HOCHSCHULE TRIER
Trier University of Applied Sciences
Informatik - Computer Science

Entwicklung eines 2D Plattformers

Dokumentationn

Fabio Gimmillaro, Daniel Schreiber, Tobias Rühl und Joscha Wülk

Medienprojekt

Betreuer: Prof. Dr. Christof Rezk-Salama

Trier, Abgabedatum

Inhaltsverzeichnis

1

Konzeptionierung

Dieses Spiel ist das erste größere Projekt mit Unity für jeden von uns. Somit wollten wir uns zunächst an einem einfacherem Spielprinzip orientieren. Das Spiel ist ein 2D Plattformer, welcher sich durch einen schnellen Ablauf, beispielsweise in den Kämpfen auszeichnet. Hauptsächlich geht es um das Bezwingen von Gegnern in verschiedenen Levels gehen. Die Level sollen hierbei ebenfalls spannende Jump and Run Einlagen bieten. Inpsirationen für dieses Spiel waren in der Grundidee “Risk of Rain“, mit seinen abwechslungsreichen und schnelleren Kämpfen.

2

Die Kamera

3

GameObject “Player“ & Komponenten

Die Spielfigur, die der Anwender bedient stellte sich während der Entwicklung als komplexestes GameObject heraus. In diesem Kapitel geben wir daher einen Überblick über die Realisierung mithilfe von Sprites sowie der verschiedenen anhängenden und selbsterstellten Skripte

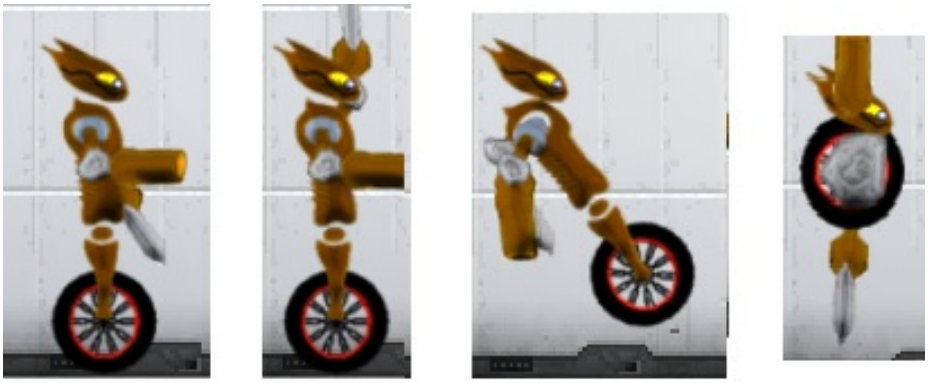

Abb. 3.1. Zusammenschnitt der Player Sprites

3.1 SpriteRenderer

3.2 AttributeComponent Skript

Die AttributeComponent dient dazu die spielmechanischen Daten zu hinterlegen und zu verwalten. Sie kann sowohl dem Spieler als auch freundlich und feindlich Gesinnten NPCs angehängt werden.

Nachfolgend werden die wichtigsten Daten vorgestellt:

```
1  //Lebenspunkte, maximale Lebenspunkte, Ruestung und Schaden
2  public float health, maxHealth, armor, damage;
3  //Maximale Ausdauer, Ausdauer und regenerierte Ausdauer pro Sekunde
4  public float maxStamina, stamina, staminaPerSecond;
5  //Munition, Munitionskapazitaet und Reichweite
6  //Range = 0 -> Nahkampf / Range > 0 -> Fernkampf
7  public int ammo, ammoCap, range;
```

```
 8  //Klon aktiv?
 9  public bool cloneAlive = false;
10
11  //Cooldown fuer Plasmaschuss
12  static float cooldown1 = 1.0f;
13  //Laeuft cooldown fuer Plasmaschuss?
14  bool cooldown1Active = false;
15
16  //Cooldown fuer Klonfaehigkeit
17  static float cooldown2 = 10.0f;
18  //Time-to-live fuer Klon
19  static float ttl = 5.0f;
20  //Aktuelle Time-To-Live
21  float attl = ttl;
22  //Laeuft cooldown fuer Klon?
23  bool cooldown2Active = false;
24
25  //Referenz auf das Nahkampfsystem
26  MeleeSystem meleeSys;
```

Die Update-Funktion der AttributeComponent wird ausschließlich dazu benutzt, die Stamina über Zeit aufzufüllen während die fixedUpdate-Funktion für das löschen des Klones verantwortlich ist.

```
 1  void Update () {
 2
 3  /*Fuelle Ausdauer ueber Zeit wieder auf solange maximalStamina nicht
        erreicht ist und die
 4  Spielfigur sich nicht bewegt*/
 5  if (staminaPerSecond > 0.0f && stamina < maxStamina && !meleeSys.
        animationRunning)
 6  {
 7  //Stelle sicher, dass Stamina nicht kleiner als 0 oder groesser als
        maximalStamina gesetzt wird
 8  stamina = Mathf.Clamp(stamina+staminaPerSecond * Time.deltaTime,0,maxStamina
        );
 9  }
10  }
11  void FixedUpdate()
12  {
13  if (attl > 0)
14  attl -= Time.deltaTime;
15  if (attl <= 0 && cloneAlive)
16  {
17  Destroy(GameObject.Find("Klon"));
18  cloneAlive = false;
19  }
20  }
```

Getter- und Settermethoden für die jeweiligen Variablen sind ebenfalls in der AttributeComponent enthalten. Desweiteren wurde eine Methode für das reduzieren der Ausdauer geschrieben.

```
 1  //Returns difference between possible Stamina Damage and really done stamina
        dmg
 2  //eg. all damage can be absorbed into stamina = returns 0
 3  //10 Stamina left but 20  damage returns 10
 4  public float reduceStamina(float amount)
 5  {
 6  float potentialDamage = stamina - amount;
 7  stamina = Mathf.Clamp(stamina - amount, 0, maxStamina);
 8  if (potentialDamage < 0)
 9  return -1 * potentialDamage;
10  else
11  return 0;
12  }
```

3.3 Projectile Pooling System

Angelehnt aus dem Pooling Verfahren aus der Spieleprogrammierung Vorlesung wollten wir ein ähnliches System für unsere Projektile umsetzen, um häufiges Erzeugen und Löschen von GameObjects zu vermeiden.

Hier besitzt jedes Spielobjekt, das in der Lage ist Projektile zu verschießen zwischen 4 und 8 Projektile die beim Intialisieren des GameObjects erstellt werden. Diese sind jedoch zunächst nicht aktiv.

Soll ein Schuss abgefeuert werden, wird aus einem Array ein Projektil genommen, zum benutzen aktiviert und freigegeben. Danach wird sich die Position des nächsten freien Projektils entsprechend angepasst.

```
1  public GameObject getProjectile()
2  {
3      if (pointer >= 0) {
4          BoxCollider2D temp = (BoxCollider2D)projectiles [pointer].GetComponent
               (typeof(BoxCollider2D));
5          SpriteRenderer tempS = (SpriteRenderer)projectiles [pointer].
               GetComponent (typeof(SpriteRenderer));
6          temp.enabled = true;
7          tempS.enabled = true;
8          return projectiles [pointer--];
9      }
10     return null;
11 }
```

Trifft ein Projektil nun auf, oder hat die maximale Distanz zurück gelegt, wird der Schaden ausgeführt, deaktiviert und anschließend der Zeiger auf das nächste mögliche Projektil angepasst.

```
1  //Dient zum lagern von Getroffenen/Abgeprallten Projektilen
2  public void storeProjectile(GameObject projectile)
3  {
4      if (pointer < projectileAmount) {
5          BoxCollider2D temp = (BoxCollider2D)projectile.GetComponent (typeof(
               BoxCollider2D));
6          SpriteRenderer tempS = (SpriteRenderer)projectile.GetComponent (typeof
               (SpriteRenderer));
7          temp.enabled = false;
8          tempS.enabled = false;
9          Rigidbody2D prorigid = (Rigidbody2D)projectile.GetComponent (typeof(
               Rigidbody2D));
10         prorigid.velocity = new Vector2 (0, 0);
11         projectiles[++pointer] = projectile;
12     }
13 }
```

3.4 InputSystem Skript

Beim InputSystem des Spielers wurde darauf geachtet das wirklich nur der Input abgefragt und nicht die AusfÃ¼hrung der jeweiligen Aktion mit implementiert wird. Alle Aktionen des Spielers werden Ã¼ber die Tastatur gesteuert.

Hier eine Auflistung der Tasten und ihrer Aktionen:

- W = springen
- K = Klon erzeugen

- S = schießen
- B = blocken
- C = Waffe wechseln
- J = schlagen
- A/D = links/rechts laufen

```
1  [...]
2  void Update()
3  {
4      float movePlayerVector = Input.GetAxis("Horizontal");
5      bool isFacingRight;
6      if (movePlayerVector >= 0)
7          isFacingRight = true;
8      else
9          isFacingRight = false;
10
11
12     //Funktion fÃ¼r Springen
13     if (Input.GetKey("w"))
14     {
15         if (!meleeSys.blocking && !movement.unableToMove)
16             StartCoroutine(movement.jump());
17     }
18     //Funktion fÃ¼r Schiessen
19     if (Input.GetKeyDown("s"))
20     {
21         StartCoroutine(rangedSys.shoot(primaryShot));
22     }
23
24     if (Input.GetKeyDown("b"))
25     {
26         meleeSys.block();
27     }
28
29     if (Input.GetKeyUp("b"))
30     {
31         meleeSys.unblock();
32     }
33
34     if (Input.GetKeyDown("c"))
35     {
36         rangedSys.switchWeapon();
37         primaryShot = !primaryShot;
38     }
39
40     if(Input.GetKeyDown("k"))
41     {
42         if (!attComp.getCooldown2Active())
43         {
44             abilitySys.clone();
45         }
46     }
47
48     if (Input.GetKeyDown ("j"))
49     {
50         if(movement.grounded && meleeSys.anim != null && !meleeSys.anim.
               GetBool("MeleeAttackInQueue"))
51         {
52             movePlayerVector = 0.0f;
53             meleeSys.punch();
54         }
55     }
56
57
58
```

```
59      if (meleeSys.animationRunning || meleeSys.blocking || movement.
            unableToMove)
60          movePlayerVector = 0.0f;
61      movement.move(movePlayerVector);
62  }
```

3.5 CharacterMovement Skript

3.6 MeleeSystem Skript

Der Nahkampf der Spielers ist wie folgt konzeptioniert: Drückt ein Spieler die entsprechende Taste, wird ein Schlag ausgeführt. Drückt der Spieler während dieser Schlag durchgeführt wird erneut die Schlagtaste, wird eine Folgeschlag eingereiht, welche unmittelbar danach begonnen wird. Somit wird eine Art Nahkampf-Kombo-System umgesetzt, welches bis zu vier Schläge ausführen kann. Danach wird wieder mit dem ersten Schlag begonnen. Erleidet der Spieler während des Kombos einen Treffer, so soll die Kombo abbrechen, und er muss erneut von vorne beginnen.

In diesem Skript arbeiten die Animator Component von Unity eng mit dem MeleeSystem Skript zusammen. Wird der Schlag betätigt, wird ein Trigger im Animator ausgelöst, der die entsprechende Animation aufruft. In dieser Animation wird an einem enstprechenden Keyframe der Schaden durchgeführt, sodass es passend zur Animation stattfindet. Während die Animation läuft, ist ein bool'scher Wert so gesetzt, dass neue Schlagaktionen als Kombo interpretiert werden. Ist das der Fall, wechselt der Animator am Ende einer Schlaganimation in die entsprechend nächste. Erfolgt kein Tastendruck zum Schlagen, wird der Spieler getroffen, oder ist die letzte Schlaganimation erreicht, so wird per Animator zurück in den Idle State gewechselt. Dann werden neue Schlagaktionen als Beginn einer neuen Kombo aufgefasst, und die erste Schlaganimation wird durchgeführt.

Um das ganze spielerisch attraktiv zu machen, ist die letzte Schlaganimation so gewählt, dass hier an mehreren verschiedenen Keyframes Schaden ausgeteilt wird.

3.7 HealthSystem Skript

3.8 RangedSystem Skript

Ähnlich wie bei dem MeleeSystem ist hier ein Zusammenspiel zwischen Animator und Skript ausschlaggebend, jedoch ist hier der Auslöser entweder durch Spielerinput oder im Falle von Gegnern durch das entsprechende KI Skript.

Erfolgt der Befehl zum Schießen, wechselt der Animator in die Animation zum schießen. Das Skript wartet nun auf den Animator. Am entsprechenden Keyframe läuft die Schussfunktion weiter und es wird je nach gewählter Munition beziehungsweise Schussart ein Projektil aus dem Pooling System geholt, aktiviert und

mit entsprechenden Werten wie Schaden und Sprite befüllt und anschließend in Blickrichtung abgeschossen.

3.9 AbilitySystem Skript

Das AbilitySystem wurde geschrieben um verschiedene FÃ¤higkeiten, die dem Spieler zur Verfügung stehen, zu implementieren. In der aktuell vorliegenden Version wurde bisher nur eine Methode für die Fähigkeit "Klonen"geschrieben.

```
1  //FÃ¤higkeit um Klon zu erzeugen
2  public void clone()
3  {
4  //Setze cooldown auf aktiv
5  attComp.setCooldown2Active(true);
6
7  //Erstelle Klon aus Prefab
8  GameObject clone = (GameObject)Instantiate(illusion);
9  clone.name = "Klon";
10 //Setze Klon auf aktuelle Spielerposition
11 clone.transform.position = transform.position;
12 //Deaktiviere Kollision zwischen Spieler und Klon
13 Physics2D.IgnoreCollision(GameObject.Find("Player").GetComponent<
       BoxCollider2D>(), clone.GetComponent<BoxCollider2D>());
14 //Setze Time-To-Live fuer Klon
15 attComp.setTTL();
16 }
```

4

Enemies

In unserem Spiel waren zwei sich ähnliche Gegner angedacht, wovon einer als Fernkämpfer und der andere als Nahkämpfer fungierte. Auf Grund mangelnder Animationen hat lediglich der Fernkämpfer seinen Weg ins Spiel gefunden. Von der Implementierung hätten sich die beiden Gegnertypen aber kaum unterschieden, da die KI möglichst modular und austauschbar gehalten werden sollte. Somit unterscheiden sich die zwei Typen lediglich durch eine andere Reichweite für den Angriff, und eine andere Angriffsfunktion, die aufgerufen wird.

Die Gegner wurde eine simple Entscheidungskaskade in Form von verschiedenen If-Anweisungen gegeben, nach denen die Gegner dann ihr Verhalten auswählen. Das Verhalten der Gegner ist stark davon abhängig ob Sichtkontakt zum Spieler besteht, was durch Raycasts zwischen dem Gegner und dem Spieler überprüft wird. Hierbei kann auch ein Blickwinkel definiert werden, in welchem der Spieler wahrgenommen wird.

```
1  [...]
2  //Blickrichtung entweder x=1 oder -1
3  if (actions.facingRight)
4      viewVector += new Vector2(1, 0);
5  else
6      viewVector += new Vector2(-1, 0);
7
8  //Winkel zwischen Blickrichtung und Verbindungsvektor
9  float currentAngle = Vector2.Angle(viewVector, difference);
10 RaycastHit2D hit = Physics2D.Raycast(visionCheck.position,viewVector,
       noticeDistance);
11
12 //Ueberpruefen ob der Winkel maximal dem angegebenen Field of View Winkel
       ist
13 if (currentAngle <= fovAngle)
14 {
15     //Raycast der von Augen zum Ziel geschossen wird, wenn das erste
           getroffene Objekt der Spieler ist, ist die sicht nicht blockiert
16     if (hit.collider != null)
17         playerVisible = hit.collider.gameObject == rigplayer.gameObject;
18    else
19       playerVisible = false;
20 }
21 else
22    playerVisible = false;
```

Mit Hilfe dieses Sichtbarkeitschecks, kann die KI verschiedene Aktionen abhängig von verschiedenen Bedingungen wählen. Diese Entscheidungskaskade könnte man als Behaviour Tree folgendermaßen visualisieren:

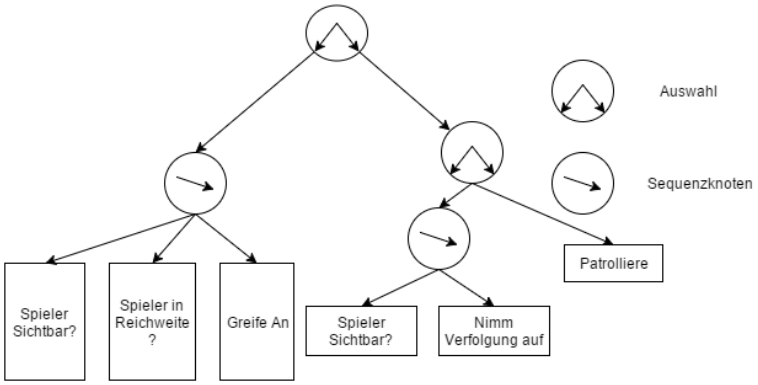

Abb. 4.1. Verhalten des Gegners als Behaviour Tree

Implementiert wurde der Entscheidungsalgorithmus jedoch nur mit if-Bedingungen und nicht mit einem Behaviour Tree oder einer State Machine.

```
1  //Wenn in Angriffsreichweite, und der Spieler sichtbar ist, bleibe stehen
       beginne anzugreifen
2  //Ruft die Angriffsmethode des RangedSystems auf
3  if ((inAttackRangex && inAttackRangey) && vision.playerVisible)
4  {
5     movement.move(0.0f);
6     anim.SetBool("AttackInProgress", true);
7     StartCoroutine(rangeSys.shoot(true));
8  }
9
10 //Wenn sich Spieler nicht in Reichweite zum Angreifen befindet, aber
       Sichtkontakt besteht, nimm Verfolgung auf
11 //Ruft die Bewegungsmethode des Movementssystem auf.
12 if (!inAttackRangex && inAttackRangey && vision.playerVisible)
13 {
14    if (movement.grounded)
15    {
16       if (walkingRight)
17          movement.move(1.0f);
18       else
19          movement.move(-1.0f);
20    }
21    else
22       movement.move(0.0f);
23 }
24
25
26 //Wenn das Ziel nicht sichtbar ist, soll patrolliert werden
27 if (!vision.playerVisible)
28 {
29    //Wenn wir auf dem Boden befinden, beweg dich, ansonsten bleib stehen
30    if (movement.grounded)
31    {
32       //Befinden wir uns vor einer Wand oder einem Abgrund, drehe um.
33       if (hittingWall || onAnEdge)
34          walkingRight = !walkingRight;
35       if (walkingRight)
36       {
37          movement.move(1.0f);
```

```
38            }
39            else
40            {
41                movement.move(-1.0f);
42            }
43        }
44        else
45            movement.move(0.0f);
46    }
```

5

Endboss

Der Kampf mit dem Endboss, welcher auf der obersten Plattform des Levels zu finden ist, wurde speziell für den Bosskampf gescriptet. Er erscheint über eine Colliderbox, die beim betreten den Spawn des Bosses startet. Nach Ablauf eines kurzen Timers beginnt dieser zu erscheinen.

Im Kampf selbst wurden ähnliche Logikabfragen wie bei der normalen Gegner KI implementiert, jedoch in einer spezielleren Form, da sich durch gewünschte Bossfähigkeiten diese nicht mit dem normalen KI Script umsetzen lassen.

Der Boss hat 3 Angriffe die von bestimmte Bedingungen haben:

Schlag mit linker Hand Wenn sich der Spieler in der Trefferzone der linken Hand befindet

Schlag mit rechter Hand Wenn sich der Spieler in der Trefferzone der rechten befindet

Schreien Keine Bedingung. Lässt den Spieler für eine gewisse Zeit bewegungsunfähig werden.

Damit diese drei Angriffe nicht etwa in statischer Reihenfolge ausgeführt werden, und etwas mehr Dynamik in den Kampfablauf bringen, wurde hier ein Zufallsalgorithmus implementiert. Dieser entscheidet zufällig welcher der möglichen Angriffe gewählt werden soll, unter Berücksichtigung der letzten ausgeführten Aktionen. Eine der möglichen Aktionen wird proportional zu ihren vergangenen Aktionsphasen, seit der letzten Verwendung immer wachsen. Beim ersten Angriff besteht also für jede Funktion eine Chance von $\frac{1}{3}$. Wird dann beispielsweise der Schlag mit der linken Hand ausgeführt, ist die Chance für nochmals diese Aktion $\frac{1}{5}$, für den Schrei und den Schlag mit der rechten Hand jeweils $\frac{2}{5}$.

Besonderheit hierbei ist, dass die Schläge zweimal hintereinander ausgeführt werden, der Zähler wird lediglich auf 1 zurückgesetzt, während es ausgeschlossen ist, dass zweimal ein Schrei ausgeführt wird.

Beispielsweise der Code für eine der gewählten Aktionen und für die Berechnung der Wahrscheinlichkeiten.

```
1  void RightHandHit()
2  {
3      //Variablen fuer den Code, welche attacke benutzt werden soll
4      timesSinceScream++;
5      timesSinceLeftHand++;
6      timesSinceRightHand = 1;
7
8      //Variablen fÃ¼r die Animation
9      idleStateExecuted = false;
10     anim.SetTrigger("RightHandAttack");
11 }
12
13
14 void getProbabilities()
15 {
16     int options = timesSinceScream;
17     if (inLeftHitBox)
18     {
19         options += timesSinceLeftHand;
20     }
21     if(inRightHitBox)
22     {
23         options += timesSinceRightHand;
24     }
25     screamProb = (float)timesSinceScream / (float)options;
26
27     leftHandProb = inLeftHitBox ? (float)timesSinceLeftHand / (float)options
           : 0.0f;
28     rightHandProb = inRightHitBox ? (float)timesSinceRightHand / (float)
           options : 0.0f;
29 }
```

Die Entscheidungen des Bosskampfes in einem Behaviour Tree dargestellt könnte folgendermaßen aussehen:

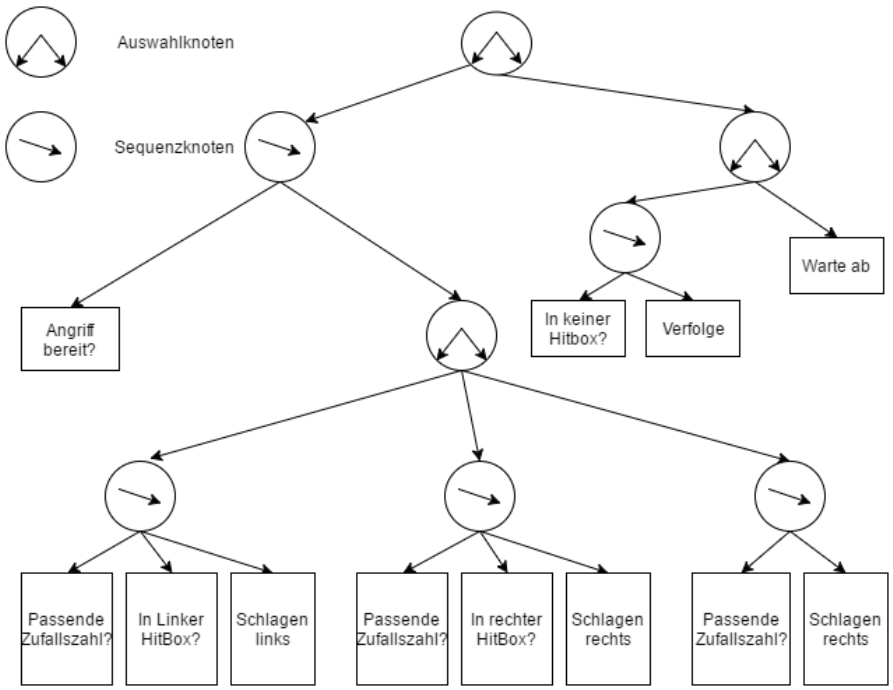

Abb. 5.1. Verhalten des Endboss' als Behaviour Tree

6

Grafiken und Animationen

7

Sounds

8

Spielobjekte

8.1 Truhe

Eine simple Truhe die mit der Taste “E“-Taste geöffnet werden kann, solange man sich im Collider befindet. Hierbei wird aus einer Liste an GameObjects zu”fallig ausgewählt und aus der Kiste gedroppt.

8.2 Pickups

Es gibt verschiedene Pickups, wie etwa Health Potions die Leben regenerieren, oder Munition für den Spezialschuss. Diese sind durch den Collider aufsammelbar, und schweben an der Stelle wo sie liegen leicht auf und ab. Dies wurde mit Hilfe einer Smoothstep Funktion umgesetzt, die in einer glatten Kurve zwischen zwei Extrempunkten interpoliert.

```
1  //Smoothstep Funktion die einen Wert der zwischen 0 und 1 oszilliert "
       glaettet" um eine gleichmaessige Kurve zu erzeugen
2  void smoothStep(float parameter)
3  {
4      xParameter =0.5f* (1.0f - Mathf.Cos(Mathf.PI * parameter * xFrequency));
5      yParameter =0.5f* (1.0f - Mathf.Cos(Mathf.PI * parameter * yFrequency));
6  }
7
8  //Letztendliche Berechnung der Position
9  void Interpolate()
10 {
11     float currentX = minX + xParameter * (maxX - minX);
12     float currentY = minY + yParameter * (maxY - minY);
13
14     this.transform.position = new Vector2(currentX, currentY);
15 }
```

9

Events

10

Demolevel

Zu Anfang des Projekts war die Gestaltung und der Aufbau des Levels durch Sprites gedacht, die den Hintergrund darstellen sollten. Durch Probleme mit Skalierungen und wiederholenden Hintergründen waren wir gezwungen das Level unlogisch aufzubauen. So mussten wir zum Beispiel viele Kisten stapeln um eine weitere Ebene zu erreichen, was nicht in die Szene gepasst hätte. Deshalb haben wir uns entschieden das Level durch Tiles zu gestalten. Zu Erstellung nutzten wir das Programm "Tiledßowie "Tiled2Unityßum importieren in Unity. Dabei wurden freie Tilesets genutzt.

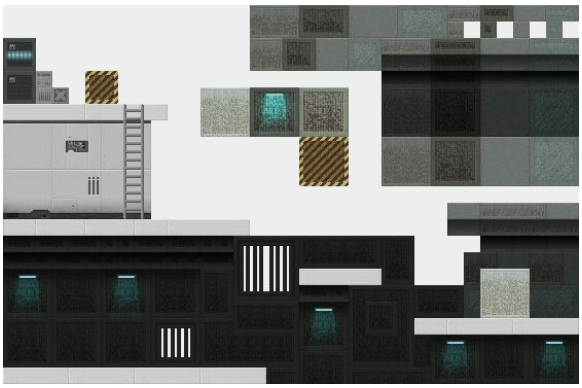

Abb. 10.1. Für das Level genutztes Tileset

Das Level sollte zu Beginn schon einem Turm ähneln, um an der Spitze den Kampf gegen den Endgegner einzufügen. Ziel beim bauen des Levels war es den Spieler dazu zu bringen alle seine Fähigkeiten zu nutzen um, möglichst unbeschadet, ans Ende des Levels zu kommen und den Endgegner zu besiegen. Um den Spieler an die Steuerung zu gewöhnen wird er schon früh mehrere Sprünge vollführen müssen um in den nächsten Abschnitt zu gelangen. Damit der Spieler nicht ständig von einer Einer ebene zur nächsten Springen muss wurden die "Luftströmeëingeführt, die ihn automatisch nach Oben tragen. Um diese darzustellen wurde ein Partikeleffekt genutzt.

```
1     [...]
2     //befindet sich der Spieler innerhalb des Colliders des Luftstrom, so
          wird ihm in Richtung der
```

```
3     // y-Achse geschwindigkeit hinzgefuegt
4     void OnTriggerStay2D(Collider2D other)
5     {
6        if(other.tag == "Player")
7           other.attachedRigidbody.velocity = new Vector2 (other.
                attachedRigidbody.velocity.x, other.attachedRigidbody.velocity.y
                +0.3f);
8     }
```

Weiterhin wurden Türen und Aufzüge genutzt um den Spieler große Abschnitte weiter nach oben Steigen oder ihn ins innere des Turms wechseln zu lassen. Dafür wurde ein Script genutzt das aktiviert wird sobald der Spieler innerhalb des Colliders ë"drückt. Dadurch färbt sich der Bildschirm schwarz und der Spieler sowie die Kamera wird zur Ausgangstür bewegt. Danach klart der Bildschirm wieder auf.

```
1     [...]
2     void OnTriggerStay2D(Collider2D other)
3     {
4        if (other.tag == "Player" && Input.GetKeyDown ("e") && !isFading) {
5           isFading = true;
6           fadeToBlack = true;
7        }
8     }
9
10    //Zum aufklaren des Bildschirms
11    //Lerpt zwischen der momentanen Farbe der Textur und "Klar" bis der
          Alphawert der Textur kleiner als 0.1 ist.
12    //Danach wird die Farbe auf "Klar" gesetzt.
13    void FadeToClear()
14    {
15       fadeTexture.color = Color.Lerp (fadeTexture.color, Color.clear, 1.5f *
             Time.deltaTime);
16       if(fadeTexture.color.a <= 0.1f)
17       {
18          fadeTexture.color = Color.clear;
19          fadeTexture.enabled = false;
20          isFading = false;
21       }
22    }
23
24    //Um den Bildschirm schwarz zu faerben
25    //Lerpt zwischen der momentanen Farbe der Textur und "Schwarz" bis der
          Alphawert der Textur groesser als 0.8 ist.
26    //Danach wird die Farbe auf "Schwarz" gesetzt.
27    void FadeToBlack()
28    {
29       fadeTexture.enabled = true;
30       fadeTexture.color = Color.Lerp (fadeTexture.color, Color.black, 1.5f *
             Time.deltaTime);
31       if (fadeTexture.color.a >= 0.8f) {
32          fadeTexture.color = Color.black;
33          fadeToBlack = false;
34          playerTransform.position = target.position;
35          cam.position = new Vector3 (target.position.x, target.position.y,
                cam.position.z);
36       }
37    }
```

11

Menüführung

Für das Hauptmenü wurde eine eigene Scene erstellt. Es besteht aus den Buttons “Play“ und “Exit“ sowie einem Hintergrund. Für den Text wurde eine lizenfreie Schriftart aus dem Internet verwendet. Fährt man mit der Maus über einen der Buttons, faded die Textfarbe von Gelb zu Rot.

Für das Handling der Aktionen wurde ein Menüführungsskript geschrieben das im folgenden vorgestellt wird.

```
1  void Start () {
2  //Canvas fuer Dialogfenster
3  quitDialogue = quitDialogue.GetComponent<Canvas>();
4  //Play-Button
5  play =  GameObject.Find("Play").GetComponent<Button>();
6  //Exit-Button
7  exit = GameObject.Find("Exit").GetComponent<Button>();
8  quitDialogue.enabled = false;
9  }
```

Je nach ausgeführter Aktion werden einzelne Komponenten der Scene deaktiviert beziehungsweise aktiviert. Als Beispiel folgt die Methode, die aufgerufen wird, wenn der Spieler den “Exit“-Button anklickt:

```
1  public void ExitPress()
2  {
3  //Canvas fuer das Dialogfenster wird auf active gesetzt
4  //Buttons ausserhalb des Dialogfensters werden deaktiviert
5  quitDialogue.enabled = true;
6  play.enabled = false;
7  exit.enabled = false;
8  }
```

12

Head-Up-Display & Tooltips

13

Zielsetzung

14

Abschlussanalyse