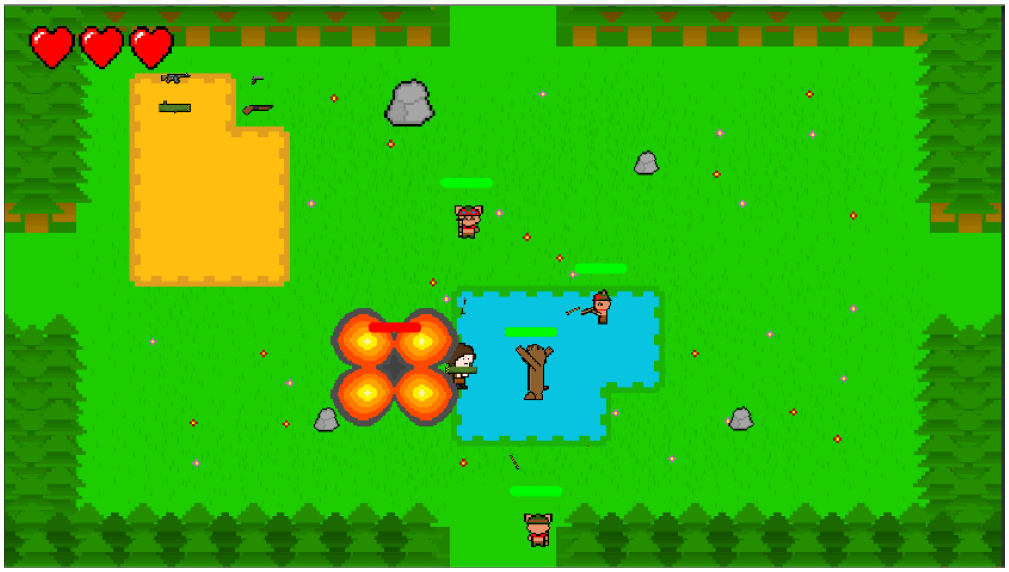
2. Milestone 1: spielbarer Prototype

Nachdem unser Konzept stand, haben wir als Gruppe beschlossen, dass wir als Erstes einen kleinen spielbaren Prototyp haben wolle in welchem wir unsere Spielidee und unser Konzept testen können.

Grund dafür war, dass man so relativ schnell feststellen kann, ob unser Konzept überhaupt funktioniert und dem Spieler spaß mach kann.

Sozusagen wurde an diesen Punkt entschieden, ob wir das Spiel so weiter führen können, wie geplant oder ob wir fundamentale Änderung vornehmen müssen.



Unser derzeitiger Stand ist ein Raum mit Spielelementen, wie einen Teich, einer Sandbank und diverse Felsen.
Zu diesen reiht sich eine saftige Wiese, die von Bäumen als Raumtrenner umgeben ist.
Unsere Hauptfigur muss sich im Prototyp – Raum gegen eine Armee von explodierenden Lilabären, aggressiven, auf Nahkampf fixierten Braunbären
und Pfeile schießende, kleine Hamster durchsetzen.
Dafür steht dem Spieler ein Arsenal an Waffen zur Verfügung: Pistole, MG, Shotgun, Raketenwerfer.

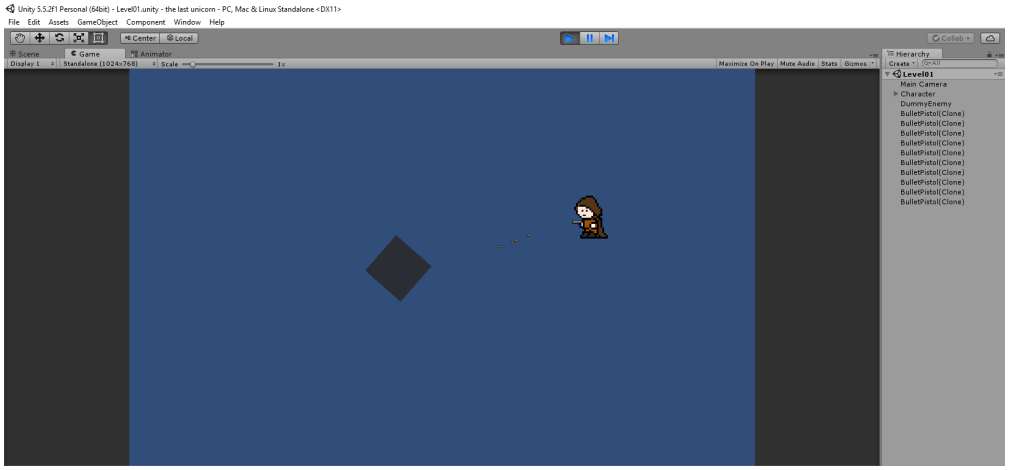
Der Spieler erleidet passend zum Angriff schaden, ebenso kann der Spieler den Tieren schaden.
Der Spieler erhält Feedback, wenn er den Tieren Schaden zufügt oder sie tötet.

Ein Menü zum Start, bei Pause und beim Tod steht dem Spieler zur Verfügung.

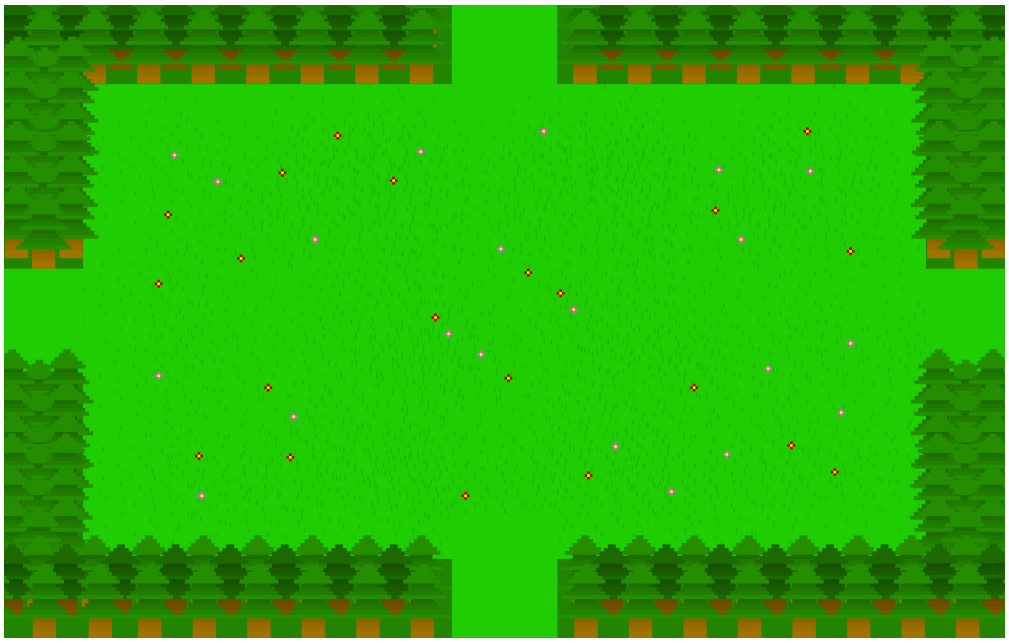
In Anbetracht unseres derzeitigen Standes sind wir überzeugt, dass wir ein gutes und spaßiges Spiel entwickeln können und halten am bestehenden Konzept fest.

Hier folgt eine Zusammenfassung, wie der Prototyp entstanden ist:

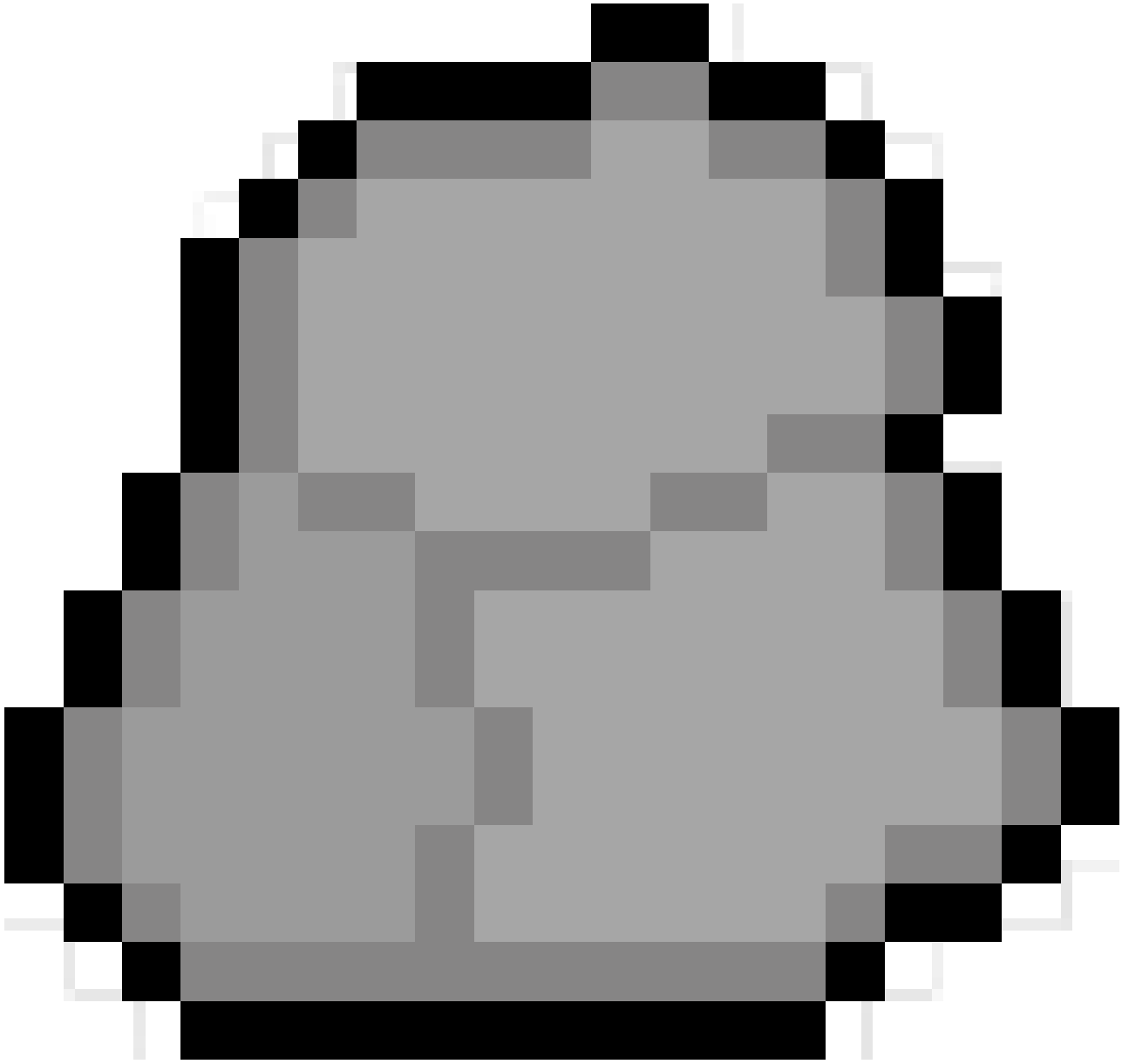
Als erstes wurde das Unity Projekt erstellt, ein Dummy und unsere Skizze des Main Charakters eingefügt.
Dazu wurden dann die ersten Einstellungen, wie Collider vorgenommen und einfache Skripte angepasst.

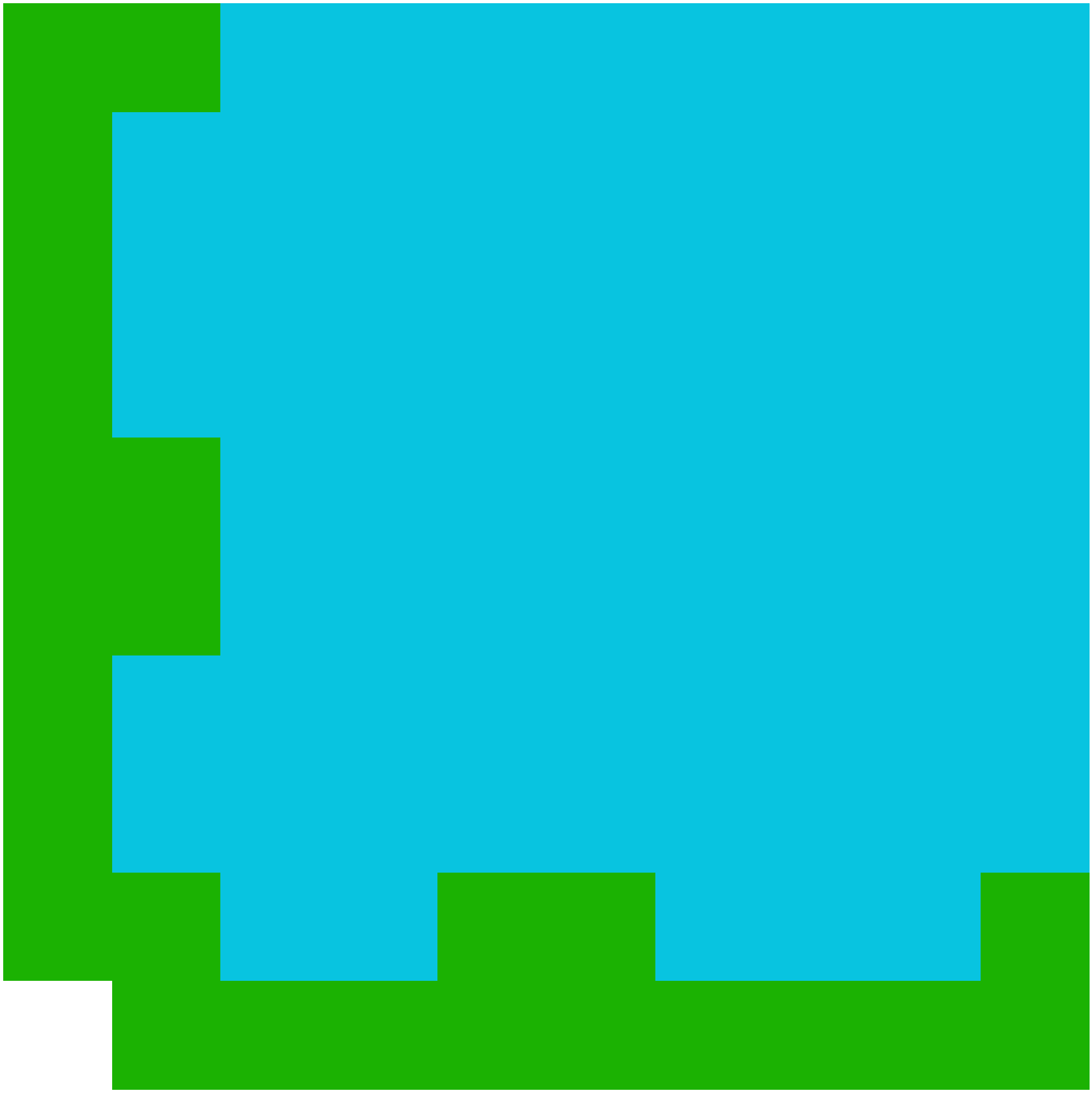


Daraufhin haben wir begonnen, die Spielwelt zu entwickeln.
Grundstein dafür war der Boden und die Wände.

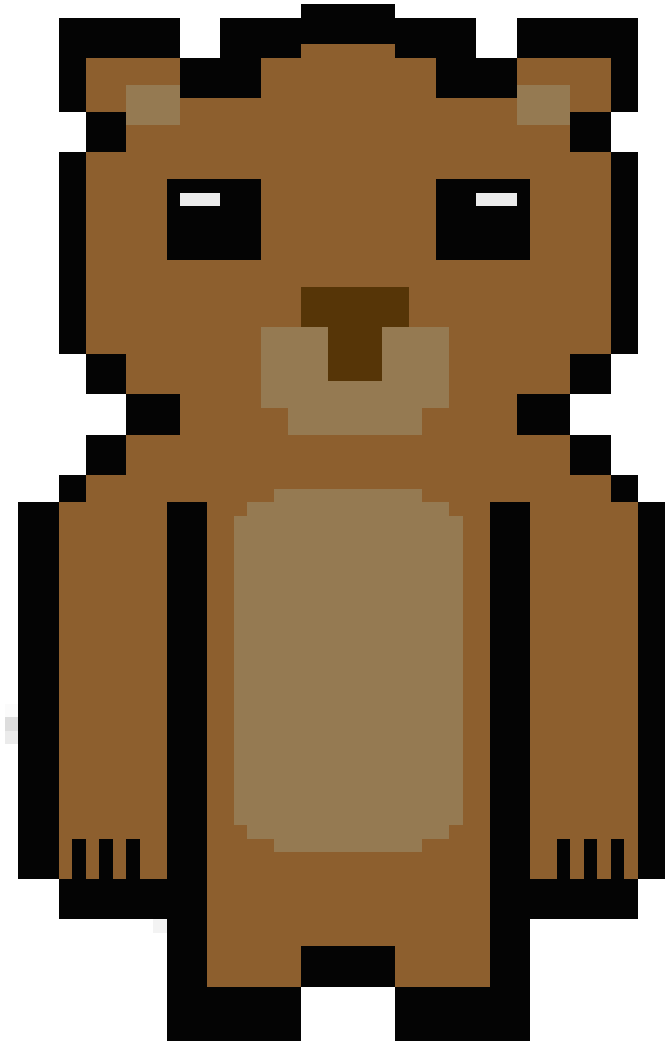


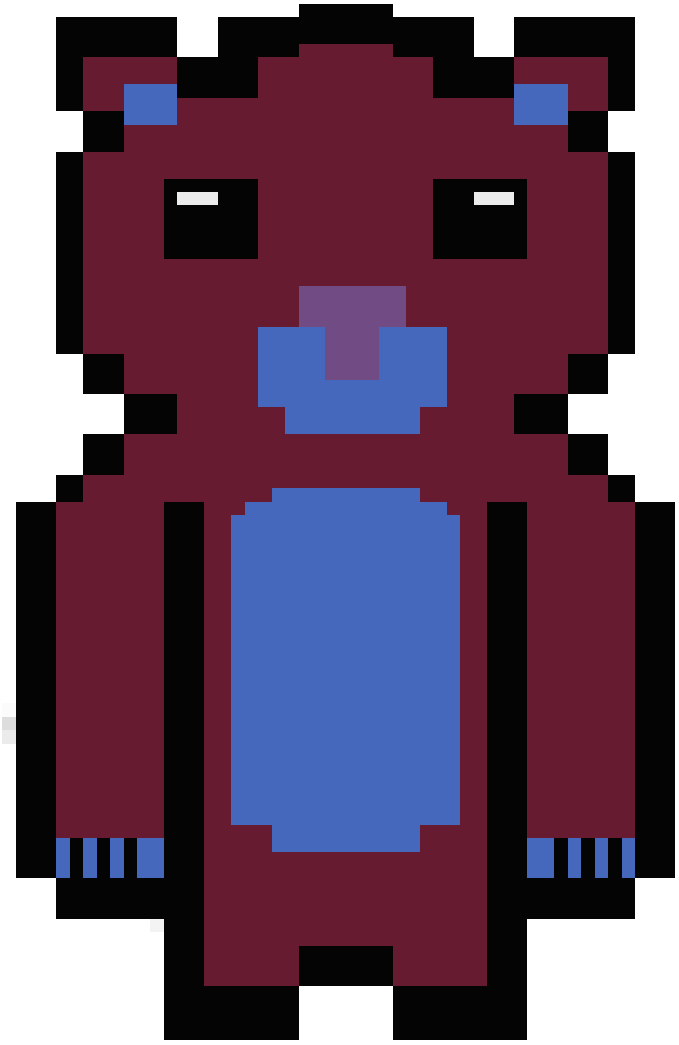
Um die Welt noch lebendiger aussehen zu lassen haben wir einige Spielobjekte konzeptioniert,
erstellt und in die Welt implementiert.





Abgerundet wird der Prototypraum
von einer Auswahl an Gegnern:

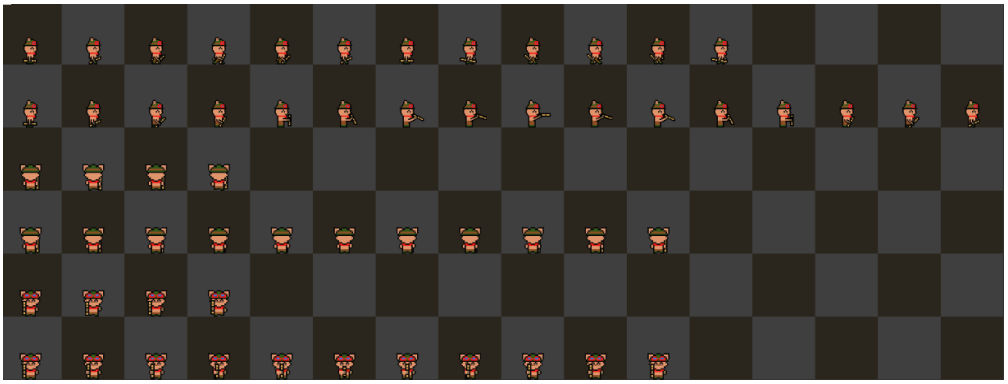




Um einen tieferen Einblick in die Entwicklung zu ermöglichen, gewähren wir eine kleine Darstellung dessen am Beispiel unseres pfeilschiessenden Hamsters.

Angefangen haben wir mit einer ersten Skizze.

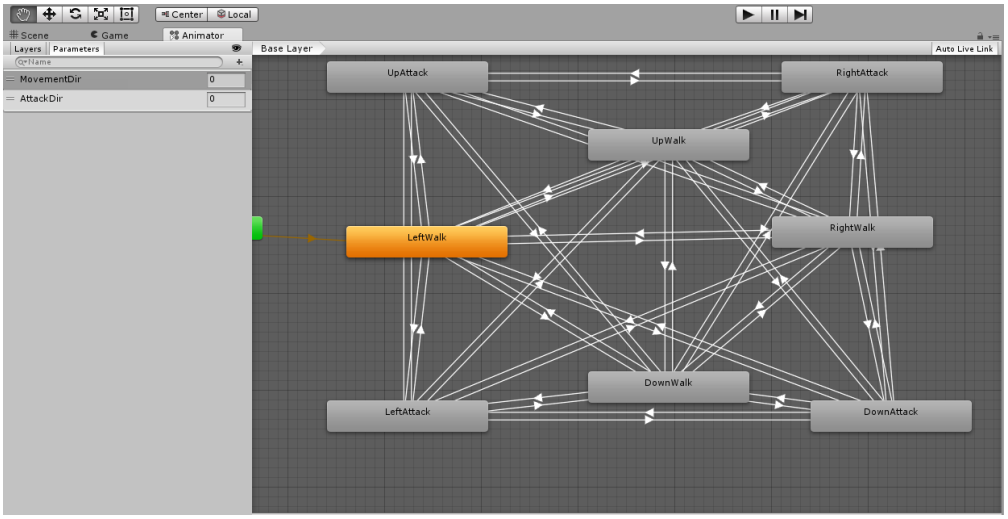
Auf dieser Skizze haben wir den Charakter aufgebaut und seine Walkcycles passend zum Grundentwurf ausgearbeitet.
Der Charakter hat 6 Walkcycles, die wir alle auf ein „Atlas“ gespeichert haben, um „draw-calls“ von Unity zu vermindern und somit eine höhere Performance zu gewährleisten.



Der Atlas in Animationsabläufe übersetzt sieht wie folgt aus.

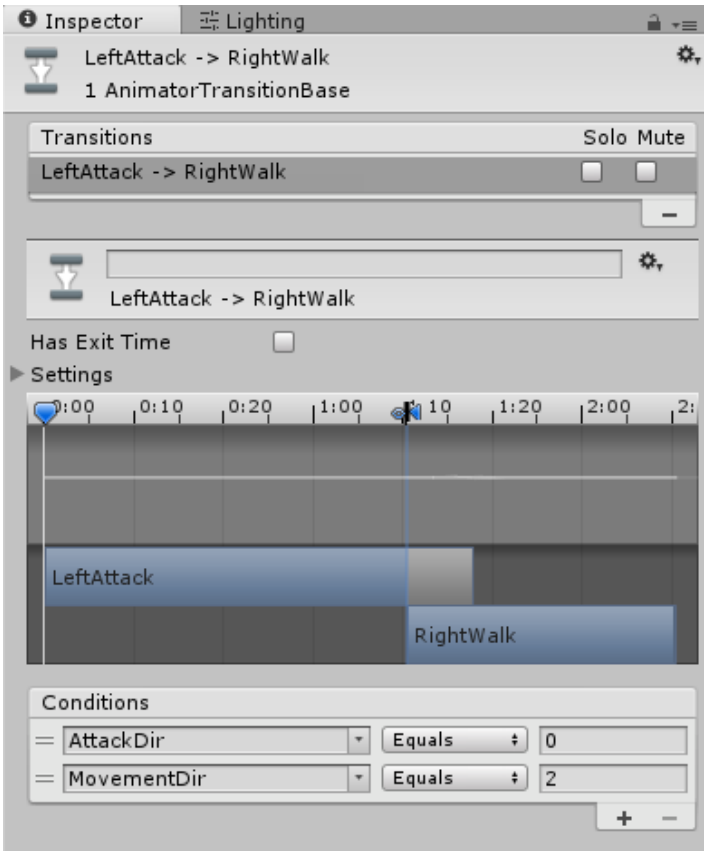
Animationen aus der frühen Phase, damals noch fehlerhaft

Nun müssen wir Unity mitteilen, woher er wissen soll, wann er welche Animation abspielen soll.
Dafür nutzen wir den Animator.



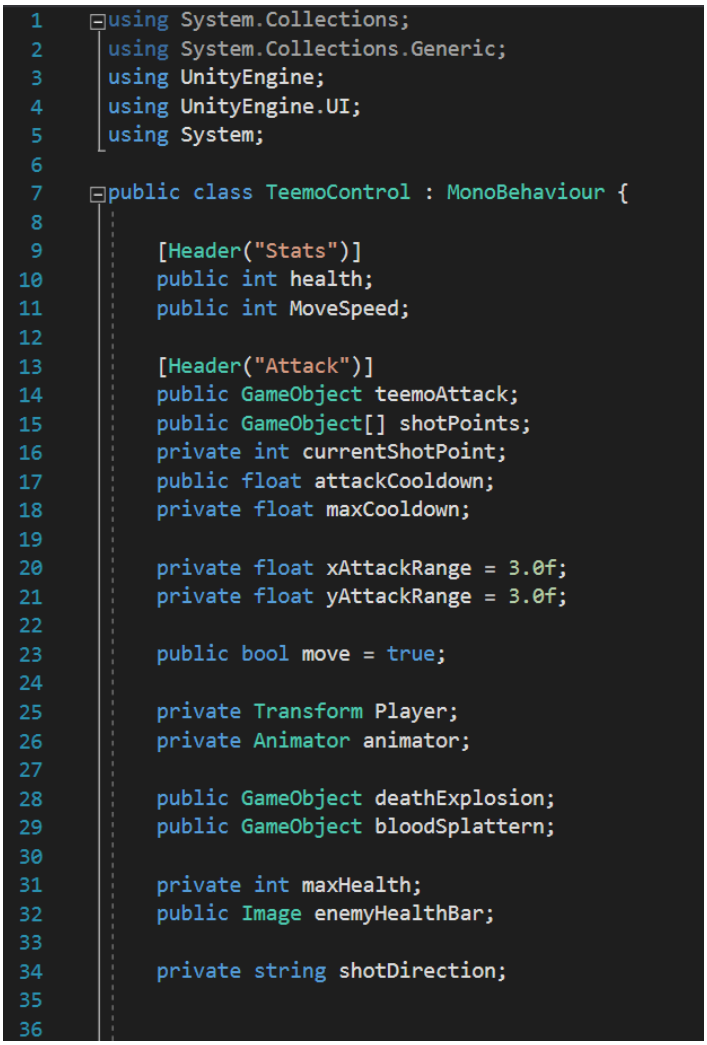
Die Variable MoveDir vom Typ Integer erhält, je nachdem in welche Richtung der Hamster laufen soll, einen bestimmten Wert (oben = 1; rechts = 2, unten = 3, links = 4).
Selbes gilt für die Variable AttackDir, welche dem Animator sagen soll, in welche Richtung der Hamster angreifen soll.

Die einzelnen Animationen verbinden wir miteinander und geben dem Spiel die Anweisung,
von der alten Animation zu der neuen Animation zu wechseln, wenn AttackDir oder MovementDir den Wert X erhält.



Nachdem wir die Animation erstellt und Unity mitgeteilt haben, wann er welche Animation einsetzten soll, kümmern wir uns um das Verhalten von unserem Hamster, anders gesagt, wir entwickeln seine künstliche Intelligenz.

Im Folgenden werden wir das Script, was dafür zuständig ist, erklären:



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System;

public class TeemoControl : MonoBehaviour {

    [Header("Stats")]
    public int health;
    public int MoveSpeed;

    [Header("Attack")]
    public GameObject teemoAttack;
    public GameObject[] shotPoints;
    private int currentShotPoint;
    public float attackCooldown;
    private float maxCooldown;

    private float xAttackRange = 3.0f;
    private float yAttackRange = 3.0f;

    public bool move = true;

    private Transform Player;
    private Animator animator;

    public GameObject deathExplosion;
    public GameObject bloodSplattern;

    private int maxHealth;
    public Image enemyHealthBar;

    private string shotDirection;


```

Hier sehen wir den Header unseres Hamsterskripts („TeemoControl“), in dem wir unsere Variablen initialisieren, die wir im Script benutzen.
Die [Header(„X“)] Anweisung dient zur Übersichtlichkeit im Unity Editor.
Public Variablen lassen sich im Unity Editor verändern, private Variablen hingegen sind nur im Skript verwendbar und änderbar.
Die Variablen wurden so benannt, wie die Funktion, die sie im Script erfüllen.

```
37      // Use this for initialization
38      void Start () {
39
40          Player = GameObject.Find("MainCharacter").GetComponent<Transform>();
41          animator = gameObject.GetComponent<Animator>();
42
43          maxHealth = health;
44          maxCooldown = attackCooldown;
45
46          //shotDirection = "right";  // up right down left
47          //currentShotPoint = 1;
48
49      }
50
```

In der vorgenerierten Funktion void Start(), welche beim Starten der Szene einmal ausgeführt wird, binden wir die Position des Spielers
an die Variable Player und weisen der Variable "animator" den Hamster - Animator zu, den wir vor dem Skript erstellt und konfiguriert haben.
Außerdem binden wir die public Variablen "health" und "attackCooldown" an private Variablen ,um feste Werte im Skript weiter zu verwenden.

Vielleicht mag man sich fragen, warum wir dann überhaupt public Variablen verwenden,
wenn die sowieso wieder in private verwandelt werden.
Der Grund dafür ist, dass man so in Unity schnell verschiedene Balancing - Veränderung durchführen kann.

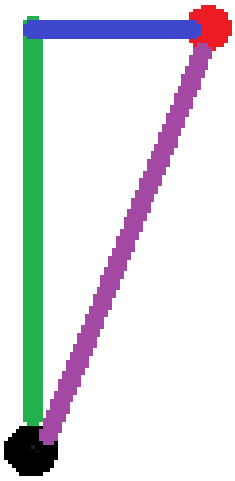
```
51      // Update is called once per frame
52      void Update () {
53
54          //healthbar
55          enemyHealthBar.fillAmount = (float) health / (float) maxHealth;
56
57          //Objekt wird zerstört
58          if(health <= 0)
59          {
60              Destroy(gameObject);
61              Instantiate(deathExplosion, new Vector3(transform.position.x, transform.position.y - 0.5f, transform.position.z), Quaternion.identity);
62          }
63
64          //movement
65          if (move == true)
66          {
67              Movement();
68          }
69
70
71          //reduce the attackcooldown
72          attackCooldown -= Time.deltaTime;
73
74          //shoot
75          if(attackCooldown <= 0 && animator.GetInteger("AttackDir") != 0)
76          {
77
78              shot();
79              resetAttackCooldown();
80          }
81
82      }
```

Die void Update() – Funktion wird auch von Unity generiert und wird nach jedem Frame aufgerufen.
Daraufhin lassen wir unsere HP – Leiste Prozentual zum aktuellen Leben füllen.
Sollte der Wert auf "kleiner gleich" 0 fallen, zerstören wir unser Spielobjekt und lassen ein neues Objekt erstellen.
Dieses neue Objekt ist eine Blutlache, die an der Stelle eines Gegners spawnt, wenn er getötet wurde.
Alle Gameobjekte haben ein eigenes Verhalten, dementsprechend auch ein Skript.
Mit der nächsten If – Abfrage prüfen wir, ob unser boolean "move" auf true gesetzt wurde, welche dann die Funktion Movement() aufruft,
zu welcher wir später noch einmal zurückkommen.
In Zeile 71 reduzieren wir die attackCooldown um die System Zeit, also wenn im Spiel 200ms vergangen sind,
geht der attackCooldown Timer auch wirklich 200ms runter.
In der letzten If – Abfrage überprüfen wir, ob unser Hamster wieder angreifen darf
und er sich in einer Angriffs Animation befindet, also ob die variable AttackDir nicht den Wert 0 hat.
Wenn dies erfüllt ist, rufen wir die Funktion shot() auf und die Funktion resetAttackCooldown(), welche später erläutert werden.

```
84      private void Movement()
85      {
86          {
87              float x = Player.transform.position.x - transform.position.x;  //Spieler Horizontal unterschied
88              float y = Player.transform.position.y - transform.position.y;  //Spieler Vertikal unterschied
89              double doubleX = System.Convert.ToDouble(Math.Abs(x));
90              double doubleY = System.Convert.ToDouble(Math.Abs(y));
91              double rangeSqrt = Math.Pow(doubleX, 2) + Math.Pow(doubleY, 2);
92              double range = Math.Sqrt(rangeSqrt);
93
94
95              if (((range >= xAttackRange - 0.1 && range <= xAttackRange + 0.1)) && ((range >= yAttackRange - 0.1) && (range <= yAttackRange + 0.1)))
96              {
97
98                  transform.position = Vector2.MoveTowards(transform.position, Player.transform.position, 0 * Time.deltaTime); //Bleibt für den Angriff stehen
99                  animator.SetInteger("MovementDir", 0);
100             }
101
102             else if ((range < xAttackRange - 0.1 && range < yAttackRange - 0.1))
103             {
104
105                 Vector2 moveDir = transform.position - Player.transform.position;
106                 transform.Translate(moveDir.normalized * MoveSpeed * Time.deltaTime);   //rennt weg
107                 animator.SetInteger("AttackDir", 0);
108             }
109
110             else
111             {
112                 animator.SetInteger("AttackDir", 0);
113                 transform.position = Vector2.MoveTowards(transform.position, Player.transform.position, MoveSpeed * Time.deltaTime); //bewegt sich zum Spieler
114             }
115
116
```

Auf dem Bild sehen wir die Funktion Movement(), die wir zur Erinnerung immer aufrufen, wenn der boolean "move" auf True gesetz
ist, also wenn sich der Gegner bewegt.
In den ersten beiden Zeilen (87 und 88) berechnen wir jeweils den Unterschied zwischen dem Spieler und unseren Gegner für die X-
und für die Y-Achse.
In den nächsten 4 Zeilen errechnen wir die Hypotenuse zu den zuvor ausgerechneten zwei Strecken, um die direkte Entfernung des
Gegners zu erhalten.

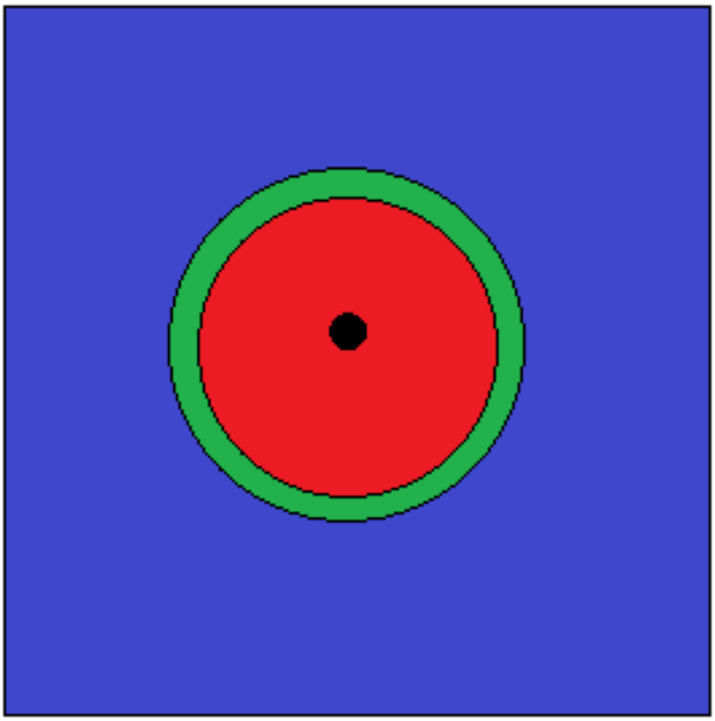
Es folgt ein Schaubild, welche die Theorie dahinter nochmal verbildlicht.



- Spieler
- Gegner
- Entfernung Y-Achse
- Entfernung X-Achse
- Direkte Entfernung

Mit der errechneten Entfernung können wir gucken, ob sich unser Hamster in Angriffsreichweite befindet
oder er vor dem Spieler wegrennen soll oder ihm entgegenrennen bzw. hinterherrennen soll, um in Angriffsreichweite zu kommen.

Die erste If – Abfrage überprüft, ob sich der Spieler in der Angriffszone befindet.
Sollte dies erfüllt sein, sorgt die Anweisung dafür, dass er stehen bleibt und auch keine Bewegungsanimation mehr abspielt.
Die darauffolgende Else if – Abfrage überprüft, ob der Spieler sich zu dicht am Gegner befindet.
Daraufhin sorgt die Anweisung, dafür dass der Gegner vom Spieler wegläuft und keine Angriffsanimation mehr aktiv sein darf.
Falls keine der beiden vorherigen Szenarien zutrifft, rennt der Hamster auf den Spieler zu.



Dazu auch ein Schaubild, das die Theorie dahinter nochmal verbildlicht.

```
117             if (Math.Abs(x) >= Math.Abs(y))
118             {
119                 if (Player.transform.position.x > transform.position.x) //rechts
120                 {
121                     //Debug.Log("rechts");
122                     animator.SetInteger("AttackDir", 0);
123                     animator.SetInteger("MovementDir", 2);
124                     if (((range >= xAttackRange - 0.1 && range <= xAttackRange + 0.1)) && ((range >= yAttackRange - 0.1) && (range <= yAttackRange + 0.1)))
125                     {
126                         //Debug.Log("rechtsAttack");
127                         setMoveFalse();
128                         animator.SetInteger("MovementDir", 0);
129                         animator.SetInteger("AttackDir", 2);
130                         shotDirection = "right";
131                         currentShotPoint = 1;
132                         attackOffset();
133                     }
134                 }
135                 if (Player.transform.position.x < transform.position.x) //links
136                 {
137                     //Debug.Log("links");
138                     animator.SetInteger("AttackDir", 0);
139                     animator.SetInteger("MovementDir", 4);
140                     if (((range >= xAttackRange - 0.1 && range <= xAttackRange + 0.1)) && ((range >= yAttackRange - 0.1) && (range <= yAttackRange + 0.1)))
141                     {
142                         //Debug.Log("linksAttack");
143                         setMoveFalse();
144                         animator.SetInteger("MovementDir", 0);
145                         animator.SetInteger("AttackDir", 4);
146                         shotDirection = "left";
147                         currentShotPoint = 3;
148                         attackOffset();
149                     }
150                 }
151             }
```

Wir befinden uns noch immer in der Movement() Funktion.
Auf diesem Bild geht es um das Verhalten des Animators, den wir relativ am Anfang konfiguriert haben.
Ziel hier ist es, dass unser Hamster, passend zur Position, die richtige Angriffs- und Bewegungsanimation abspielt.

Mit der If - Abfrage (Zeile 117) überprüfen wir erstmal, ob der Abstand zum Spieler auf der X-Achse größer ist als auf der Y-Achse.
Sollte dies der Fall sein wissen wir, dass der Spieler sich mehr rechts oder links vom Hamster befindet, als über oder unter ihm.

Da wir jetzt wissen, dass wir entweder die Bewegungsanimation nach links oder nach rechts abspielen müssen,
müssen wir noch überprüfen, ob der Spieler sich weiter rechts oder weiter links befindet.
Zuerst übernehmen wir den Fall rechts.
Dafür prüfen wir in Zeile 119, ob der X-Wert vom Spieler größer als der X-Wert vom Hamster ist.
Trifft dieses Szenario ein, setzten wir die Variable AttackDir auf 0, um die Angriffsanimation aufjedenfall zu unterdrücken,
und setzten die Variable MovementDir auf 2, was nach unserer anfänglichen Konfiguration der Bewegungsanimation nach rechts entspricht.

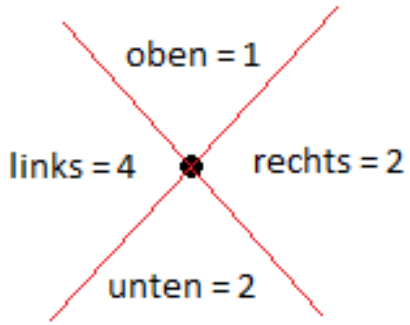
Da wir wissen, dass der Spieler sich rechts von uns befindet, nutzen wir den Zustand aus
und gucken, ob wir uns in Angriffsreichweite befinden mit der if - Abfrage in Zeile 124.
Sollten wir uns in besagter Angriffsreichweite befinden, rufen wir die Funktion setMoveFalse() auf, worauf wir später nochmal zurückkommen,
kurz gesagt setzt sie unser boolean "move" auf False, setzten die Variable MovementDir auf 0 und die Variable AttackDir auf 2,
was nach unsere anfänglichen Konfiguration Angriff nach rechts entspricht.
Zusätzlich weisen wir dem String shotDirection die Zeichenkette „right“ zu und setzten den currentShotPoint auf 1. Letztendlich rufen wir die Funktion attackOffset() auf.

Die Befehle in Zeile 130 bis 132 werden später erläutert.

Falls sich der Spieler Links befindet wird die If - Abfrage in Zeile 135 ausgelöst,
welche dasselbe wie die If -Abfrage in Zeile 119 macht, nur sind alle Anweisung angepasst, um die Animation nach Links zu steuern.

```
152             else
153             {
154                 if (Player.transform.position.y > transform.position.y) //oben
155                 {
156                     //Debug.Log("oben");
157                     animator.SetInteger("AttackDir", 0);
158                     animator.SetInteger("MovementDir", 1);
159                     if (((range >= xAttackRange - 0.1 && range <= xAttackRange + 0.1)) && ((range >= yAttackRange - 0.1) && (range <= yAttackRange + 0.1)))
160                     {
161                         //Debug.Log("obenAttack");
162                         setMoveFalse();
163                         animator.SetInteger("MovementDir", 0);
164                         animator.SetInteger("AttackDir", 1);
165                         shotDirection = "up";
166                         currentShotPoint = 0;
167                         attackOffset();
168                     }
169                 }
170                 if (Player.transform.position.y < transform.position.y) //unten
171                 {
172                     //Debug.Log("unten");
173                     animator.SetInteger("AttackDir", 0);
174                     animator.SetInteger("MovementDir", 3);
175                     if (((range >= xAttackRange - 0.1 && range <= xAttackRange + 0.1)) && ((range >= yAttackRange - 0.1) && (range <= yAttackRange + 0.1)))
176                     {
177                         //Debug.Log("untenAttack");
178                         setMoveFalse();
179                         animator.SetInteger("MovementDir", 0);
180                         animator.SetInteger("AttackDir", 3);
181                         shotDirection = "down";
182                         currentShotPoint = 2;
183                         attackOffset();
184                     }
185                 }
186             }
187         }
188     }
189
```

Der Vollständigkeitshalber der letzte Teil der Movement() Funktion.
Sollte die If – Abfrage, in der wir testen, ob der horizontale Abstand zum Spieler größer ist als der vertikale Abstand, fehlschlagen, wird die Else – Abfrage in Zeile 152 ausgelöst.
Dementsprechend testen die nachfolgenden If – Abfragen, ob der Spieler sich über oder unter unseren Hamster befindet und führt dementsprechend dieselben Anweisungen aus, nur auf die richtige Richtung getrimmt.



Um das Thema abzuschließen, noch ein Schaubild, um das Ganze noch einmal zu verbildlichen.

```
190     private void resetAttackCooldown()
191     {
192         attackCooldown = maxCooldown;
193     }
194
195     private void attackOffset()
196     {
197         attackCooldown = +0.7f;
198     }
199
200     private void shot()
201     {
202       GameObject bullet = (GameObject) Instantiate(teemoAttack,shotPoints[currentShotPoint].transform.position,Quaternion.identity);
203        bullet.GetComponent<TeemoBulletControl>().setTargetDirection(shotDirection);
204     }
205
206
207
```

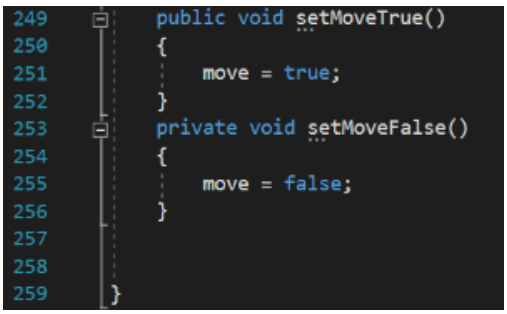
In Zeile 190 definieren wir die Funktion resetAttackCooldown(),
welche lediglich dafür sorgt, dass bei jedem Aufruf die derzeitige attackCooldown wieder auf den definierten Wert von maxCooldow gesetzt wird.
Die nächste Funktion, die wir definieren, attackOffset() sorgt dafür, das auf attackCooldown 0.7f addiert wird.
Der Grund dafür ist, dass der Hamster den Pfeil erst schießen soll, wenn bei der Animation der richtige Frame abgespielt wird, daher müssen wir die künstliche Verzögerung einbauen.
Die Letzte Funktion auf dem Bild (Zeile 200) shot() sorgt dafür, dass, wenn die Funktion aufgerufen wird ,
unser Hamster ein GameObject erzeugt, welches sein Pfeil ist, und ihm mitteilt, wo er erzeugt wird und in welche Richtung er fliege soll.
Aus diesen Grund setzten wir in der Movement() Funktion den currentShotPoint und die shotDirection.
Diese Information Teilen wir dem Skript mit, der mit dem Pfeil vom Hamster verbunden ist und der regelt das Verhalten vom Pfeil.

```
208     private void OnTriggerEnter2D(Collider2D collision)
209     {
210         //Instantiate(bloodSplattern, new Vector3(transform.position.x, transform.position.y, transform.position.z), Quaternion.identity);
211
212         if (collision.tag == "BulletMG")
213         {
214             health -= 10;
215             Destroy(collision.gameObject);
216             Instantiate(bloodSplattern, new Vector3(transform.position.x-0.5f, transform.position.y-0.5f, transform.position.z), Quaternion.identity);
217         }
218
219         if (collision.tag == "BulletPistol")
220         {
221             health -= 10;
222             Destroy(collision.gameObject);
223             Instantiate(bloodSplattern, new Vector3(transform.position.x-0.5f, transform.position.y-0.5f, transform.position.z), Quaternion.identity);
224         }
225
226         if (collision.tag == "BulletRocket")
227         {
228             health -= 50;
229             Destroy(collision.gameObject);
230             Instantiate(bloodSplattern, new Vector3(transform.position.x - 0.5f, transform.position.y - 0.5f, transform.position.z), Quaternion.identity);
231         }
232
233         if (collision.tag == "BulletShotgun")
234         {
235             health -= 5;
236             Destroy(collision.gameObject);
237             Instantiate(bloodSplattern, new Vector3(transform.position.x - 0.5f, transform.position.y - 0.5f, transform.position.z), Quaternion.identity);
238         }
239
240         if (collision.tag == "BulletArrow")
241         {
242             health -= 5;
243             Destroy(collision.gameObject);
244             Instantiate(bloodSplattern, new Vector3(transform.position.x - 0.5f, transform.position.y - 0.5f, transform.position.z), Quaternion.identity);
245         }
246
247     }
248
```

Auf diesen Bild sehen wir die Funktion OnTriggerEnter2D(Collider2D collision).
Diese macht letztendlich nichts anderes, als die Information von Unity zu bekommen,
was für eine Berührung mit der Hamster Hitbox, die wir in Unity definiert haben, stattgefunden hat.

Dafür überprüfen wir den Tag des Objektes, welches uns berührt hat und führen dementsprechend eine Anweisung aus.

Als Beispiel gehen wir davon aus, dass den Hamster eine Maschinengewehrkugel, mit dem Tag „BulletMG“, getroffen hat.
Daraufhin wird von der health Variable der Wert 10 abgezogen. Die Kugel, die uns getroffen hat, wird zerstört
und wir initialisieren das Objekt bloodSplattern an der derzeitigen Position des Hamsters.
Das bloodSplattern sind die Blutspritzer, die einen Treffer anzeigen, welche auch wieder von einem eigenen Skript gesteuert werden



Im letzten Teil des Skripts definieren wir die Funktion setMoveTrue()
und die Funktion setMoveFalse() welche jeweils den boolean "move" auf true oder false setzten.

Letztendlich haben wir dann mit unseren vorhandenen Gegnern, Weltobjekten
und unserem Spieler den am Anfang besprochenen Prototypen unseres Raums entwickelt.

Ziel für den nächsten Milestone:
In diesem Milestone planen wir unseren nächsten Raumtyp, den Fallenraum, umzusetzen. Zudem soll unser bisheriger Raum mehr Vielfalt an Gegner bekommen.

Schlusswort:

Was wir auch niemanden vorenthalten wollen, ist die Wichtigkeit des stetigen Testens von dem, was man macht.
Als optimal hat sich erwiesen, wenn andere Leute die Sachen testen, die man selbst erstellt hat.
So konnten immer mehr Fehler gefunden werden, als der Ersteller selbst entdecken konnte.

Hier nur ein kleiner Ausschnitt unserer Bug Liste die wir hatten, als es nur einen Gegner gab und den Spieler.

BUG |Braun Bär dmg am Player

3

A SH

BUG| Kugeln in Unity

1

A

BUG| Pistole nach unten

1

A

BUG| HP Leiste Gegner

2

A

BUG| Image beim Waffenwechsel

1

A

BUG| Character animation beim schießen in 2 richtungen

1

A