

Projektbericht
Studiengang : Informatik

Spieleprogrammierung

von

Lisa-Marie Hege

74974

Betreuender Professor: Prof. Dr. Carsten Lecon

Einreichungsdatum : 30. Juli 2020

Eidesstattliche Erklärung

Hiermit erkläre ich, **Lisa-Marie Hege**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Heitersheim, 30.07.2020

Ort, Datum

A handwritten signature in black ink, appearing to read 'L. Hege', written in a cursive style.

Unterschrift (Student)

Kurzfassung

Die vorliegende Projektarbeit befasst sich mit der Konzeption und Implementierung eines 2D Platform Spiels in C#. Hierzu wird die Game Engine Unity verwendet. Es wurde ein eine Spielidee ausgearbeitet und anhand aktueller Game Design Vorgaben ein Lösungskonzept erstellt. Die Design- und Implementierungsphase beschreibt den Entwicklungsprozess von der Erstellung von Grafiken und Animationen bis zur Programmierung des Spiels.

Das Ergebnis ist ein kleines PC-Spiel, bestehend aus mehreren Menü's und vorerst zwei Leveln.

Inhaltsverzeichnis

Eidesstattliche Erklärung	i
Kurzfassung	ii
Inhaltsverzeichnis	iii
Abbildungsverzeichnis	vi
Quelltextverzeichnis	vii
1 Einleitung	1
1.1 Ziel	1
1.2 Vorgehen	1
2 Grundlagen	2
2.1 Projektmanagement	2
2.1.1 Trello	2
2.2 Unity	2
2.2.1 Scene	2
2.2.2 GameObject	2
2.2.3 Component	2
2.2.4 Assets	3
2.2.5 Raycast	3
2.2.6 Layer	3

2.2.7	Tag	4
2.2.8	Canvas	4
3	Game Design / Anforderungen	5
3.1	Spielidee	5
3.2	Konzept	5
3.2.1	Screens/Navigation	5
3.2.2	Level	5
3.2.3	Schwierigkeit	6
3.2.4	Sound	6
3.2.5	Plattform	6
3.2.6	Steuerung	6
3.3	Setting	6
3.3.1	Welt	6
3.3.2	Charaktere	7
3.3.3	Spielobjekte	8
4	Lösungskonzept	9
5	Implementierung	11
5.1	Grafiken/Sprites erstellen	11
5.2	Animationen	12
5.3	Spieler	13
5.3.1	Bewegung	13
5.3.2	Jump, Double Jump und Wall Jump	14
5.3.3	Health	17
5.3.4	Animation	18

5.4	Waffe	19
5.5	Gegner	21
5.5.1	Virus/Patrolling Enemy	21
5.5.2	Boss	22
5.6	Manager	25
5.6.1	GameMaster	25
5.6.2	AudioManager	25
5.7	UI	26
5.7.1	Score	26
5.7.2	HealthBar	26
5.8	Spielobjekte	27
5.9	Level	28
5.10	Kameraführung	28
5.11	GUI	28
5.12	Sound	28
6	Evaluierung	29
7	Ergebnisse und Ausblick	30
7.1	Erreichte Ergebnisse	30
7.2	Ausblick	33
7.2.1	Erweiterbarkeit der Ergebnisse	33
7.2.2	Übertragbarkeit der Ergebnisse	33

Abbildungsverzeichnis

1.1	Trello Board	1
3.1	Von links nach rechts: Wissenschaftler, Virus, Virenboss	7
3.2	Waffe mit Toilettenpapier-Projektil	8
3.3	Spielobjekte	8
4.1	Endgültiges Klassendiagramm	10
5.1	Farbpalette erzeugt mithilfe der Website https://coolors.co	11
5.2	Spieler Animator	12
5.3	Bossvirus Animator	12
7.1	Hauptmenü	30
7.2	Level 1: In-Game	31
7.3	Level 1: Übersicht über das gesamte Level	31
7.4	Level 2: Virenboss	32
7.5	Pause Menü	32

Listings

5.1	Auszug aus PlayerController Skript: Bewegung	13
5.2	Auszug aus PlayerController Skript: Ergaenzung von Jumps	15
5.3	Auszug aus PlayerHealth Skript	17
5.4	Auszug aus dem PlayerAnimation Skript	18
5.5	Auszug aus dem Weapon Skript	19
5.6	Auszug aus dem ToiletPaper Skript	20
5.7	Auszug aus dem Patrolling Enemy Skript	21
5.8	Auszug aus dem Boss Skript	22
5.9	Auszug aus dem GameMaster Skript	25
5.10	Auszug aus dem Score Skript	26
5.11	Auszug aus dem HealthBar Skript	26

1 Einleitung

1.1 Ziel

Ziel dieser Arbeit ist es, einen grafisch ansprechenden 2D Plattformer (Jump 'n' Run) von Grund auf zu konzipieren und zu implementieren. Hierbei war es mir wichtig, möglichst viele Aspekte der Spieleprogrammierung kennenzulernen und meine Fähigkeiten weiter auszubauen. Das Ergebnis sollte ein funktionierendes Spiel sein, das vollständig und intuitiv spielbar ist.

1.2 Vorgehen

Zu Beginn des Projekts musste eine vage Spielidee ausgearbeitet werden. Nachdem das grundlegende Thema des Spiels fest stand, stand die Definition der Anforderungen an das Spiel im Vordergrund. Dazu wurden Skizzen von Welt und Charakteren angefertigt und das gesamte Spielkonzept ausgearbeitet. Anschließend wurde mithilfe von ersten Klassendiagrammen ein Lösungskonzept erstellt und implementiert. Zum Management der zu erledigenden Aufgaben wurde, wie in Abbildung 1.1 zu sehen, ein **Trello** Board angelegt. Dieses Board hat mich durch die komplette Implementierungsphase begleitet. Die dort gelisteten Aufgaben wurden dann nach und nach abgearbeitet. Als Game Engine kam Unity zum Einsatz, da dies für mich, aufgrund meiner Vorerfahrung, die beste und einfachste Option zur 2D-Spieleprogrammierung ist.

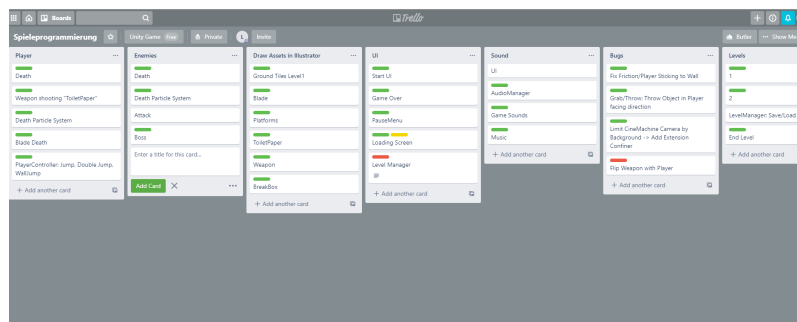


Abbildung 1.1: Trello Board

2 Grundlagen

2.1 Projektmanagement

2.1.1 Trello

Trello ist ein Online-Dienst zur Aufgabenverwaltung von Atlassian. Es werden Boards mit Aufgaben erstellt, die dann z.B. mit Prioritäten und Terminen versehen werden können.

2.2 Unity

Eine Cross-Platform Game Engine.

2.2.1 Scene

Ein Unity Projekt besteht aus einer Anzahl von verschiedenen Szenen, die einzeln erstellt werden. Eine Szene besteht aus einer Menge von **GameObjects**, die hierarchisch im Projektfenster angeordnet werden.

2.2.2 GameObject

Jedes Element in einer Unity Szene ist ein GameObject. Es ist die Basisklasse aller Entitäten in Unity.

2.2.3 Component

Sie stellen die Funktion eines **GameObjects** in Unity bereit. Die wichtigste Komponente ist die Transform-Komponente, welche die Position, Rotation und Skalierung eines Objekts bestimmt. Weitere Komponenten können z.B. Collider, Rigidbody, Partikelsysteme oder Skripte sein.

Rigidbody

Mit einer Rigidbody 2D Komponente wird ein **GameObject** unter Kontrolle der Unity Physics Engine gestellt. So kann z.B. die Schwerkraft für ein Objekt festgelegt werden.

Collider

Ein Collider definiert die Form eines 2D-**GameObjects** zur Erkennung physischer Kollisionen.

2.2.4 Assets

Assets sind alle Medieninhalte des Spiels, wie z.B. Grafiken, Audiodateien oder Skripte. Sie können selbst erstellt oder aus dem Unity Asset Store geladen werden.

Sprite

Sprites sind 2D Grafikobjekte/Texturen.

Tiles

Tiles sind Assets, die auf einer Tilemap angeordnet werden können, um eine 2D Spielumgebung zu konstruieren. Jeder Tile referenziert ein bestimmtes **Sprite**, welches dann am festgelegten Standort in der Tilemap gerendert wird.

2.2.5 Raycast

Sendet einen Strahl vom Ursprung in eine festgelegte Richtung mit einer bestimmten Länge und erkennt andere **Collider** in der **Scene**.

2.2.6 Layer

Layers werden im Unity Editor verwendet, um Gruppen aus Spielobjekten zu bestimmen, die gemeinsame Merkmale besitzen. Layers können z.B für **Raycasting** verwendet werden, sodass nur relevante Objekte erkannt werden.

2.2.7 Tag

Tags können an Spielobjekte im Unity Editor vergeben werden, um diese Objekte im Code zu identifizieren.

2.2.8 Canvas

Der Canvas beinhaltet alle UI-Elemente eines Spiels.

3 Game Design / Anforderungen

In diesem Kapitel wird die anfängliche Spielidee dargestellt und Anforderungen an das Spiel definiert. Es können auch Ideen enthalten sein, die nicht umgesetzt, aber im Hinblick auf die weitere Entwicklung festgehalten wurden.

3.1 Spielidee

Ein Wissenschaftler muss sich mit seiner Toilettenpapier-Kanone durch eine gefährliche Welt voller Viren und anderen Hindernissen kämpfen. Er sammelt DNA/Antikörper, um einen Impfstoff zu entwickeln und das Virus letztendlich zu besiegen.

3.2 Konzept

3.2.1 Screens/Navigation

- Hauptmenü: Das Spiel kann gestartet oder beendet werden.
- Level Auswahl: Es soll einen Screen zur Levelauswahl geben. Der Fortschritt wird gespeichert und hier angezeigt.
- Loading Screen: Wird beim Laden einer **Scene**/eines Levels angezeigt.
- Pause Menu: Das Spiel kann jederzeit pausiert werden.

3.2.2 Level

Vorerst sollen aus Zeitgründen nur ein bis drei Level kreiert werden. Wenn der Spieler am Ende eines Levels angekommen ist und genügend DNA gesammelt hat, öffnet sich ein Portal zum nächsten Level. Falls nicht genügend DNA gesammelt wurde, startet das Level neu.

3.2.3 Schwierigkeit

Es gibt nur einen Schwierigkeitsgrad zur Auswahl, allerdings soll sie von Level zu Level steigen.

3.2.4 Sound

Die Hintergrundmusik soll elektronisch gehalten werden und eine bedrohliche Grundstimmung erzeugen. Die Aktionen des Spielers und sonstige Ereignisse sollen mit Soundeffekten untermalt werden (z.B beim Springen oder Einsammeln von DNA). Die Sounds sollen aus externen Quellen beschafft werden.

3.2.5 Plattform

Das Spiel soll vorerst für den PC entwickelt werden. Später kann eine mobile Steuerung implementiert und die Plattform auf Android umgestellt werden.

3.2.6 Steuerung

Der Spieler kann die Laufrichtung des Hauptcharakters mit dem üblichen Eingabetasten (A, D) oder den Pfeiltasten bestimmen. Bei Drücken der Leertaste, springt der Spieler. Je länger sie gedrückt wird, desto höher ist der Sprung. Bei zweimaligem Drücken der Leertaste wird ein Double Jump ausgeführt. Mit der Maus kann gezielt und die Waffe abgefeuert werden.

3.3 Setting

3.3.1 Welt

Die Welt soll mechanisch/wissenschaftlich wirken und nach einer Art Laborumgebung aussehen. Sie soll trotz des Themas sehr farbenfroh und comicartig gestaltet sein. Die Welt soll dynamisch wirken und daher aus vielen interaktiven, beweglichen Elementen und verschiedenen Hindernissen aufgebaut sein.

3.3.2 Charaktere

Spieler: Wissenschaftler

Er ist der steuerbare Charakter im Spiel und soll vorerst folgende Fähigkeiten besitzen: Run, Jump, Double Jump, Wall Jump, Shoot.

Virus

Das Virus ist der Standardgegner in den normalen Leveln. Es schwebt ziellos durch die Luft und tötet bei Berührung sofort. Der Spieler kann es durch beschießen mit Toilettenpapier zerstören. Bei Zerstörung lässt es DNA fallen, die vom Spieler eingesammelt werden kann.

Virenboss

Der Virenboss ist der Endgegner im letzten Level und soll den Spieler gezielt auf verschiedene Arten angreifen können. Es zu zerstören dauert länger als beim gewöhnlichen Virus.

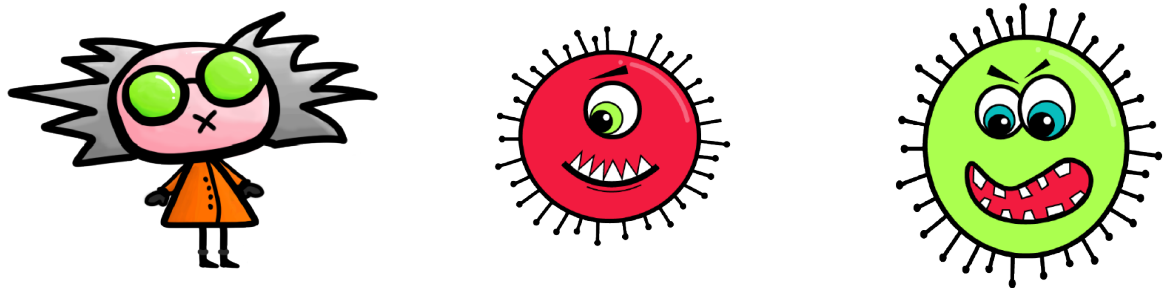


Abbildung 3.1: Von links nach rechts: Wissenschaftler, Virus, Virenboss

3.3.3 Spielobjekte

Im Spiel soll es verschiedene Objekte geben, mit denen der Spieler interagieren kann:

- Waffe: Der Spieler kann mit Toilettenpapier schießen, um Gegner/Viren zu töten. Trifft das Toilettenpapier einen Gegner, so wird ihm Gesundheit abgezogen.
- Plattformen: Statisch in der Luft schwebend oder beweglich.
- Verschiedene rotierende Sägeblätter: Fix an einer Stelle oder beweglich. Der Spieler stirbt bei Berührung sofort.
- DNA: Sammelobjekt, nach dem Einsammeln erhöht sich die Punktzahl.
- Box: Kann durch Springen vom Spieler zerstört werden und lässt DNA fallen.
- Portal: Öffnet sich bei erfolgreichem Ende des Levels und beim Spawn ins Spiel/Level

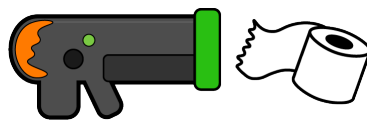


Abbildung 3.2: Waffe mit Toilettenpapier-Projektil



Abbildung 3.3: Spielobjekte

4 Lösungskonzept

Um die in Kapitel 3 vorgestellten Anforderungen und Konzepte sinnvoll und effektiv umzusetzen, habe ich mir den grundlegenden Aufbau und die Zusammenhänge der benötigten Klassen überlegt. In dieser Phase sind erste Klassendiagramme für die Charaktere und die wichtigsten Objekte entstanden. Da mit der Zeit noch kleinere Klassen und Komponenten (wie z.B. Sound) hinzu kamen, haben sich diese mit der Zeit, wie in Abbildung 4.1 zu sehen, zu einem einzigen großen Klassendiagramm entwickelt.

Jedes Spiel benötigt zudem Grafiken für die Spielobjekte. Bevor die Klassen implementiert werden konnten, mussten zunächst die jeweiligen Sprites für sämtliche Spielobjekte und die UI erstellt, aus dem Unity Asset Store oder anderen externen Quellen heruntergeladen werden. Da möglichst wenige fremde Assets verwendet werden sollten, wurden die meisten Assets in Adobe Photoshop und Illustrator angefertigt. Anschließend mussten in Unity passende Animationen für den Spieler und die Gegner erstellt werden.

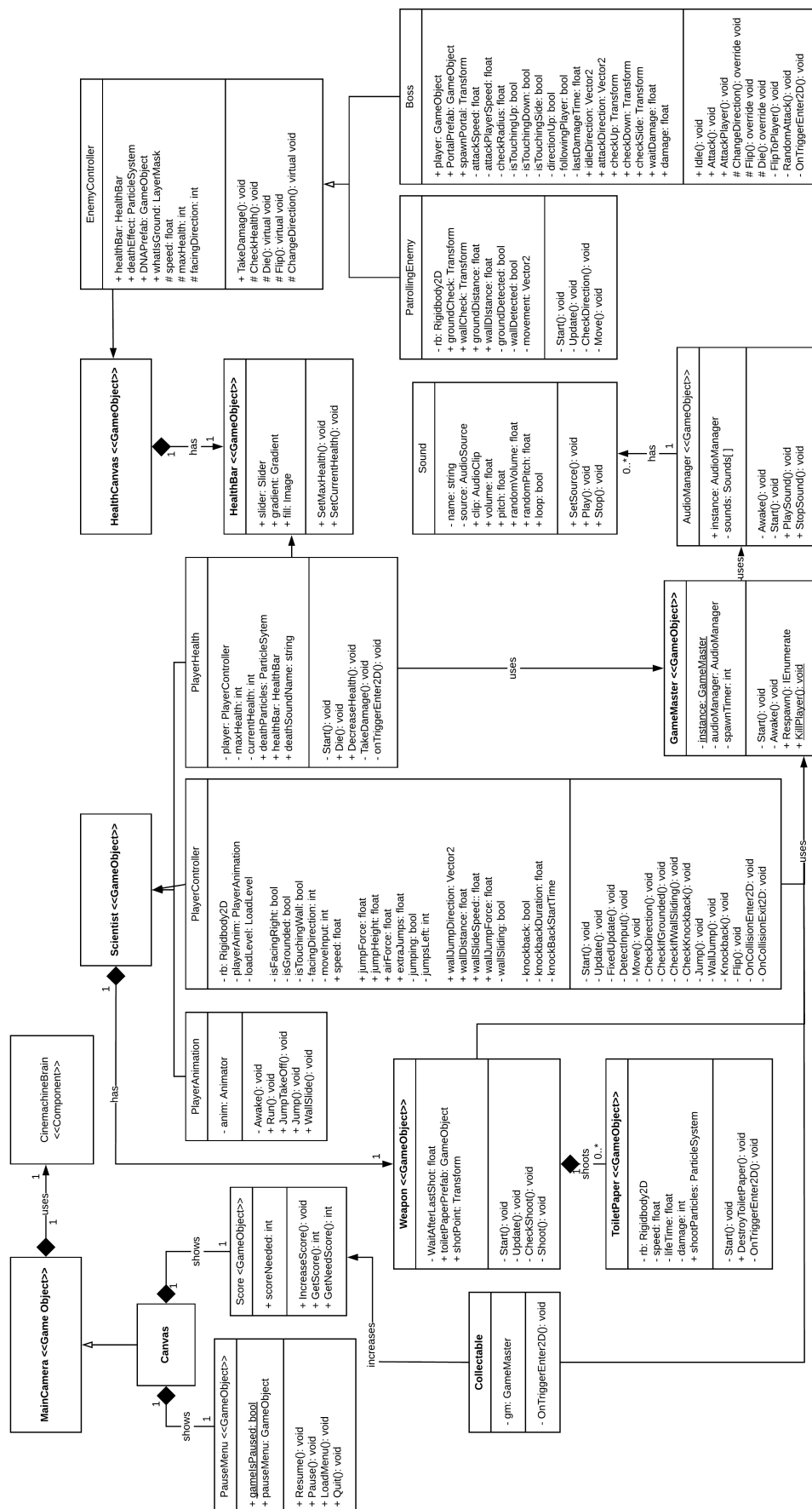


Abbildung 4.1: Endgültiges Klassendiagramm

5 Implementierung

Dieses Kapitel beschreibt chronologisch die einzelnen Stationen der Implementierungsphase.

5.1 Grafiken/Sprites erstellen

Zuerst werden die zuvor erwähnten Skizzen von Welt, Charakteren und Spielobjekten erstellt. Die Grafiken werden anschließend in einer Art "Comic-Style" digital gezeichnet. Im Vorfeld wird außerdem eine Farbpalette (Abbildung 5.1) erstellt, damit alle Farben ein stimmiges Gesamtbild ergeben. Für die Welt werden Ground-Tiles erstellt. Dabei ist es eine Herausforderung, diese so zu zeichnen, dass sie perfekt zusammenpassen und keine Übergänge zwischen den einzelnen Tiles zu sehen sind. Schließlich werden die fertigen Assets in Unity importiert und mithilfe des Sprite Editors zugeschnitten. Die Charaktere werden in Einzelteilen importiert, sodass sie sich später einfach animieren lassen. Zu Testzwecken wird vorerst eine kleine Szene erstellt. Dazu wird eine Palette aus den importierten Tiles angelegt und mit dieser dann der Ground-Layer in eine Tilemap gezeichnet. Abschließend werden der Szene die wichtigsten Spielobjekte hinzugefügt.

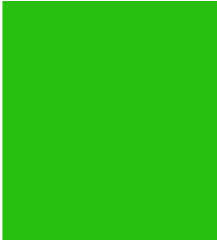
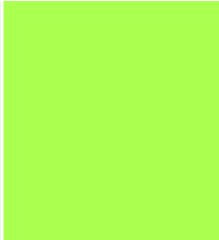
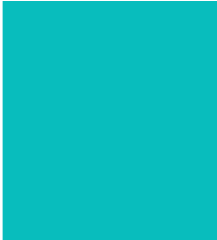
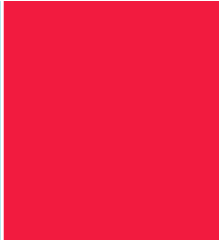

									
Color 1		Color 2		Color 3		Color 4		Color 5	
HEX	299F12	HEX	ABFF4F	HEX	08B0B0	HEX	F2185F	HEX	FF9914
RGB	41, 191, 18	RGB	171, 255, 79	RGB	8, 189, 189	RGB	242, 27, 63	RGB	255, 153, 20
HSB	112, 91, 75	HSB	89, 69, 100	HSB	180, 96, 74	HSB	350, 89, 95	HSB	34, 92, 100
CMYK	78, 0, 90, 25	CMYK	32, 0, 69, 0	CMYK	95, 0, 0, 25	CMYK	0, 88, 73, 5	CMYK	0, 40, 92, 0

Abbildung 5.1: Farbpalette erzeugt mithilfe der Website <https://colors.co>

5.2 Animationen

Im Unity Animation Fenster werden einige Animationen für die Charaktere erstellt. Der Spieler bekommt eine Run-, Jump- und WallSlide-Animation, wobei die Jump Animation nochmal in drei kleinere TakeOff, Jump und Land Animationen unterteilt wird. So können der Absprung, der eigentliche Moment in der Luft, sowie die Landung im Code zeitlich genau den jeweiligen Animationen angepasst werden. Für das Virus wird eine Idle Animation erstellt und das Bossvirus erhält Animationen für seine drei Zustände: Idle, Attack und AttackPlayer. Die erstellten Animationen werden anschließend im Unity Animator Fenster des jeweiligen Charakters miteinander verknüpft und Parameter als Bedingung für die Ausführung festgelegt.

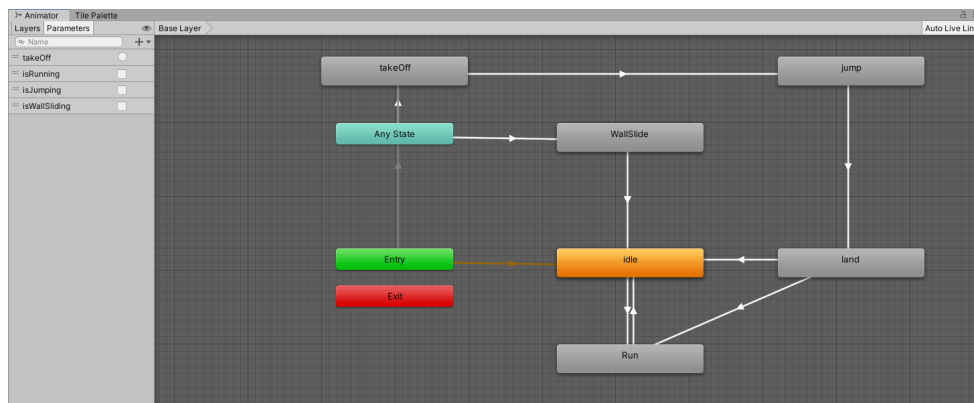


Abbildung 5.2: Spieler Animator

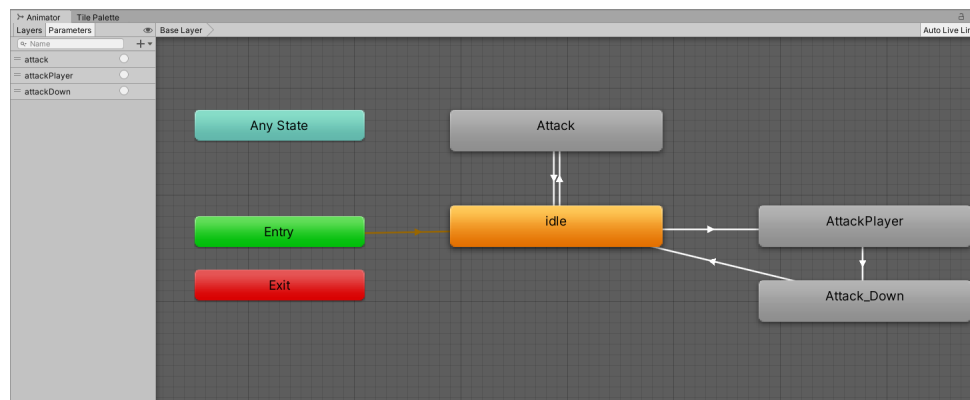


Abbildung 5.3: Bossvirus Animator

5.3 Spieler

In diesem Abschnitt wird die Implementierung der wichtigsten Funktionen des Spielers erläutert.

5.3.1 Bewegung

Der Spieler benötigt Methoden zum Erkennen des Tasten-Inputs, um die Steuerung umzusetzen. Dazu wird aus Update() die Methode DetectInput() aufgerufen. Hier wird für jeden Frame die Bewegungsrichtung in die Variable moveInput gespeichert. Der moveInput wird dann mit einer festgelegten Geschwindigkeit multipliziert und der **Rigidbody**-Geschwindigkeit des Spielers zugewiesen. Ab jetzt kann sich der Spieler nach links und rechts bewegen. In CheckDirection() wird die aktuelle Bewegungsrichtung überprüft und der Spieler dementsprechend in Laufrichtung rotiert.

```
1     private void DetectInput()
2     {
3         moveInput = Input.GetAxis("Horizontal");
4     }
5
6     private void Move()
7     {
8         rb.velocity = new Vector2(speed * moveInput,
9             rb.velocity.y);
10    }
11
12    private void CheckDirection()
13    {
14        if (isFacingRight && moveInput < 0)
15        {
16            Flip();
17        }
18        else if (!isFacingRight && moveInput > 0)
19        {
20            Flip();
21        }
22    }
23
24    private void Flip()
25    {
26        transform.Rotate(0f, 180f, 0f);
27        facingDirection *= -1;
28        isFacingRight = !isFacingRight;
```

Quelltext 5.1: Auszug aus PlayerController Skript: Bewegung

5.3.2 Jump, Double Jump und Wall Jump

Nun müssen noch die eigentlichen Fähigkeiten des Spielers implementiert werden. Für die Jumps muss überprüft werden, ob der Spieler den Boden oder eine Wand berührt. Hierzu bekommt der Spieler in Unity ein leeres GameObject als Child-Element zugeordnet, das auf Fußhöhe platziert wird. Durch einen Radius um dieses GameObject wird überprüft, ob der Spieler den Ground-Layer berührt. Zur Erkennung von Wänden, wird ein horizontaler Raycast in Spieler Blickrichtung erzeugt, der true zurückgibt, wenn er nach einer gewissen Distanz auf den Ground-Layer trifft.

In DetectInput() wird nun überprüft, ob die Leertaste gedrückt wurde. Da dem Spieler auch ein Double Jump ermöglicht werden soll, wird eine Variable jumpsLeft deklariert und mit der gewünschten Anzahl an erlaubten Sprüngen initialisiert. Falls der Spieler den Boden, aber keine Wand berührt und genügend Jumps auf dem Zähler übrig sind, wird die Jump() Funktion aufgerufen. Hier wird die y-Geschwindigkeit des Spielers festgelegt. Damit er sich nicht unendlich in y-Richtung bewegt, wird außerdem überprüft, wann die Leertaste losgelassen wird und somit auch die Sprunghöhe bestimmt.

Wurde die Leertaste gedrückt und der Spieler berührt nicht den Boden, aber stattdessen eine Wand, wird die WallJump() Funktion aufgerufen. Später mussten weitere Bedingungen ergänzt werden, da der Spieler zuerst auch einzelne Wände einfach hochspringen konnte. Um einen Wall Jump auszuführen, muss der Spieler nun stattdessen in die entgegengesetzte Richtung bewegt werden. Damit er währenddessen nicht zu schnell die Wand herunterfällt, wird die y-Geschwindigkeit des Spielers bei Berühren einer Wand auf eine festgelegte Geschwindigkeit beschränkt. Um die Steuerung zu vereinfachen, wird währenddessen auch keine Bewegung in x-Richtung zugelassen. Der Spieler haftet an der Wand bis ein Sprung ausgeführt wird oder er wieder den Boden berührt.

```
1  void Update()
2  {
3      DetectInput();
4      CheckDirection();
5      CheckIfWallSliding();
6      if (isGrounded)
7      {
8          jumpsLeft = extraJumps;
9      }
10 }
11
12 private void FixedUpdate()
13 {
14     isGrounded = Physics2D.OverlapCircle(feet.position,
15         checkRadius, whatIsGround);
16     isTouchingWall = Physics2D.Raycast(front.position,
17         transform.right, wallDistance, whatIsGround);
18     Move();
19 }
20
21 private void DetectInput()
22 {
23     moveInput = Input.GetAxis("Horizontal");
24
25     if (Input.GetButtonDown("Jump"))
26     {
27         if (isGrounded || (jumpsLeft > 0 && !isTouchingWall))
28         {
29             Jump();
30         }
31         if (!isGrounded && isTouchingWall && moveInput != 0
32             && moveInput != facingDirection)
33         {
34             WallJump();
35         }
36     }
37
38     if (!Input.GetButton("Jump") && jumping)
39     {
40         rb.velocity = new Vector2(rb.velocity.x,
41             rb.velocity.y * jumpHeight);
42         jumping = false;
43     }
44
45 private void Move()
46 {
47     rb.velocity = new Vector2(speed * moveInput,
48         rb.velocity.y);
49 }
```

```
45         if (!isGrounded && !wallSliding && moveInput != 0)
46         {
47             Vector2 addForce = new Vector2(airForce * moveInput,
48                 0);
49             rb.AddForce(addForce);
50         }
51         if (wallSliding)
52         {
53             if (rb.velocity.y < -wallSlideSpeed)
54             {
55                 rb.velocity = new Vector2(0, -wallSlideSpeed);
56             }
57         }
58     }
59
60     private void Jump()
61     {
62         rb.velocity = new Vector2(rb.velocity.x, jumpForce);
63         jumpsLeft--;
64         jumping = true;
65     }
66
67     private void CheckIfWallSliding()
68     {
69         if (isTouchingWall && !isGrounded)
70         {
71             wallSliding = true;
72         }
73         else
74         {
75             wallSliding = false;
76         }
77     }
78
79     private void WallJump()
80     {
81         rb.velocity = new Vector2(rb.velocity.x, 0);
82         Vector2 force = new Vector2(wallJumpForce *
83             wallJumpDirection.x * moveInput, wallJumpForce *
84             wallJumpDirection.y);
85         rb.AddForce(force, ForceMode2D.Impulse);
86         wallSliding = false;
87         jumping = true;
88     }
```

Quelltext 5.2: Auszug aus PlayerController Skript: Ergaenzung von Jumps

5.3.3 Health

Der Spieler bekommt aus Gründen der Übersichtlichkeit ein separates Skript für Schaden und Gesundheit. Er erhält außerdem einen HealthBar, der mit einfachen Standard-UI Elementen in Unity erstellt wird. Dieser kann dann je nach Gebrauch ein- oder ausgeblendet werden. Zu Beginn wird die Gesundheit des Spielers festgelegt und der HealthBar auf den selben Wert gesetzt. Erleidet der Spieler Schaden, wird der jeweilige Wert von der Gesundheit abgezogen und der Balken entsprechend angepasst. Sinkt die Gesundheit unter 0 oder kollidiert der Spieler mit einem **GameObject**, das den **Tag** Respawn oder Enemy besitzt, wird die Funktion Die() aufgerufen. Hier wird ein zuvor erstelltes Partikelsystem aus einem Prefab instanziiert und die Funktion KillPlayer() aus dem in 5.6 erläuterten GameMaster Skript aufgerufen.

```
1      public void DecreaseHealth(float amount)
2      {
3          currentHealth -= (int)amount;
4          if (currentHealth <= 0)
5          {
6              Die();
7          }
8          healthBar.SetCurrentHealth(currentHealth);
9      }
10
11     private void TakeDamage(int amount)
12     {
13         DecreaseHealth(amount);
14     }
15
16     public void Die()
17     {
18         Instantiate(deathParticles, transform.position,
19                     Quaternion.identity);
20         GameManager.instance.KillPlayer(player);
21     }
22
23     private void OnTriggerEnter2D(Collider2D other)
24     {
25         if (other.CompareTag("Respawn") ||
26             other.CompareTag("Enemy"))
27         {
28             Die();
29         }
30     }
```

Quelltext 5.3: Auszug aus PlayerHealth Script

5.3.4 Animation

Um die zuvor in 5.2 erstellten Animationen auch per Skript auslösen zu können wird ein getrenntes PlayerAnimation Skript erstellt, auf das aus dem PlayerController zugegriffen werden kann. Um die Animationen zu starten, werden hier die entsprechenden Parameter gesetzt und so die gewünschte Animation gestartet. In Abbildung 5.4 wird als Beispiel der Code für das Setzen der Run-Animation gezeigt.

```
1      public void Run(bool run)
2      {
3          anim.SetBool("isRunning", run);
4      }
```

Quelltext 5.4: Auszug aus dem PlayerAnimation Skript

5.4 Waffe

Aus Update() wird die Methode CheckShoot() aufgerufen. Hier soll die Position der Maus in Relation zur Position des Spielers herausgefunden werden. Da zuerst die Bildschirmposition der Maus auf die Unity Welt transformiert werden muss, wird die Funktion Camera.main.ScreenToWorldPoint(Input.mousePosition) verwendet. Von der Mausposition wird anschließend die Position des Spielers abgezogen und das Ergebnis in einen Vector gespeichert. Aus diesem wird die benötigte z-Rotation der Waffe berechnet. Die Rotation der Waffe wird somit immer auf die Mausposition abgestimmt. Es wird eine Variable zum Speichern der vergangenen Zeit seit dem letzten Schuss deklariert. Ist genug Zeit vergangen, wird Shoot() aufgerufen. In Shoot() wird das Toilettenpapier-Projektile instanziiert und die Zeit zurückgesetzt.

In der Start() Funktion des ToilettenPapier-Skripts bekommt dessen **Rigidbody** direkt eine Geschwindigkeit zugewiesen. Trifft es einen Gegner, wird dessen TakeDamage() Funktion aufgerufen, an dieser Stelle ein Partikelsystem instanziiert und das **GameObject** zerstört. Ansonsten zerstört es sich nach einer festen Lifetime selbst.

```
1     private void CheckShoot ()
2     {
3         Vector3 mousePosition =
4             Camera.main.ScreenToWorldPoint (Input.mousePosition) -
5             transform.position;
6         float rotationZ = Mathf.Atan2(mousePosition.y,
7             mousePosition.x) * Mathf.Rad2Deg;
8         transform.rotation = Quaternion.Euler(0, 0, rotationZ);
9
10        if (WaitAfterLastShot <= 0)
11        {
12            if (Input.GetButtonDown("Fire1"))
13                Shoot ();
14        }
15        else
16        {
17            WaitAfterLastShot -= Time.deltaTime;
18        }
19    }
20
21    private void Shoot ()
22    {
23        Instantiate(toiletPaperPrefab, shotPoint.position,
24            transform.rotation);
25        WaitAfterLastShot = 0.5f;
26    }
```

Quelltext 5.5: Auszug aus dem Weapon Skript

```
1 private void Start()
2 {
3     rb = GetComponent<Rigidbody2D>();
4     rb.velocity = transform.right * speed;
5     Invoke("DestroyToiletPaper", lifeTime);
6 }
7
8 private void OnTriggerEnter2D(Collider2D other)
9 {
10     if (other.CompareTag("Enemy") || other.CompareTag("Boss"))
11     {
12         EnemyController enemy =
13             other.GetComponent<EnemyController>();
14         if (enemy != null)
15         {
16             enemy.TakeDamage(damage);
17             DestroyToiletPaper();
18         }
19     }
20
21 public void DestroyToiletPaper()
22 {
23     Instantiate(shootParticles, transform.position,
24                 Quaternion.identity);
25     Destroy(gameObject);
26 }
```

Quelltext 5.6: Auszug aus dem ToiletPaper Skript

5.5 Gegner

Für alle Gegner wird ein `EnemyController` Script implementiert, von dem die verschiedenen Arten von Gegnern erben. Hier werden die wichtigsten gemeinsamen Variablen und Funktionen deklariert. In `CheckHealth()` wird überprüft, ob die Gesundheit des Gegners unter 0 sinkt. Falls ja, wird `Die()` aufgerufen. Aus den anderen erbenden Skripten kann der Gegner durch Aufrufen von `Flip()` und `ChangeDirection()` die Bewegungsrichtung ändern. Aus dem Toilettenpapier Skript wird bei Kollision mit einem Gegener die Funktion `TakeDamage()` aufgerufen, die einen bestimmten Betrag von der Gesundheit des Gegner abzieht und den Lebensbalken entsprechend senkt.

5.5.1 Virus/Patrolling Enemy

Das kleine Virus soll Boden und Wände erkennen können. Dafür werden ihm zwei `GameObjects` als Child-Elemente hinzugefügt, von denen dann jeweils ein Ray (2.2.5) Richtung Boden und Seite gecastet wird, der eine Kollision mit dem Ground-Layer feststellt. So werden Boden und Wände erkannt. Falls kein Boden oder eine Wand erkannt wird, ändert das Virus seine Bewegungsrichtung. Die Bewegungsgeschwindigkeit kann im Inspector des Gameobjects festgelegt werden.

```
1     private void CheckDirection()
2     {
3         groundDetected = Physics2D.Raycast(groundCheck.position,
4             Vector2.down, groundDistance, whatIsGround);
5         wallDetected = Physics2D.Raycast(wallCheck.position,
6             transform.right, wallDistance, whatIsGround);
7
8         if (!groundDetected || wallDetected)
9         {
10             Flip();
11             ChangeDirection();
12         }
13         else
14         {
15             Move();
16         }
17     }
18     private void Move()
19     {
20         rb.velocity = new Vector2(speed * facingDirection,
21             rb.velocity.y);
22     }
23 }
```

Quelltext 5.7: Auszug aus dem Patrolling Enemy Skript

5.5.2 Boss

Das Bossvirus soll sich durch die Luft bewegen und seine Bewegungsrichtung ändern, sobald eine Wand erkannt wird. Dazu bekommt es ebenfalls drei leere GameObjects als Child-Elemente in Unity zugewiesen, die dann von allen Seiten mithilfe eines Raycasts Wände erkennen können.

Das Virus kann immer einen von drei Zuständen annehmen: Idle, Attack und AttackPlayer. Für jeden dieser Zustände gibt es eine Animation. Die Animationen erhalten alle ein StateMachineBehaviour Script, das bei starten der zugehörigen Animation ausgeführt wird. In diesem wird dann die passende Funktion im Boss Script aufgerufen. Der Gegner startet mit einer 3 Sek. Idle Animation. Am Ende der Idle Animation wird ein Event gesetzt und so direkt die Funktion randomAttack() aufgerufen. Diese wählt zufällig einen der anderen beiden Zustände aus, spielt dessen Animation ab und führt die zugehörige Funktion aus. Anschließend wird wieder die Idle Animation abgespielt und der Kreislauf beginnt von vorne.

Währenddessen verfolgt der Boss die Position des Spielers. In FlipToPlayer() wird die Position des Spielers in Relation zur Position des Gegners berechnet und je nachdem die Bewegungs- und Blickrichtung des Virus bestimmt.

Kollidiert der Gegner mit dem Spieler, so wird erst überprüft, ob eine gewisse Zeit seit dem letzten Schaden vergangen ist (Pro Attacke nur einmal die Gesundheit des Spielers reduzieren). Falls das der Fall ist, wird die TakeDamage() Funktion des Spielers aufgerufen.

```
1      public void Idle()
2      {
3          if (topDetected && directionUp)
4          {
5              ChangeDirection();
6          }
7          else if (bottomDetected && !directionUp)
8          {
9              ChangeDirection();
10         }
11         if (wallDetected)
12         {
13             if (facingLeft)
14             {
15                 Flip();
16             }
17             else if (!facingLeft)
18             {
19                 Flip();
20             }
21         }
```

```
22         rb.velocity = speed * idleDirection;
23     }
24
25     public void Attack()
26     {
27         if (topDetected && directionUp)
28         {
29             ChangeDirection();
30         }
31         else if (bottomDetected && !directionUp)
32         {
33             ChangeDirection();
34         }
35         if (wallDetected && facingLeft)
36         {
37             Flip();
38         }
39         else if (!facingLeft && wallDetected)
40         {
41             Flip();
42         }
43         rb.velocity = attackSpeed * attackDirection;
44     }
45
46     public void AttackPlayer()
47     {
48         if (!followingPlayer)
49         {
50             playerPos = player.transform.position -
51                 transform.position;
52             playerPos.Normalize();
53             followingPlayer = true;
54         }
55         if (followingPlayer)
56         {
57             rb.velocity = playerPos * attackPlayerSpeed;
58         }
59         if (wallDetected || bottomDetected)
60         {
61             rb.velocity = Vector2.zero;
62             followingPlayer = false;
63             bossAnim.SetTrigger("attackDown");
64         }
65     }
66
67     private void randomAttack() // called from idle animation
68     {
69         switch (Random.Range(0, 2))
```

```
70         case 0:
71             bossAnim.SetTrigger("attack");
72             break;
73         case 1:
74             bossAnim.SetTrigger("attackPlayer");
75             break;
76     }
77 }
78
79 private void OnTriggerEnter2D(Collider2D other)
80 {
81     if (other.CompareTag("Player"))
82     {
83         if (Time.time >= lastDamageTime + waitDamage) // wait
84             for some Time before next Player Damage
85         {
86             lastDamageTime = Time.time;
87             other.gameObject.SendMessage("TakeDamage",
88                 damage);
89         }
90     }
91 }
```

Quelltext 5.8: Auszug aus dem Boss Skript

5.6 Manager

5.6.1 GameMaster

Der GameMaster bleibt über alle Szenen immer erhalten. Auf die Funktion KillPlayer() kann von allen anderen Klassen zugegriffen werden. In ihr wird das Spieler Objekt zerstört und Respawn() aufgerufen. Nach einer Wartezeit von 2 Sekunden wird das Level neu geladen und der Spieler am Anfang des Levels gespawnt.

```
1     public IEnumerator Respawn()
2     {
3         yield return new WaitForSeconds(SpawnTimer);
4         SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
5     }
6
7     public void KillPlayer(PlayerController player)
8     {
9         Destroy(player.gameObject);
10        instance.StartCoroutine(instance.Respawn());
11    }
```

Quelltext 5.9: Auszug aus dem GameMaster Skript

5.6.2 AudioManager

Der AudioManager wird der Szene als GameObject mit dem AudioManager Skript hinzugefügt. In der Skript Komponente kann ein Array mit verschiedenen Audiodateien gefüllt werden, um die Verwaltung/Verwendung von Musik und Soundeffekten zu vereinfachen. So muss nur die Funktion PlaySound(string soundName) mit dem Namen des gewünschten Titels aus anderen Skripten aufgerufen werden, um eine Audiodatei abzuspielen. (Der Audiomanager wurde aus einem Youtube Tutorial weitestgehend übernommen: <https://www.youtube.com/watch?v=HhFKtiRd0qI>)

5.7 UI

5.7.1 Score

Für die Anzeige der Punktzahl wird dem **Canvas** in der oberen linken Ecke ein Textfeld hinzugefügt. Hier soll der Score angezeigt werden. Bei Einsammeln von DNA wird die Methode `IncreaseScore()` aufgerufen, die Punktzahl um 1 erhöht und der Text für die UI festgelegt.

```
1      public void IncreaseScore()
2      {
3          score += 1;
4          scoreText.text = score.ToString() + "/" + scoreNeeded;
5      }
```

Quelltext 5.10: Auszug aus dem Score Skript

5.7.2 HealthBar

Jedem Charakter wird ein eigener **Canvas** mit einem HealthBar als Child-Objekt hinzugefügt. Es gibt 2 Funktionen, die von anderen Skripten aufgerufen werden können. So kann der max. Wert für den Balken festgelegt und je nach Gesundheit des Charakters reduziert werden. (Das Skript für den Health Bar wurde aus einem Youtube Tutorial übernommen: https://www.youtube.com/watch?v=BLfNP4Sc_iA)

```
1      public void SetMaxHealth(int health) {
2          slider.maxValue = health;
3          slider.value = health;
4          fill.color = gradient.Evaluate(1f);
5      }
6      public void SetCurrentHealth(int health) {
7          slider.value = health;
8          fill.color = gradient.Evaluate(slider.normalizedValue);
9      }
```

Quelltext 5.11: Auszug aus dem HealthBar Skript

5.8 Spielobjekte

Es sind auch einige kürzere Skripte für die Spielobjekte der Welt entstanden.

- BreakableBox

Kollidiert der Spieler von unten mit einer Box, wird ein Partikelsystem und DNA Objekt instanziiert und die Box zerstört.

- MovingPlatform

Die Plattform bekommt zwei leere GameObjects als Child-Elemente. Sie sind die beiden Positionen, zwischen denen sich die Plattform bewegen kann. Die Geschwindigkeit kann im Inspector für jede Plattform individuell festgelegt werden. Als nächste Position wird immer diejenige festgelegt, an der sich die Plattform gerade nicht befindet.

- Collectable/DNA

Bei Kollision mit Spieler wird das GameObject zerstört und erhöht den Score.

- SpawnPoint

Das Spawn-Portal wird nach Laden des Levels angezeigt und nach 3 Sekunden zerstört.

- Rotation

Ein kurzes Skript für die Rotation der Sägeblätter und anderer GameObjects. Im Inspector kann die Geschwindigkeit und Richtung der Rotation eingestellt werden.

- CheckScore

Unsichtbares GameObject, das kurz vor Ende des Levels platziert wird. Läuft der Spieler durch diese Art Schranke, werden `getScore()` und `GetScoreNeeded()` aus dem Score Script aufgerufen und so die Punktzahl überprüft. Wenn genügend DNA gesammelt wurde, öffnet sich das Portal zum nächsten Level, sonst startet das Level neu.

5.9 Level

Wenn die wichtigsten Funktionen implementiert sind, können die endgültigen Level entworfen werden. Dazu werden dem Projekt zwei Szenen hinzugefügt. Die Level werden aus einem Grid aus **Tiles** und den verschiedenen Spielobjekten zusammengestellt.

5.10 Kameraführung

Um die Kameraführung besonders angenehm zu machen, kommt zusätzlich zur MainCamera das Unity Package Cinemachine zum Einsatz. Der Szene wird eine virtuelle Kamera mit vielen verschiedenen Einstellungen hinzugefügt, an der die MainCamera orientiert wird. So kann bspw. die Distanz oder eine Dead Zone definiert werden, in welcher sich der Spieler bewegen kann, ohne dass die Kamera direkt mitbewegt wird. Außerdem lässt sich durch die Komponente Cinemachine Confiner eine Region festlegen, in der sich die Kamera bewegen kann. So bleibt die Kamera immer innerhalb der angegebenen Begrenzung und bewegt sich nicht aus der Welt. Dafür bekommt der Hintergrund einen Polygon Collider, welcher diese Region definiert.

5.11 GUI

Zum Abschluss wird sowohl ein Hauptmenü, als auch ein Pause Menü gestaltet. Für das Hauptmenü wird eine eigene neue Szene mit Hintergrund und jeweils einem UI-Button für Start und Beenden des Spiels hinzugefügt. Das Pause Menü wird dem Canvas Prefab hinzugefügt und initial ausgeblendet. Bei Drücken der ESC-Taste wird das Menü eingeblendet und die Zeit mit `Time.timescale = 0` angehalten. Im Pause Menü kann entweder zum Hauptmenü gesprungen werden oder das Spiel über die jeweiligen Buttons entweder fortgesetzt oder beendet werden.

5.12 Sound

Als letzten Schritt wurden dem Spiel noch die Sounds hinzugefügt. Dazu wurden Musik und Effekte aus externen Quellen heruntergeladen und dem AudioManager hinzugefügt. An den entsprechenden Stellen im Code wird dann `PlaySound()` mit dem gewünschten Titel aufgerufen.

6 Evaluierung

Die in Kapitel 3 gestellten Anforderungen konnten weitestgehend erfüllt werden. Die zuerst geplante Levelübersicht mit Speicher-/Ladesystem zur Anzeige des Fortschritts war in, sowie eine größere Anzahl an Waffen war in der vorgegebenen Zeit nicht mehr umsetzbar. Außerdem fehlen noch einige Soundeffekte und evtl. ein Abwechseln der Hintergrundmusik, damit das Spiel mit der Zeit nicht zu monoton wird.

7 Ergebnisse und Ausblick

7.1 Erreichte Ergebnisse

Innerhalb der zweimonatigen Implementierungsphase ist ein vollständiges, leicht bedienbares Spiel mit Hauptmenü, Pausemenü und vorerst zwei Leveln entstanden. Die jeweiligen **Scenes** sind im Folgenden als Screenshot eingefügt:



Abbildung 7.1: Hauptmenü

Nach Spielstart wird zuerst das Hauptmenü aufgerufen. Hier kann das erste Level gestartet oder das Spiel beendet werden.

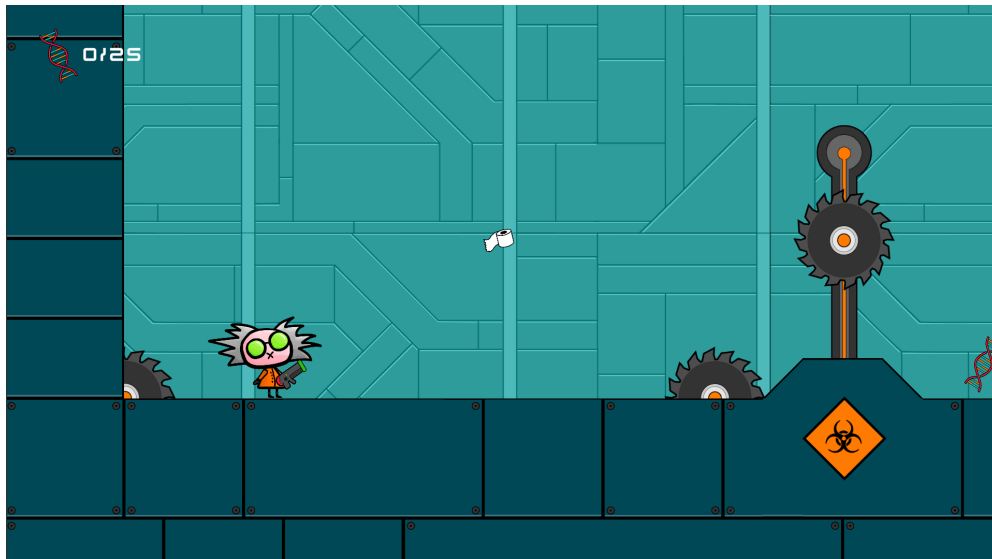


Abbildung 7.2: Level 1: In-Game

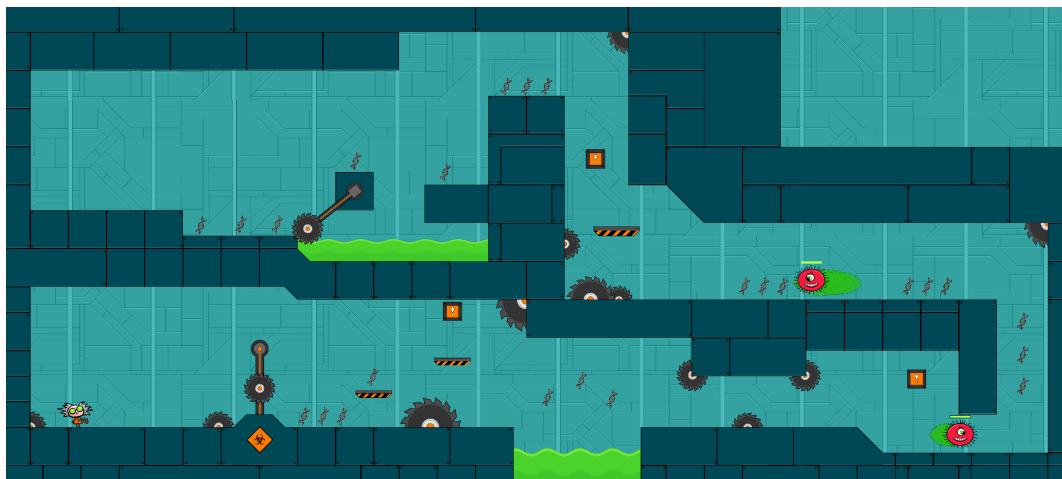


Abbildung 7.3: Level 1: Übersicht über das gesamte Level

Wurde der Start-Button gedrückt, beginnt das Spiel nach einem kurzen Loading Screen mit dem ersten Level. Da bisher nur zwei Level existieren, ist das erste Level im Moment etwas größer und schwieriger als ursprünglich geplant, um alle Assets und Funktionen zu verwenden, die erstellt/implementiert wurden. Nach weiterer Entwicklung würde das Spiel zuerst mit ein paar einfacheren Einführungsleveln beginnen. Nach erfolgreichem Abschluss des Levels öffnet sich ein Portal zum nächsten Level.

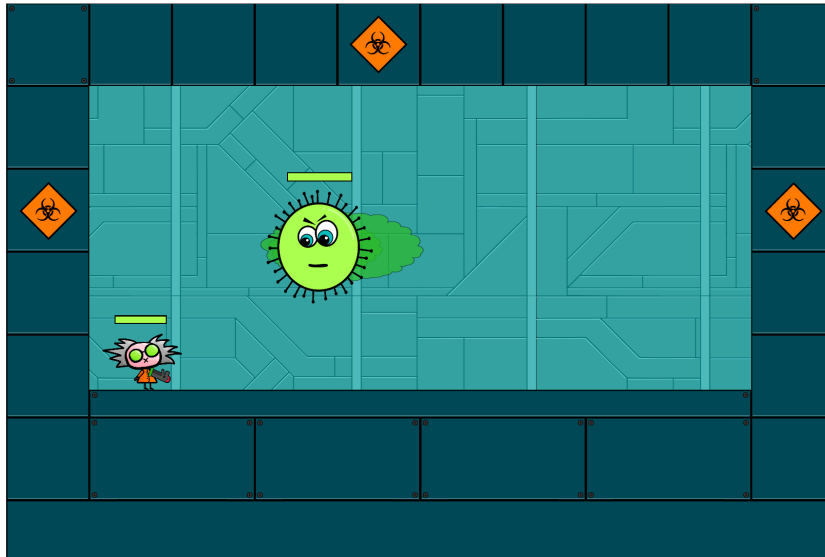


Abbildung 7.4: Level 2: Virenboss

Im nächsten Level muss dann der Virenboss besiegt werden. Der Spielercharakter hat nun auch einen Lebensbalken, um den erlittenen Schaden verfolgen zu können. Wird der Boss besiegt, öffnet sich wieder ein Portal. Allerdings endet das Spiel vorerst an dieser Stelle und lädt das Hauptmenü.



Abbildung 7.5: Pause Menü

Das Spiel kann jederzeit mit der ESC-Taste pausiert werden. Hier kann das Spiel fortgesetzt oder beendet, sowie zum Menü gesprungen werden.

7.2 Ausblick

7.2.1 Erweiterbarkeit der Ergebnisse

- Weitere Level
- Tutorial zum Erlernen der Steuerung und des Spielprinzips
- Ein Speicher- und Ladesystem, damit der Spielspaß durch ständigen Neuanfang nicht verdorben wird
- Denkbar wären außerdem noch mehr Fallen/Todesursachen
- Ein Angriffssystem für die gewöhnlichen Viren inkl. Erkennen des Spielers
- Mehr/bessere Sound Effekte und Musik
- Mobile Steuerung und Umstellung der Plattform auf Android

7.2.2 Übertragbarkeit der Ergebnisse

Die Grundmechanik des Spiels und Scripte für Spieler, Gegner und Objekte können grundsätzlich auf jedes andere Unity 2D Spiel übertragen und wiederverwendet werden.