Dokumentation von Dustin Büchner

Teamname	SoulThieves
Rolle(n)	Scrum-Master und Entwickler

Aufgaben:

- Projektorganisation
- Diagramme entwerfen
- Grundlegende Bombe
- Bomben Animationen und Signale
- Sounddesign
- Powerup Spawn

Gliederung:

1 Einführung in das Projekt
    1.1 Allgemein
    1.2 Projektüberblick
    1.3 Projektorganisation
    1.4 Rolle des Scrum-Masters im Projekt
    1.5 Entwurf von Diagrammen
2 Implementierung der grundlegenden Bombe
    2.1 Szenengraph und dessen Aufbau
    2.2 Implemtierung der Logik
        2.2.1 Bomb
        2.2.2 Powerup Spawn
        2.2.3 BombExplosion
3 Animationen und Signale der Bombe
    3.1 Signale von Timern und Animationen
    3.2 Signale mithilfe von Kollisionen
4 Sounddesign

4.1 Kurzer Überblick
4.2 Abspielen einzelner Sounds

1 Einführung in das Projekt

1.1 Allgemein

Das Projekt wurde im Rahmen des Praktikums Software Entwicklung entwickelt. Ziel dieses Projektes war es ein größeres Softwareprojekt erfolgreich im Team zu bearbeiten und somit einen Einblick in die professionelle Softwareentwicklung zu erhalten.

1.2 Projektüberblick

Zusammen im Team viel die Entschiedung für ein Projektthema relativ leicht. Drei unserer Mitglieder spielten in der Berufsschule ein Spiel mit dem Namen “Granatier”, welches nur einen lokalen Multiplayer an einer Tastatur unterstützt. Aus diesem Grund haben wir uns dazu entschieden dieses Spiel mit einem globalen Multiplayer neu zu entwickeln, damit jeder an seinem Endgerät mitspielen kann. Hierfür ist eine Client-und eine Serverseitige Implementierung notwendig. Auf die genaue Architektur geht Andreas in seiner Dokumentation genauer ein.

Für dieses Projekt lassen sich folgende Aufgabengebiete definieren:

- Client-Server Kommunikation
- UI-Design
- Implementierung der Spiellogik

Es war uns vorallem wichtig ein übersichtliches und nutzerfreundliches Menü zu gestalten, um den Einstieg in das Spiel möglichst einfach zu gestalten. Die Spiellogik sollte sich natürlich von dem Originalspiel nicht unterscheiden, da wir auch nur die Ressourcen für Bilder und Ton zur Verfügung haben, die bereits mitgeliefert werden.

1.3 Projektorganisation

Das Projekt wurde mittles Scrum, einer agilen Methode des Projektmanagements , umgesetzt. Der Vorteil hierbei liegt in der iterativen Vorgehensweise, wodurch ein Projektüberblick einfacher möglich ist und man flexibel auf Änderungen reagieren

kann. Bei dieser Methode des Projektmanagements gibt es verschieden Rollen, die folgendermaßen verteilt wurden:

- Scrum Master:
  Dustin Büchner
- Product Owner:
  Dipl.-Inf. Stefan Müller
- Entwickler:
  Andreas Riegg, Max Schwarz, Alexej Kunz, Frank Müller, Dustin Büchner

Um regelmäßige Kontrollen zu ermöglichen gibt es sogenannte Scrum-Meetings, die zum Ende eines Sprints abgehalten wurden. Dort wurde der letzte Sprint nochmals aufgearbeitet und eine Aufgabenverteilung für den neuen Sprint und weiteres Vorgehen geplant. Die Sprint Dauer wurde auf 14 Tage festgelegt. Die Aufgabe des Protokollführers wurde für jedes Meeting neu vergeben.

Außerhalb der Scrum-Meetings gab es auch noch kleinere Treffen um mögliche Zustzaufgaben zu verteilen und für jede Aufgabe auch Stellvertreter zu definieren. Dies dient dem Zweck, dass falls ein Entickler ausfallen sollte die Aufgabe von seinem Stellvertreter übernommen werden kann. Des Weiteren hat er die Aufgabe mögliche Probleme zusammen mit dem Teammitglied zu lösen und Kontrollen gemäß der Funktionalität durchzuführen.

Ein großteil der Aufgaben sind in Eigneverantwotung zu bearbeiten, um möglichst effektive Arbeitsteilung zu gewährleisten.

1.4 Rolle des Scrum-Masters im Projekt

Der Scrum-Master trägt die Verantwortung dafür, dass das Team effektiv und ungestört arbeiten kann. Er legt dem Product Owner die Ziele und den Umfang von Aufgabenpaketen verständlich dar und schafft eine Kommunikationsebene mit dem Entwicklerteam. Die Aufgabenverteilung wurde zusammen mit allen Mitglieder im Team besprochen und auf den einzelnen angepasst. Außerdem achtet der Scrum-Master darauf das die Arbeit im Team ohne Probleme funktioniert und versucht mögliche Hindernisse zu beseitigen die den Eeinzelnen oder womöglich auch das ganze Team aufhalten könnten.

Des Weiteren moderiert er sämtliche Meetings, auch solche die außerhalb von den Geplanten stattfinden. Die Zusammenarbeit im Team hat reibungslos funktioniert und jeder ist seinen Tätigkeiten und Aufgaben meist vollständig nachgekommen.

Probleme von einzelnen Mitgliedern wurden nach Absprache mit dem Scrum-Master oder anderen Entiwcklern aus dem Weg geschafft.

1.5 Entwurf von Diagrammen

Um den Start in die Entwicklungsphase möglichst einfach zu gestalten wurde nach einem groß angelegten Meeting in der Planungsphase der Projektaufbau und Schnittstellen definiert. Des Weiteren wurden erste Aufgabepakete anglegt, die die anschließende Verteilung vereinfachen sollten.

Mit diesen Informationen wurden dann drei wesentliche Diagramme erstellt:

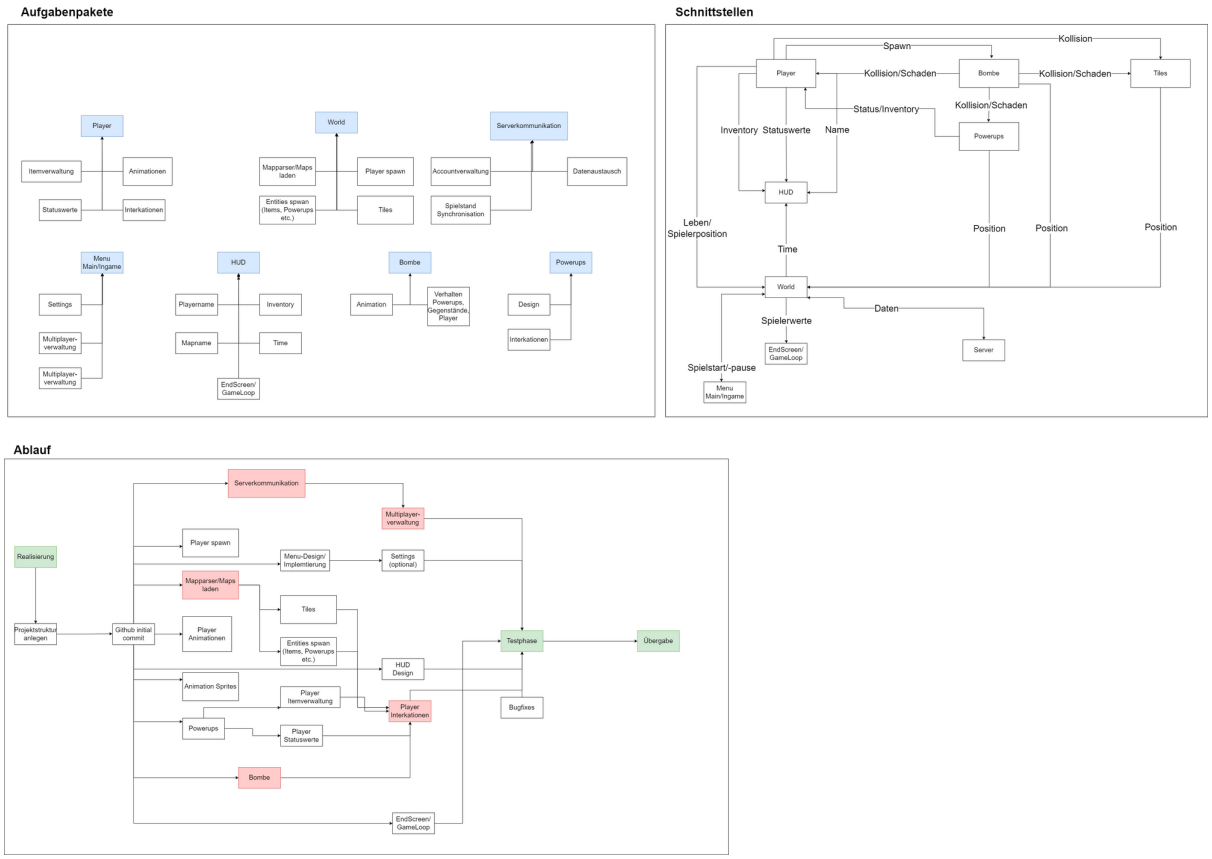

Abbildung 1

Auf die einzelnen Diagramme und deren Beschreibung werde ich hier nicht weiter eingehen. Diese werden in der Projektdokumentation von Max Schwarz genauer beleuchtet.

Mit diese Informationsgrafiken wurde der Umfang des Projektes relativ schnell ersichtlich. Der Umfang der Grafiken kann aber von der tatsächlichen Implementierung abweichen.

2 Implementierung der grundlegenden Bombe

2.1 Szenengraph und dessen Aufbau

Das unter anderem wichtigste Spielelement ist die Bombe. Um die Funktionalitäten der Engine möglichst gut nutzen zu können, wurde die bereits erstellte Szene der Bombe von Alexej leicht überarbeitet. Die “AnimatedSprite”-Node für die vier Explosionsrichtungen wurde in eine neue Szene verlagert um später Berührungen mit der Bombe über die Kollisionen abfragen zu können und die Szenenstruktur übersichtlicher zu halten.

Abbildung 2.1

Die Haupt-Node “Bomb” (Abbildung 2.1) ist ein “KinematicBody2D”, da in der Engine damit sich bewegende Körper leichter modelliert werden können. Die Bombe soll sich im späteren Verlauf durch Pfeile oder gewissen Powerups in Bewegung setzen. Die Node “BombExplosion” hat keinerlei Funktionalität. Im Code werden die

Explosionsstrahlen an diese Baumposition angehängt um die Übersichtlichkeit des Szenegraphen zu gewährleisten.

Des weiteren sind zwei Timer (Abbildung 2.1) hinterlegt, einer kümmert sich um die Dauer bis die Bombe explodiert und der andere bezeiht sich auf die reine Zeit der Explosion bis die Bombe verschwindet. Die beiden Nodes “BombAnim” und “AnimatedExplosion”(Abbildung 2.1) beziehen sich auf die Darstellung der Bombe, auf der einen Seite das pulsieren bis zur Explosion und auf der anderen der explodierende Kern der Bombe. Der “CollisionShape2D”(Abbildung 2.1) ist für die allgemeine Kollisionserkennung mit anderen Objekten in der Engine zuständig.

Wichtig ist noch die Node “PlayerIntersection”(Abbildung 2.1), die sich um Berührungen mit dem Spieler nach dem legen der Bombe kümmert. Mit der Funktionalität wird sich allerdings erst in Kapitel 3 beschäftigt. Abschließend gibt es noch zwei Nodes für das Sounddesign.

Abbildung 2.2

In dieser Szene wird auschließlich der Explosionstrahl der Bombe dargestellt. Hier besteht die Haupt-Node aus einer “Area2D”, damit Berührungen leichter abgefragt werden können. Ddie Node “CollisionShape2D” legt die Form der “Area” fest in welcher nach Kollisionen geprüft werden soll. Der “AnimatedSprite” kümmert sich um die Darstellung des Strahls und wurde von Alexej genauer beschrieben. Allerdings ist anzumerken, dass es nicht mehr für jede Richtung eine Darstellung gibt, sonder insgesamt nur noch eine und das korrekte Darstellen durch die Rotation geregelt wurde.

2.2 Implemtierung der Logik

2.2.1 Bomb

Das wichtigste Spielelement ist wie bereits gennant, die Bombe. Es muss gewährleistet werden, dass sie in alle vier möglichen Richtungen exploddieren kann,

aber nur soweit bis Hindernisse zerstört werden müssen oder die Explosionsrecihweite erreicht wird.

Bei der Implementierung wurde die Bombe in zwei wesentliche Bestandteile aufgeteilt, die Explosion im allgemeinen und Interaktionen mit der Bombe wie kicken oder werfen. Letzteres wird von Max genauer beleuchtet.

```
var player

var bomb_explosion_scene

## Member variables
var bomb_range = 1# Range of the bomb explosion
var explode_directions = [Vector2.UP,Vector2.DOWN,Vector2.RIGHT,Vector2.LEFT]
var explode_rotations = [0,PI,PI/2,1.5*PI]
var cell_size
var exploding = false # Is the bomb exploding?
```

Für die Implementierung der Bombe gibt es ein paar wesentliche Variablen (Abbildung 2.3). Die Variablen “player” und “bomb_explosion_scene” speichern Instanzen, die sich aus dem Name bereits schließen lassen. Einmal wird der Spieler, der die Bombe gelegt hat, gespeichert und einmal wird die Szene für den Explosionsstrahl gespeichert. Die anderen Variablen sind relativ selbst erklärend. Die Bombenreichweite wird gespeichert, sowie die Richtungen in der sich die Explosion ausbreitet und wie sich die Szene des Explosionstrahls entsprechend zur Richtung rotieren muss. Die Größe einer Zelle wird in der Spielwelt wird in der Variable “cell_size” entsprechend gespeichert.

Initial besitzt die Bombe keinerlei Kollision mit einem Spieler, da dieser sich beim Setzen der Bombe noch in ihr befindet.

```
func _ready():
  if player == null:
    set_collision_mask_bit(0,true)
  bomb_explosion_scene = load("res://scenes/BombExplosion.tscn")
  ...
  cell_size = get_node("../../TileMap").get_cell_size()
  ...
```

Wenn kein Spieler die Bombe gesetzt hat, sondern der Server nach Zeitablauf den SuddenDeath aktiviert, dann muss bereits die Kollision gesetzt werden, wenn die Szene im Szenengraph initialisiert wird. Danach wird der Explosionstrahl geladen und die Zellengröße wird gesetzt. (Abbildung 2.4)

Die wichtigste Funktionalität fehlt allerdings noch. Die Explosion der Bombe, wenn der entsprechende Timer dafür abgelaufen ist.

```
func explode():
  exploding=true
  ...
  ...
  var tilemap = get_parent().get_parent().get_node("TileMap")
  var coords = get_map_coords()
  ...
  if get_celltype_from_coords() == "arena_wall":
    tilemap.destroy_cell(coords.x,coords.y)
  for y in range(4):
    for i in range(bomb_range):
      var current_coords = coords+(explode_directions[y]*(i+1))
      match get_celltype_from_coords(current_coords):
        "arena_greenwall":
          break
        "arena_wall":
          tilemap.destroy_cell(current_coords.x,current_coords.y)
          var bomb_explosion = bomb_explosion_scene.instance()
          $AnimatedExplosion.add_child(bomb_explosion)
          bomb_explosion.global_position = get_center_coords_from_cell_in_world_coords
()+(explode_directions[y]*i*40)
          bomb_explosion.rotation = explode_rotations[y]
          break
        _:
          var bomb_explosion = bomb_explosion_scene.instance()
          $AnimatedExplosion.add_child(bomb_explosion)
          bomb_explosion.global_position = get_center_coords_from_cell_in_world_coords
()+(explode_directions[y]*i*40)
          bomb_explosion.rotation = explode_rotations[y]
  get_node("TimerAnim").start()
  ...
```

Zunächst werden ein paar Variablen mit Werten versorgt die für spätere Berechnungen notwendig sind. Die erste “if”-Verzweigung prüft, ob die Bombe selbst auf einer Wand ist und ruft die entsprechende Methode auf, damit die Wand zerstört wird. Hier fällt bereits eine Besonderheit auf. Die Bombe kümmert sich nicht selbst um das entfernen von Blöcken, sondern die TileMap.

```
func destroy_cell(x,y):
  set_cell(x,y,tile_set.find_tile_by_name("arena_ground"))
```

In Abbildung 2.6 wird lediglich die Zelle auf den neuen Typ gesetzt.

In der verschachtelten “for”-Schleife (Abbildung 2.5) werden alle vier Richtungen enstprechend der Reichweite abgelaufen. Um die Koordinaten für die jeweiligen Richtungen möglichst einfach zu berechnen, wird die Variable “explode_directions” verwendet, welche mit Einheitsvektoren für alle vier Richtungen vordefiniert ist. Nun muss natürlich geprüft werden, welche Zellentypen sich an den jeweiligen Koordinaten befinden. Enstprechend des Typs “arena_greenwall” (eine nicht zerstörbare Wand) muss die Überprüfung in diese Richtung abgebrochen werden. Bei dem Typ “arena_wall” wird die enstprechende Zelle zerstört und der Bombenstrahl mit der enstprechenden Rotation dem Szenengraph hinzugefügt und gezeichnet. Danach wird die aber auch hier die Überprüfung in diese Richtung abgebrochen, da immer nur eine Wand zerstört werden kann. Das gleiche Vorgehen trifft auch auf den letzten Typ “_” zu. Dieser ist ein deafult-Wert für Fälle die vorher nicht eingetreten sind. Dort wird der Bombenstrahl mit der enstprechenden Rotation dem Szenengraph hinzugefügt und gezeichnet, aber die Überprüfung nicht abgebrochen, sondern fortgesetzt.

Nachdem nun alle Richtungen abgearbeitet wurden kann abschließend der Timer für die Dauer der Explosion gestartet werden. Die vorausgegangen beiden Absätze beziehen sich auf Abbildung 2.5.

2.2.2 Powerup Spawn

Die Powerups werden durch das zerstören von Blöcken gespawnt. Allerdings immer nur mit einer gewissen Warscheinlichkeit und einem zufälligem Typ. Die Logik dafür befindet sich auch in der Funktion aus Abbildung 2.6. Das Spawnen von Powerups passiert nur Serverseitig, der wiederum dann diese Infomationen an den Client schickt (Dokumentation Andreas).

```
func destroy_cell(x,y):
  rng.randomize()
  ...
  if rng.randf_range(0.0,1.0)>0.33:
    var center_point_from_cell = self.map_to_world(Vector2(x,y))+Vector2(20,20)+self.p
osition
    var powerup = powerup_scene.instance()
    #powerup.set_type()
    var powerups = get_node("../Powerups")
    powerup.name = "pu_" + str(powerups.get_child_count())
    powerups.add_child(powerup, true)
    powerup.global_position = center_point_from_cell
```

Der Zufallsgenerator wird anfangs initialisiert und dann werden mit einer Warscheinlichkeit von 33% zufällige Powerups erzeugt (Abbildung 2.7). Zunächst

muss die Position des Powerups berechnet werden. Danach wird es dem Szenegraph hinzugefügt und entsprechend an die Clients für die Anzeige verteilt.

2.2.3 BombExplosion

Der Explosionstrahl der Bombe behinhaltet auch einen kleinen teil der Logik, da wie bereits in Kapitel 2.1 erwähnt, alle Interaktionen, die nicht Blöcke betreffen, über die Kollision abgefragt werden sollen.

Wenn eine Kollision festgestellt wird, dann wird die Funktion `body.destroy()` aufgerufen. Im nachfolgendem Kapitel werde ich auf Signale bzw. Kollisionserkennung nochmal genauer eingehen. “Bodys” können entweder Spieler oder Powerups sein. Enstprechend des Typs wird dann die Funktion “destroy()” aufgerufen.

```
func destroy():
  if !stats.has_shield:
    stats.is_dead = true
    ...
  else:
    stats.has_shield = false
```

Hier sehen wir die Implemtierung der Funktion beim Spieler. Es wird geprüft ob er sterben kann, wenn nicht dann muss sein Schild entfernt werden, ansonsten wird sein Statuswert ensprechend gesetzt.

```
get_parent().get_parent().players_alive.erase(int(name))
```

Serverseitig wird der Spieler noch aus den Liste der lebenden Spieler entfernt, damit jeder Client die Information erhält, welcher Spieler noch am Leben ist.

```
func destroy():
  queue_free()
```

Die Implemtierung der Funktion für Powerups ist relativ kurz, da diese lediglich zerstört werden müssen wenn sie von der Explosion getroffen werden.

3 Animationen und Signale der Bombe

3.1 Signale von Timern und Animationen

Bereits am Aufbau der Szene der Bombe wird klar, dass mit Timern gearbeitet wurde um den zeitlichen Ablauf der Bombe bestmöglich steuern zu können.

```
func _ready():
  ...
  self.get_node("BombAnim/AnimationPlayer").play("Bomb")
  ...
```

Beim Legen der Bombe wird der Timer bis zur Explosion automatisch gestartet. Diese Vaerhalten kann bei der Konfigutation des Timers hinterlegt werden. Wie in Abbildung 3.1 zu sehen, wird die Animation für das pulsieren der Bombe gestartet, wenn die Bombe gezeichnet wird bzw. in den Szenengraph der Welt geladen wird.

```
func _on_ExplotionTimer_timeout():
  explode()
```

Sobald der vorher genannte Timer abläuft wird die Funktion “explode()” aufgerufen und das Explosionsverhalten beginnt.

```
func explode():
  ...
  get_node("AnimatedExplosion").show()
  ...
  $BombAnim.hide()
  ...
  get_node("TimerAnim").start()
  get_node("AnimatedExplosion").play()
```

Dort angekommen wird die enstprechende Animation für die Explosion des Kerns sichtbar geschaltet und die pulsierende Animation versteckt. Am Ende der Funktion wird der Timer für die Dauer der Explosin und auch die Animation selbst noch gestartet.

```
func _on_TimerAnim_timeout():
  get_parent().remove_child(self)
```

Sobald auch der Timer für die Explosionsdauer ausläuft wird die Bombe aus der Welt entfernt und auch alle zugehörigen Explosionsstrahlen.

3.2 Signale mithilfe von Kollisionen

Wie bereits in einem vorherigen Abschnitt erwähnt, besitzt eine Bombe initial keine Kollision mit Spielern. Um diese zu setzen, wenn ein Spieler nach dem Legen der Bombe sich aus ihrem Darstellungsradius heraus bewegt hat, gibt es ein enstprechendes Signal einer “Area”-Node, welches dann ausgelöst wird.

```
func _on_PlayerIntersection_body_exited(body):
  $CollisionShape2D.set_deferred("disabled", false)
  set_collision_mask_bit(0,true)
  ...
```

Sobald dieses Signal ausgelöst wird, wird die Kollision der Bombe aktiviert und Spieler können nicht mehr in die Bombe hinein laufen. Auf diese Weise ist gewährleistet, dass der Spieler nicht mit der Bombe kollidiert, wenn er sich noch in ihr befindet. Würde diese Überprüfung nicht stattfinden, dann kann es zu unvorhersehbaren Verhalten kommen, dass der Spieler sich entweder nicht mehr bewegt, weil durch Kollisionen die Bewegung blockiert wird oder er wird aus der Bombe heraus gedrückt.

Die beiden wichtigsten Signale befinden sich in der Szene “BombExplosion”.

```
func _on_BombExplosion_body_entered(body):
  if body.is_in_group("Players"):
    body.destroy()


func _on_BombExplosion_area_entered(area):
  if area.is_in_group("Powerups"):
    area.destroy()
```

Diese Signale werden ausgelöst, wenn ein Objekt mit dem ein Bombenstrahl kollidiert und sich somit in dem Darstellunsbereich befinden (Abbildung 3.6). Wichtig hierbei ist, dass die Kollision nicht interkativ ist, also der Spieler nicht blockiert wird, sondern nur abgefragt wird, ob ein Objekt (in diesem Fall der Spieler) sich in einem gewissen Bereich befindet.

Spieler sind “body´s”, deswegen wird überprüft ob eine Node diesen Typs mit dem Explosionstrahl kollidiert und ob es dann enstprechend ein Player ist. Ist dies der Fall dann wird auf diesem die Funktion “destroy()” aufgerufen.

Selbiges gilt für die zweite Funktion. Allerdings wird hier auf den Typ “area” geprüft, weil Powerups als solche repräsentiert sind. Ist es dann tatsächlich ein Powerup,

dann wird die gleiche Funktion “destroy()” aufgerufen allerdings mit der entsprechenden Funnktionalität für Powerups.

4 Sounddesign

4.1 Kurzer Überblick

Das Sounddesign ist aus dem Originalspiel übernommen. Aus diesem Grund gibt es auch nur für begrenzte Fälle einzelne Sounds. Es liegt allerdings keine Musik für Menü und Gameplay vor.

4.2 Abspielen einzelner Sounds

Sounds werden durch “AudiStreamPlayer” in GoDot abgespielt, welche im Szenengraph hinterlegt werden müssen.

```
func _ready():
  ...
  $PutBombSound.play()
```

In Abbildung 4.1 wird, sobald die Bombe gelegt und in der Welt gezeichnet ist, der Sound für das Legen einer Bombe abgespielt.

```
func explode():
  ...
  $AudioStreamPlayer.play()
  ...
```

Kommt es zur Explosion der Bombe und entsprechend zum Funktionsaufruf, dann wird der Sound für die Explosion abgespielt.

An diesen Beispielen wird realtiv schnell deutlich, dass sich das Sounddesign relativ leicht und flexibel gestalten lässt. Weitere Sounds, etwa wie das aus der Map fallen oder das aufsammeln von Pwerups, funktionieren analog zu den beiden Beispielen. Das Starten muss nur an den jeweiligen Stellen im Code geregelt werden.