Die Liste next_positions stellt die Warteschlange der Positionen dar, die in 2.4 Entity_Interpolation erwähnt wurde. Für beide Listen wird jeweils überprüft, ob das nächste zu verarbeitende Element vor mindestens 50 ms vom Server angewendet wurde. Ist dies der Fall, so werden die Statuswerte angepasst bzw. zur nächsten Position interpoliert.

Die Interpolation findet statt, indem ein Vektor aus der derzeitigen und der zu erreichenden Position berechnet wird. Dieser Vektor wird dann als Richtungsvektor für die Bewegung des Spielers verwendet. Genau wie beim *LocalPlayer* wird dann ein *amplifier* festgelegt um bestimmte Statuswerte in die Bewegung mit einzubeziehen.

Wenn die Position mit einer Abweichung von weniger als 4 Pixeln, also einer quadratischen Entfernung von weniger als 16, erreicht wurde, wird die Position des Spielers auf die zu erreichende Position gesetzt und selbige aus der Warteschlange entfernt, damit der Spieler sich zur nächsten Position Bewegen kann. Da sich der Spieler nicht in Ein-Pixel-Schritten bewegt, ist es notwendig, diese Abweichung mit einzubeziehen.

Außerdem wird jeweils überprüft, ob der Spieler gestorben oder heruntergefallen ist. Ist dies der Fall, so werden die entsprechenden Animationen und Sound-Effekte abgespielt.

Die Variable *died* ist notwendig, um zu verhindern, dass diese Animationen mehrmals abgespielt werden.

4 Multiplayer-Implementierung

4.1 Godot high-level Multiplayer API

Für die Implementierung der Kommunikation zwischen Client und Server wurde die von Godot bereitgestellte hight-level Multiplayer API verwendet.

Diese stellt sogenannte “RPCs” (remote procedure calls) zur Verfügung. Auf diese Weise können Funktionen über das Netzwerk aufgerufen werden.

Dabei gibt es zwei verschiedene Arten solcher Aufrufe, nämlich zuverlässige (*rpc*) und unzuverlässige (*rpc_unreliable*). Der wesentliche Unterschied dieser beiden Methoden ist, dass ein zuverlässiger Aufruf gewisse Überprüfungen beinhaltet, die

sicherstellen, dass der Aufruf tatsächlich ankommt. Sollte ein zuverlässig gesendeter Aufruf “verloren gehen”, so wird er erneut angefordert. Diese Überprüfung ist zwar hilfreich, bei Spielen aber meist ungewollt, da sie die Übertragung stark verlangsamt.

4.2 Client

Der Server und der Client wurden jeweils in einem eigenen Godot-Projekt implementiert, im Folgenden wird zunächst der Aufbau des Clients und dessen Kommunikation mit dem Server näher beleuchtet.

Am Client wird für die Kommunikation mit dem Server ein Singleton erstellt. Dieser Singleton enthält dann verschiedene Funktionen, die RPC-Aufrufe an den Server senden.

Wichtig ist hierbei, dass nur dieser Singleton solche Aufrufe senden darf. Godot gibt diese Aufrufe nämlich an die Nodeweiter, die sich im Scene-Tree an derselben stelle befindet wie dieser Singleton. Aus demselben Grund muss der Singleton auch genauso heißen wie der Wurzelknoten im Server-Projekt, in diesem Fall heißt er “Server”.

Außerdem enthält selbiger Singleton auch Funktionen, die mit dem Schlüsselwort “remote” versehen sind. Diese Funktionen stellt der Client zur Verfügung und können vom Server aufgerufen werden.

Jede Funktion des Server-Singleton bis ins Detail zu behandeln würde den Rahmen dieser Dokumentation sprengen, deshalb werden die meisten Funktionen nur oberflächlich behandelt. Nur die wichtigsten stellen werden tiefgreifender behandelt.

Die Funktion *ConnectToServer* bat die Verbindung mit dem Server auf.

Außerdem verknüpft sie die beiden Signale “connection_failed” und “connection_succeeded mit einer jeweils dafür vorgesehenen Funktion.

Diese beiden Funktionen behandeln jeweils den Fall einer gelungenen bzw. einer fehlgeschlagenen Verbindung.

```
28 func ConnectToServer(): #address, port
29     network.create_client(address, port)
30     get_tree().set_network_peer(network)
31 
32     network.connect("connection_failed", self,"_OnConnectionFailed")
33     network.connect("connection_succeeded", self, "_OnConnectionSucceeded")
34 
35 func _OnConnectionFailed():
36     connection_failed = true
37     load_scene("res://scenes/ChooseServer.tscn")
38     print("Failed to connect")
39 
40 func _OnConnectionSucceeded():
41     load_scene("res://scenes/MainMenu.tscn")
42     print("Succesfully connected")
```

Abb. 7 Ausschntt Server.gd (client)

Die Funktionen *create_session, join_session, join_world* und *leave_session*, teilen dem Server jeweils mit, dass der Spieler dieses Clients eine Session (also ein neues Spiel) erstellen möchte, einer bestehenden Session betreten möchte, der Welt dieser Session beitreten möchte oder die Session verlassen möchte. Diese Funktionen rufen alle lediglich eine Funktion am Server auf und enthalten keine wesentliche Logik. Dasselbe gilt für die übrigen Funktionen, deren Aufgabe anhand ihres jeweiligen Namens ersichtlich sein sollte.

Die Funktionen *move_player* und *place_bomb* sind für das Bewegen des Spielers und das Platzieren von Bomben zuständig. Während die *place_bomb*-Funktion lediglich den Aufruf an den Server weitergibt, muss die Funktion move_player den Zeitpunkt der Bewegung mitgeben, da dieser für die Reconciliation wichtig ist.

```
59 func move_player(motion):
60     if motion != Vector2.ZERO:
61         var timestamp = OS.get_system_time_msecs()
62         var world_state = world.get_world_state()
63         past_states[timestamp] = world_state
64         rpc_unreliable_id(1, "move_player", motion, timestamp)
65     else:
66         rpc_unreliable_id(1, "stop_player")
67
68 func place_bomb():
69     rpc_id(1, "place_bomb")
70
```

Abb. 8 Ausschntt Server.gd (client)

Außerdem wird hier über die Funktion *get_world_state* der Welt der Zustand der Welt abgespeichert, damit er später mit den Daten verglichen werden kann, die vom Server gesendet werden.

Die “remote”-Funktionen sind die eigentlich interessanten Funktionen, so kann der Server beispielsweise über die *start_game* Funktion das Spiel am Client starten. Die *spawn_player* und *despawn_player* Funktionen sind jeweils dafür zuständig, Spieler zu erschaffen bzw. zu entfernen. Diese Funktionen delegieren jeweils an die Welt, wo die eigentliche Logik implementiert wurde.

Die wichtigste Funktion ist hier allerdings *update_world_state* diese Funktion wird vom Server aufgerufen, um die regelmäßigen Updates an die Clients zu senden.

Alle Daten, die der Server als “world_state” an den Client sendet, werden hier verarbeitet.

Beispielsweise wird die Liste aller Spieler durchlaufen, wobei für die lokalen Spieler die Funktion *reconcile_player* aufgerufen wird. (siehe 2.3 Server Reconciliation)

Bei allen übrigen Spielern wird die übermittelte Position zu deren Warteschlange hinzugefügt (siehe 2.4 Entity Interpolation und 3.3 OnlinePlayer)

Jeder Spieler hat hierbei einen eindeutigen Namen, der der *network_peer_id* des Clients entspricht. Dies ermöglicht es leicht, den lokalen Spieler zu identifizieren und auch die Daten der anderen Spieler mit den Informationen des Servers abzugleichen.

Auch die übermittelte Liste der auf der Map vorhanden Popups wird durchlaufen und die Spielwelt wird dementsprechend angepasst. Dasselbe geschieht mit den Bomben, bei denen zusätzlich noch die Zeit angepasst wird, die bleibt, bis sie explodieren.

Außerdem wird die Map aufgrund der vom Server übermittelten Daten aktualisiert.

Über die Funktion *ping*, die zu Beginn jedes Updates vom Client am Server aufgerufen wird, wird außerdem der Zeitunterschied zwischen Server und Client berechnet. Dies ist notwendig, da schon kleine Abweichungen der Zeit zwischen Server und Client bei der Entity Interpolation zu Problemen führen. Diese Abweichung wird dann in der Variable time_diff gespeichert und in der Funktion *get_time* verwendet um die Zeit des Clients in “Server-Zeit” umzuwandeln.

4.3 Server

Der Scene-Tree des Servers enthält bei Programmstart lediglich die Server-Scene. Diese befindet sich genau wie der Singleton des Clients direkt unter der Wurzel des Baums. Somit hat sie denselben Pfad im Baum wie dieses. Wie bereits erwähnt ist dieser Aufbau durch die Godot-Engine festgelegt.

Wenn am Server eine neue Session erzeugt wird, wird zum Scene-Tree ein neuer *ViewPort* hinzugefügt. Jeder *ViewPort* wird von Godot völlig separat behandelt, was es möglich macht, mehrere Welten parallel zu betreiben.

Diese Sessions werden in der Liste *sessions* gespeichert, die jeweils die der Session beigetretenen Spieler enthält. Außerdem ist in der *player_session_map* jedem Spieler, der einer Session beigetreten ist, seine Session zugeordnet.

Alle Aufrufe, die an die Clients gesendet werden, werden immer an alle Clients gesendet, die der entsprechenden Session zugeordnet sind.

Am Server gibt es eine StartServer-Funktion, die dafür zuständig ist, den Server zu starten.

Auch hier werden wieder zwei Signale mit Funktionen verknüpft.

Diese sind handhaben das Verbinden und Trennen der Verbindung eines Clients.

```gdscript
func StartServer():
    network.create_server(port, max_players)
    get_tree().set_network_peer(network)
    print("Server started")

    network.connect("peer_connected", self, "_Peer_Connected")
    network.connect("peer_disconnected", self, "_Peer_Disconnected")

func _Peer_Connected(player_id):
    print("User " + str(player_id) + " Connected")

func _Peer_Disconnected(player_id):
    if player_session_map.has(player_id):
        leave_session(player_id)
        print("User " + str(player_id) + " left sesson")
    print("User " + str(player_id) + " Disconnected")
```

Abb. 9 Ausschntt Server.gd (server)

Die *send_world_state*-Funktion ist das Gegenstück zu der am Client erwähnten Funktion *update_world_state*. Sie wird dreißigmal pro Sekunde von der *_physics_process*-Funktion jeder Welt aufgerufen. Der von der Welt generierte Zustand wird hier an den Client gesendet.

Die Funktionen *move_player* und *stop_player* sind jeweils dafür zuständig, den Spieler, von dessen Client die Anfrage gesendet wird, zu bewegen bzw. die Bewegung anzuhalten.

```gdscript
remote func move_player(motion, timestamp):
    var player_id = get_tree().get_rpc_sender_id()
    var world = get_node(player_session_map[player_id] + "/World")
    if world.get_node("Players").has_node(str(player_id)):
        var player = world.get_node("Players/" + str(player_id))
        player.move(motion)
        world.players[str(player_id)]["T"] = timestamp
        world.players[str(player_id)]["P"] = player.position

remote func stop_player():
    var player_id = get_tree().get_rpc_sender_id()
    var world = get_node(player_session_map[player_id] + "/World")
    var player = world.get_node("Players/" + str(player_id))

    player.move(Vector2.ZERO)
```

Abb. 10 Ausschntt Server.gd (server)

Die Funktion *player_ready* wird ausgeführt wenn ein Spieler zu Beginn einer Runde auf den Button “Ready” bzw. “Start” drückt. Sie überprüft, ob alle Spieler bereit sind und startet ggf. das Spiel.

Die Funktion *create_world* wird vom Client aufgerufen, wenn ein Spieler eine neue Session erstellt.

Hier wird zunächst ein neuer *ViewPort* erstellt.
Diesem wird dann eine neue Welt hinzugefügt, deren ausgewählte Map zuvor aufgrund des übergebenen Namens gesetzt wird. Wie die Session

```gdscript
remote func create_world(name, map):
    var viewport = Viewport.new()
    viewport.world = World2D.new()
    viewport.set_size(Vector2(1920, 1080))
    var world = world_scene.instance()
    world.map = map
    world.session_owner = get_tree().get_rpc_sender_id()
    #world.name = name
    var actual_name
    if name != "":
        actual_name = name
    else:
        actual_name = "World_"+str(sessions.size())
    viewport.name = actual_name
    viewport.add_child(world)
    sessions[actual_name] = []
    add_child(viewport, true)
```

heißen soll, wird ebenfalls übergeben, sollte kein Name übergeben werden, werden die Sessions durchnummeriert.

Abb. 11 Ausschntt Server.gd (server)

Die Funktion *place_bomb* stammt teilweise von einem anderen Gruppenmitglied. Es mussten allerdings einigen Anpassungen vorgenommen werden, um die Multiplayer-Funktionalität zu ermöglichen.

```
197 remote func place_bomb(player_id = get_tree().get_rpc_sender_id()):
198     var world = get_node(player_session_map[player_id] + "/World")
199     if world.get_node("Players").has_node(str(player_id)):
200         var player = world.get_node("Players/" + str(player_id))
201         if !player.stats.is_restrained:
202             if player.stats.layable_bombs > 0 and not player.in_bomb:
203                 var tilemap = world.get_node("TileMap")
204                 var bomb = bomb_scene.instance()
205                 bomb.name = str(player_id) + "-" + str(placed_bomb_count[player_id])
206                 placed_bomb_count[player_id] += 1
207                 bomb.position = tilemap.map_to_world(tilemap.world_to_map(player.position - tilemap.position)) + tilemap.position
208                 bomb.bomb_range = player.stats.bomb_blast_range
209                 bomb.player = player
210                 world.get_node("Bombs").add_child(bomb, true)
211             elif player.stats.can_throw and player.in_bomb:
212                 var tilemap = world.get_node("TileMap")
213                 var coords = tilemap.world_to_map(player.position - tilemap.position)
214                 if player.in_bomb and has_node(player_session_map[player_id] + "/World/Bombs/" + player.in_bomb.name):
215                     player.in_bomb.throw(coords+player.viewing_direction*2)
```

Abb. 12 Ausschntt Server.gd (server)

So muss beispielsweise jede Bombe einen eindeutigen Namen habe, um den Client mit dem Server abgleichen zu können. Außerdem muss die Bombe zur Welt aus der korrekten Session hinzugefügt werden.

Die übrigen Funktionen sollten relativ selbsterklärend sein, da sie im wesentliche keine Logik enthalten, sondern lediglich Funktionen des Clients aufrufen.