

## Civil War Nation

Entwicklung eines rundenbasierten Strategiespiels / Development of a turnbased strategy game

Daniel Schreiber 957819,  
Tobias Meyering 954574,  
Joscha Wülk 958193,  
Benjamin Rätze 954436,  
Tobias Rühl 957798,  
Alexander Hanf 954155

Interdisziplinäres Teamprojekt / Interdisciplinary team project  
Betreuer: Prof. Dr. Linda Breitlauch, Prof. Dr. Christof Rezk-Salama

Trier, den 30.04.2016

---

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	1
<b>2</b>	<b>Squad Builder</b>	2
<b>3</b>	<b>Levelaufbau</b>	3
<b>4</b>	<b>Manager Objekt</b>	4
4.1	Manager System	4
4.2	Shooting System	4
4.3	Inventory System	5
4.4	Player Assistance System	5
4.5	Ability System	5
4.6	Health System	5
<b>5</b>	<b>Spieler</b>	7
5.1	Player Component	7
5.2	Input System	7
<b>6</b>	<b>Spielfiguren</b>	9
6.1	Bewegung	9
6.2	Attribute Component	9
6.3	Inventory Component	9
<b>7</b>	<b>Pathfinding</b>	11
<b>8</b>	<b>Kamera</b>	12
<b>9</b>	<b>User-Interface</b>	13
9.1	Action-Points Leiste	13
9.2	Dynamische Ability-Icons	13
9.3	OverHeadUnitInfo	13
<b>10</b>	<b>3D Modelling</b>	14

<b>11 Animationen</b> .....	16
11.1 Motion Capture Aufnahmen .....	16
11.2 Einbindung der Animationen .....	16
<b>12 Sounds</b> .....	18
<b>13 Effekte</b> .....	19
<b>14 Fazit</b> .....	20

# 1

---

## Einleitung

Civil War Nation wurde im Rahmen eines interdisziplinären Teamprojektes von 6 Informatikern und einem Artist an der Hochschule Trier entwickelt. Es handelt sich um ein rundenbasiertes 3D Strategiespiel, welches mit Hilfe der aktuellsten Unity Engine realisiert wurde. In der vorliegenden Dokumentation wird auf die technischen Aspekte der Umsetzung eingegangen um einen Überblick des Aufwandes zu bieten.

Civil War Nation was developed by six computer scientists and one artist in the context of an interdisciplinary team project at the university of applied sciences in Trier. It is a round-based 3D strategy game that was developed using the latest version of the Unity engine. In an attempt to provide an overview of the work input this paper will address technical aspects of the game's implementation.

## **2**

---

### **Squad Builder**

Der Squad Builder ermöglicht es beiden Spielern ihre Teams zusammenzustellen. Die Spieler wählen abwechselnd ihre Figuren, bis die vorgegebene Teamgröße erreicht wurde.

Die linke Hälfte des Bildschirms zeigt die Teamzusammenstellung von Spieler 1. Er kann für jede Einheit einzeln Primärwaffe, Sekundärwaffe und 2 Utility Items wählen. Die Auswahl erfolgt über eigens programmierte Dropdown-Menüs, die sich dynamisch aus den vorgegebenen Enumeratoren und Icon-Listen generieren. Das Bestätigen der Zusammenstellung erfolgt über den Bestell-Button unterhalb der linken Anzeigeneiste. Die ausgewählten Einheiten werden in der linken Anzeigeneiste durch die ihnen zugeteilte Ausrüstungsgegenstände dargestellt.

Die rechten drei Buttons ermöglichen es Spieler 2 seine Figuren aus vorgegebenen Archetypen auszuwählen. Die gewählten Einheiten werden in der rechten Anzeigeneiste durch ihre jeweiligen Icons dargestellt.

# 3

---

## Levelaufbau

Das Level wird durch ein Skript (BattlefieldCreator) aufgrund der Plane generiert. Je größer der Localscale der Plane, desto größer wird das Level. Auf einer Skalierung von 1/1/1 entsteht ein  $10 * 10$  Grid aus Quads(Zellen). Bei Skalierungen im Komma Bereich werden dem entsprechen viele Zellen erstellt. Zum Beispiel bei  $x = 1,6$  und  $z = 1,7$  entsteht ein  $16 * 17$  Feld.

```
1 // Initialisiert alle Zellen
2 for (float z = 0; z > -(sizeZint); z--) {
3     for (float x = 0; x < (sizeXint); x++) {
4         GameObject zelle = GameObject.CreatePrimitive(PrimitiveType.Quad);
5         zelle.transform.Rotate(new Vector3(90, 0, 0));
6         zelle.AddComponent<Cell>();
7         zelle.tag = "Cell";
8         zelle.name = x + "|" + -z;
9         zelle.transform.position = new Vector3((x + 0.5f), 0.001f, (z - 0.5f))
    );
```

Die Objekte werden durch das Skript ObjectSetter beim Spielstart auf dem Grid platziert. Sollte ein Objekt größer als eine Zelle sein so wird dies in der ObjectComponent durch X und Z wert angegeben. Ebenso wird dort vermerkt ob das Objekt Deckung bietet und die Zelle besetzt.



**Abb. 3.1.** Das finale Level mit Spieleraufstellung

# 4

---

## Manager Objekt

Um die verschiedenen Systeme, die für den korrekten Ablauf der Spielzüge und allgemein spielregeltechnischen Abläufe zu handeln, wurde ein Spielobjekt, das als Manager bezeichnet wird, erstellt. Im folgenden Kapitel wird auf die einzelnen Skripte die an diesem Manager Objekt hängen genauer eingegangen.

### 4.1 Manager System

Das Manager System ist für den korrekten Ablauf der einzelnen Züge zuständig. Es zählt die Runden hoch, stellt sicher, dass nur das die Eingabe des Spielers, der aktuell an der Reihe ist, abgehandelt wird, merkt sich die aktuell ausgewählte Spielfigur, damit das User-Interface korrekt dargestellt wird, fügt jedem Spieler seine Spielfiguren zu und setzt die Spielfiguren zu Beginn der Sitzung an zuvor festgelegte Positionen.

### 4.2 Shooting System

Das ShootingSystem beschreibt das Schussverhalten der verschiedenen Waffen im Spiel. Dabei werden hier verschiedene Boni und Mali auf die Angriffskraft der Waffe verteilt, um so situationsabhängig agieren zu können. Zum Beispiel wird auch die Distanz zum Gegner in Betracht gezogen . Die Veränderung des Angriffs wird wie folgt unterteilt:

#### Bonus:

- Das Ziel befindet sich nahe zum Angreifer
- Das Ziel hält sich hinter keiner Deckung auf

#### Malus:

- Das Ziel befindet sich weit weg vom Gegner
- Das Ziel befindet sich hinter einer hohen Deckung
- Das Ziel ist in einer Rauchgranate nur schwer sichtbar

Hinzu kommt noch die Frage ob ein Spieler überhaupt die Möglichkeit besitzt zu schießen, dies kann durch mangelnde Ressourcen wie Munition der Fall sein. Des Weiteren wird eine Wahrscheinlichkeit berechnet, die zufällig bestimmt, ob ein Spieler seinen Gegner trifft, diese steigt jedoch je besser sich der Angreifer mit den oben erwähnten Boni aufgestellt hat.

### 4.3 Inventory System

Das Inventory System wird aufgerufen sobald ein Spieler eine der folgenden Aktionen ausführt um die Anzahl der im Inventar der Spielfigur enthaltenen Gegenstände zu verringern:

- Nachladen der Primärwaffe
- Einsatz von Handgranaten
- Einsatz von Tränengas
- Einsatz von Rauchgranaten
- Einsatz von Molotovcocktails

### 4.4 Player Assistance System

Das Player Assistance System wird dazu benutzt, um die Kachel einzufärben, über der man sich mit der Maus befindet. Die Information, welche Kachel ausgewählt ist, wird durch das Input System gesetzt. Zusätzlich wird beim auswählen einer Bewegung, der Pfad eingezeichnet, den die Figur wählen wird. Dadurch lässt sich erkennen, ob eine Figur einen Weg durch beispielsweise Feuer zurücklegen muss.

### 4.5 Ability System

Die Funktionen im AbilitySystem werden über das InputSystem aufgerufen. Je nach Wahl des Spielers wird die dazugehörige Methode ausgeführt. Diese kopiert einen Effekt, weist diesem Zelle und Partikeleffekt zu und ruft den Start der Animationen auf. Dem Effekt-GameObject wird ein weiteres Script angefügt, welches die Zellen überwacht und gegebenenfalls Schaden an Figuren, die diese betreten, austeilte. Nach einer festgelegten Anzahl an Runden zerstört sich das Objekt selbst.

### 4.6 Health System

Die Schadensberechnung sowie auch Heilung erfolgt durch das HealthSystem. Hierfür wird für die Schadensberechnung beispielsweise die Angriffskraft der Waffe betrachtet. Beim Einsatz eines Medipacks wird einfach ein definierter Wert an Lebenspunkten wieder hergestellt. Der eigentliche Vorgang wird dabei unterteilt in Berechnung des Schadens- bzw. Heilwertes und das entsprechende verändert der Werte, dies sorgt für weitere Übersicht und lässt sich wie folgt verbildlichen:

```
1 public void doDamage(AttributeComponent attackingPlayerAttr , PlayerComponent
                      attackingPlayerComp , AttributeComponent damageTakingPlayerAttr , int
                      damageFlag)
2 {
3     switch(damageFlag)
4     {
5         case SHOOT:
6             int damage = generateShootDamage( attackingPlayerAttr ,
7                                              damageTakingPlayerAttr );
7             inflictShootDamage( attackingPlayerAttr , attackingPlayerComp ,
8                                 damageTakingPlayerAttr , damage );
8             break;
9
10        default:
11
12            break;
13    }
14 }
```

# 5

---

## Spieler

Das Spieler Objekt enthält als Kindobjekte seine Spielfiguren. Als Skripte hängen ihm eine Player Component, sowie eine Input Component an.

### 5.1 Player Component

```
1 GameObject[] figurines = new GameObject[3]; // Alle Figuren ueber die ein  
    Spieler verfügt  
2 public int actionPoints = 0; // Anzahl an verfügbaren Aktionspunkten  
3 int maxAP; // Maxcap fuer AP
```

Das Skript speichert die maximale Anzahl an Aktionspunkten, die für die verschiedenen Fraktionen variieren, füllt nach dem Ende der Runde die Aktionspunkte beider Fraktionen auf und stellt sicher, dass dabei die Zahl der erhaltenen Aktionspunkte nicht die Grenze überschreiten.

#### Maximale Aktionspunkte Rebellen

(n Figuren + 4) \* 2

#### Maximale Aktionspunkte Regierungstruppen

(n Figuren + 2) \* 2

#### Aktionspunktregeneration Rebellen

Aktionspunkte + Anzahl an Figuren + 4

#### Aktionspunktregeneration Regierungstruppen

Aktionspunkte + Anzahl an Figuren + 2

**Abb. 5.1.** Berechnung der Aktionspunkte

### 5.2 Input System

Jedes Spielerobjekt, welches die verschiedenen Spielfiguren besitzt, besitzt jeweils ein Input System. Dieses Inputsystem ist für das Starten von Aktionen, sowie die

Auswahl von Zielen oder Figuren zuständig. Mit Hilfe von Raycasts in die Szene hinein, wird ermittelt, welches Objekt oder Zelle getroffen wird.

Das Input System wird auch durch das UI System benutzt. Wenn Buttons für Aktionen gedrückt werden, wird im Input System die entsprechende Aktion ausgewählt, die als nächstes ausgeführt werden soll. Diese Aktionen werden dann an die entsprechenden Systeme (Movement-, Ability- oder Shooting System) weitergegeben, wo die Logik ausgeführt wird.

# 6

---

## Spielfiguren

### 6.1 Bewegung

Bei unseren Spielfiguren um kleine Plastiksoldaten handelt, die sich in einem kindlich dargestelltem Nah-Ostkonflikt befinden. Die Bewegung der Einheiten wird daher über eine Parabelkurve angedeutet, an der sich die Figur beim laufen entlang bewegt. Somit wird der Eindruck erzeugt, die Figur werde wie von einer unsichtbaren Hand in einem Brettspiel über das Feld bewegt.

### 6.2 Attribute Component

Die Attribute Component dient dazu, die spieletechnischen Werte der Figur abzuspeichern. Sollte es sich um eine Figur aus der Fraktion "Regierungstruppen" sein wird die Attribute Component mit den passenden Werten initialisiert, die für die ausgewählte Klasse vordefiniert wurden. Handelt es sich um eine Figur aus der Fraktion "Rebellen" wird diese mit den Ausrüstungsgegenständen bestückt.

### 6.3 Inventory Component

Jeder einzelnen Spielfigur wird eine Inventory Component angehängt. In dieser wird das gesamte Inventar der jeweiligen Figur gespeichert. Das Inventory System kümmert sich dabei um die Berechnungen und Aktualisierung der Inventory Komponenten.

Es folgt ein Auszug der verschiedenen Variablen:

```
1 //Inventar (primaerwaffe , sekundaerwaffe , equipment1 , equipment2) siehe  
    Enums.cs  
2 public Enums.PrimaryWeapons primaryWeaponType;  
3 public Enums.SecondaryWeapons secondaryWeaponType;  
4 public Enums.Equipment utility1;  
5 public Enums.Equipment utility2;  
6  
7 public WeaponComponent primary; //Primaerwaffe  
8 public WeaponComponent secondary; //Sekundaerwaffe  
9 public bool isPrimary; //Ist Primaerwaffe ausgewählt?  
10 public int amountSmokes; //Anzahl Rauchgranaten
```

```
11 public int amountTeargas; //Anzahl Teargas
12 public int amountGrenades; //Anzahl Granaten
13 public int amountMolotovs; //Anzahl Molotovs
14 public int amountMediKits; //Anzahl Medikits
15 public int amountMagazines; //Anzahl Magazine//Anzahl Magazine
```

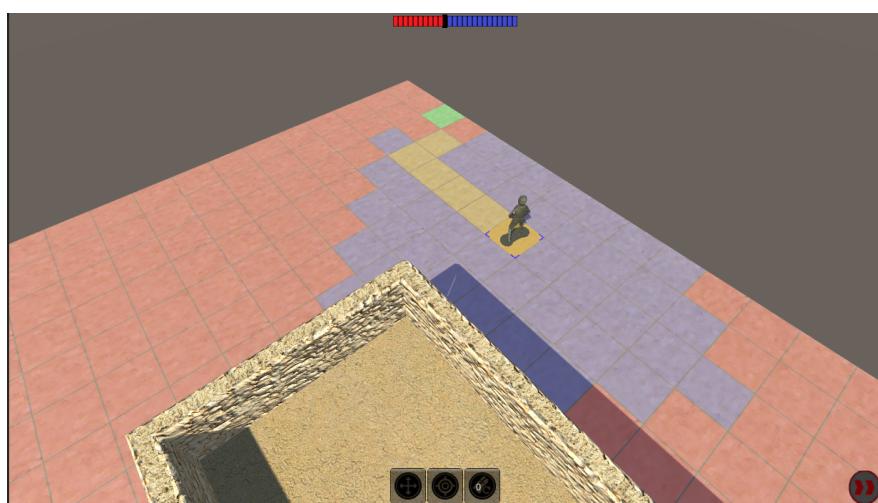
---

## Pathfinding

”Civil War Nation“ benutzt ein in Zellen aufgeteiltes Spielfeld. Um die Bewegung der Figuren auf diesem Spielfeld zu ermöglichen, müssen die günstigsten Pfade gefunden werden. Hierbei wird der ”Dijkstra Algorithmus“ eingesetzt, der von der aktuell ausgewählten Figur die Entfernung zu allen anderen Zellen auf dem Spielfeld zu berechnen. Diese Entfernung wird wiederum benutzt um Aktionen mit begrenzter Reichweite, wie schießen, Granaten werfen, oder Laufen, auf ihre Verfügbarkeit zu überprüfen. Der Dijkstra Algorithmus wurde gegenüber dessen Erweiterung, den A\*-Algorithmus gewählt, da wir ungerichtet über den Graphen laufen möchten, und somit die Kosten zu allen umliegenden Knoten erhalten möchten.

In dem Bild sind alle verschiedenen Zelleinfärbungen auf einmal zu sehen.

- **Gelb mit Rand:** Aktuell ausgewählte Figur
- **Blau:** Terrain in Bewegungsreichweite
- **Rot:** Terrain in Angriffsreichweite
- **Gelb:** Geplanter Pfad
- **Grün:** Zelle, über die sich die Maus aktuell befindet



**Abb. 7.1.** Verschiedene Zelleinfärbungen

---

## Kamera

Die Kamera ist eine beweglich Orbitkamera mit Zoom und Rotation die sich nur innerhalb des Levels bewegen kann. Realisiert wird dies durch einen konstanten Focus auf ein bewegliches, unsichtbares Objekt. Die Kamera lässt sich über das Feld bewegen in dem die Maus an den jeweilige Rand bewegt wird der in die gewünschte Richtung führt.

Die folgende Funktion prüft ob die Kamera im Feld ist:

```

1  public bool inBattlefield()
2  {
3      bool inField = true;
4      if (target.transform.position.x < 0) {
5          inField = false;
6          target.transform.position = new Vector3(0, target.transform.
7              position.y, target.transform.position.z);
7      }
8      if (target.transform.position.z > 0) {
9          inField = false;
10         target.transform.position = new Vector3(target.transform.position.x
11             , target.transform.position.y, 0);
11     }
12     if (target.transform.position.x > (plane.transform.position.x * 2)) {
13         inField = false;
14         target.transform.position = new Vector3((plane.transform.position.x
15             * 2), target.transform.position.y, target.transform.position.z)
15     }
16     if (target.transform.position.z < (plane.transform.position.z * 2)) {
17         inField = false;
18         target.transform.position = new Vector3(target.transform.position.x
19             , target.transform.position.y, (plane.transform.position.z * 2))
19     }
20     return inField;
21 }
```

# 9

---

## User-Interface

Das UI besteht aus verschiedenen Komponenten.

### 9.1 Action-Points Leiste

Die Aktionspunkte Leiste am oberen Bildrand zeigt für beide Spieler die maximalen sowie die aktuell verfügbaren Aktionspunkte an.

### 9.2 Dynamische Ability-Icons

Wenn ein Spieler eine Einheit auswählt, so werden am unteren Bildrand die erforderlichen Aktionsbuttons angezeigt. Es werden nur die Buttons dargestellt, deren Aktionen von der ausgewählten Figur durchgeführt werden können. Anzahl und Positionierung werden dynamisch justiert.

Die Zahl auf dem Reload Button gibt die Anzahl der verbleibenden Magazine an.

### 9.3 OverHeadUnitInfo

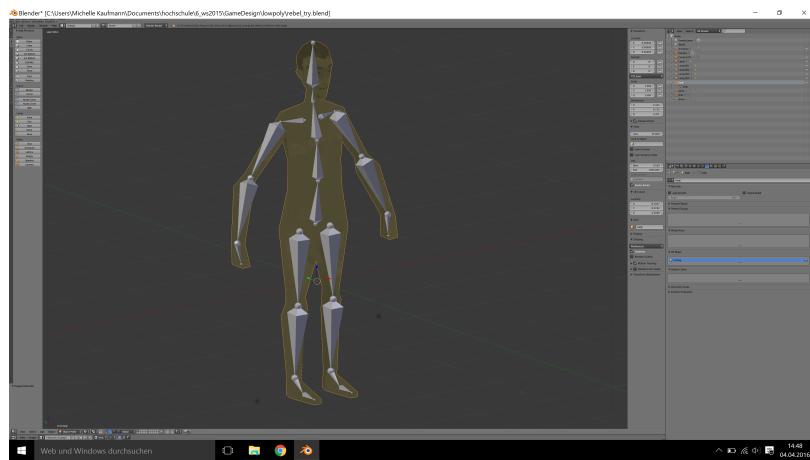
Bei ausgewähltem Angriff, sowie durch drücken der Tabulator Taste, werden für alle Figuren Overhead-Infos dargestellt. Diese zeigen die aktuellen Lebenspunkte und, sofern eine Schusswaffe ausgewählt ist, die aktuell im Magazin verbleibende Munition. Für Regierungstruppen wird zusätzlich noch ein Klassen-Icon angezeigt.

# 10

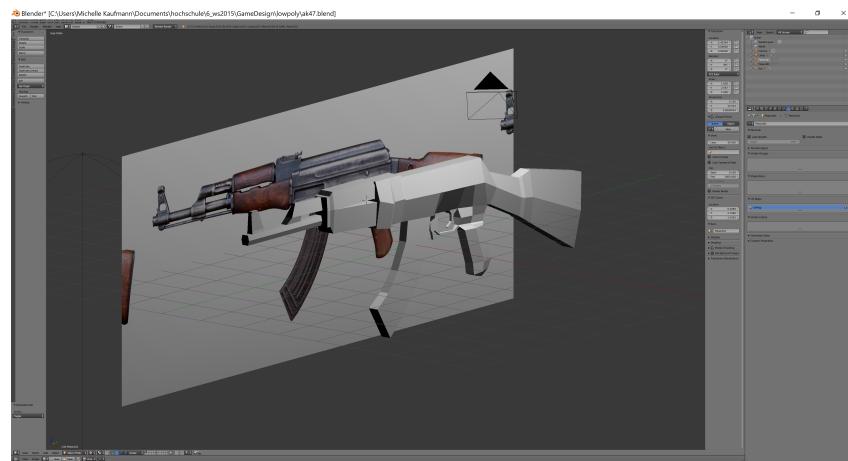
---

## 3D Modelling

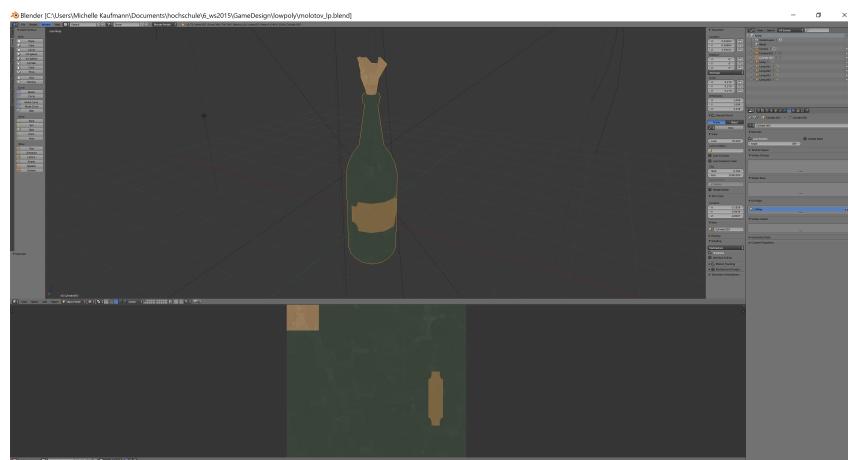
Sowohl die Charaktere als auch Assets wurden ausschließlich in Blender gemodelet. Hierbei wurde sich stark an reellen Vorgaben, was Kleidung oder einprägsame Details betrifft, orientiert. Um den angestrebten Lowpoly-Stil konstant umzusetzen wurden teilweise erst Highpoly-Modelle erstellt um diese in der sogenannten "retopology" später detailarmer zu gestalten.



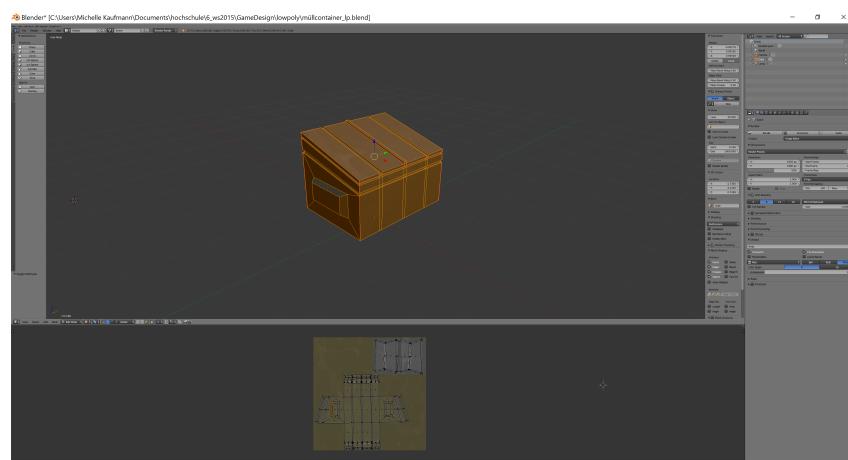
**Abb. 10.1.** Charaktermodell in Blender



**Abb. 10.2.** Waffenmodellierung in Blender



**Abb. 10.3.** Propmodellierung in Blender



**Abb. 10.4.** Propmodellierung in Blender

# 11

---

## Animationen

### 11.1 Motion Capture Aufnahmen

Alle eingebundenen Animationen wurden zuvor von den Projektmitgliedern mit dem Motion Capture System im Keller der Hochschule Trier aufgenommen. Nachdem eine überschaubare Anzahl an qualitativ hochwertigen Aufnahmen ausgewählt wurden, gingen diese direkt als .bvh Format zur Weiterverarbeitung.

### 11.2 Einbindung der Animationen

Die Einbindung der Animationen wurde wie das Modelling ebenfalls direkt in Blender vorgenommen. Mithilfe des Add-Ons “MakeWalk“ ist es einfach, falls der selbst erstellte Rig keine Fehler aufweist, die rohen Motion Capturing Aufnahmen in die 3D Software zu übertragen. Für jede Charakterklasse im Spiel wurde eine eigene “Pose Libraries“ erstellt. Diese erwiesen sich durch einen strukturierteren und übersichtlichere Importierung in Unity als äußerst nützlich.

Diese Animationen wurden in Unity importiert, und mit Hilfe eines Animators in eine Animation State Machine überführt, die je nach gewählter Aktion und aktueller Haltung (Einhändige Waffe, Zweihändige Waffe, Nahkampfwaffe, Einsatzzschild) , die korrekte Animation auswählt und abspielt.

Zusätzlich wurden an einige der Animationen an bestimmten Zeitpunkten in der Animationen Funktionen angehängt. Somit kann erreicht werden, dass beispielsweise der Schusssound zum korrekten Zeitpunkt ausgelöst wird, oder die Granaten im entsprechenden Frame erstellt, oder geworfen werden.

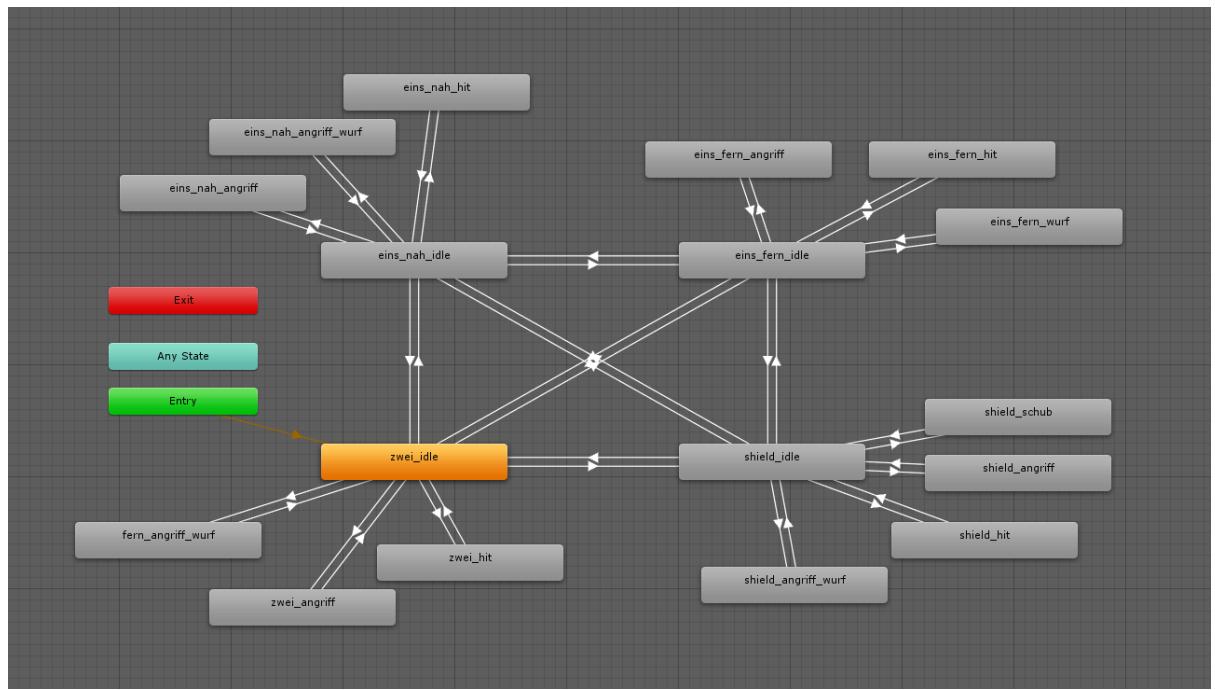


Abb. 11.1. Animation Tree der Polizei Charaktere

## **12**

---

### **Sounds**

Für den Sound wurden größtenteils hochwertige Audioaufnahmen aus dem "The GameaAudioGDC Bundle Part 2" verwendet, einem großen Paket an Audiofiles das kostenlos durch die Game Developer Conference bereitgestellt wurde. Die Audioeffekte der Kriegselemente, wie Schüsse und Detonationen, sollen sehr echt wirken, andere Audioelemente eher kindlich, um den Kontrast der jungen Kontrahenten und die Ernsthaftigkeit des Krieges zu verdeutlichen. Beim Anklicken der GUI-Elemente des Squadbuilders wurde bei den Tönen mit verschiedenem Pitch gearbeitet, um den Sound nach einer Weile nicht eintönig wirken zu lassen, aber auch der Soundeffekt bei einem Sprung wird durch Zufallsprinzip aus vier Audioclips ausgeählt. So erhalten die Töne Varianz und wirken auch nach mehrmaligen Hören nicht repetetiv. Programmatisch wurde der Sound über mehrere Skripte übergreifend implementiert und an den jeweiligen passenden Stellen einfach abgespielt.

# 13

---

## Effekte

Für die verschiedenen Fähigkeiten sowie schießen wurden Partikeleffekte erstellt. Dazu wurde das Partikelsystem von Unity genutzt.



**Abb. 13.1.** Von Links nach Rechts: Explosion, Feuer, Gas, Rauch, Heileffekt

## **14**

---

### **Fazit**

Schlussendlich ließ sich feststellen dass die Entwicklung eines (rundenbasierten) Strategiespiels mit sehr viel Vorarbeit verbunden ist, was einen ersten Prototyp sehr verzögert. Es gibt sehr viel Logik zur Zugreihenfolge, der Einheiten Auswahl oder der Bewegung über das Spielfeld zu implementieren. Um möglichst schnell einen ersten spielbaren Prototyp zu haben, wurde stellenweise sehr unvorsichtig und undurchdacht implementiert. Dadurch kam es dazu, dass sehr viele Strukturen durcheinander gereicht wurden, Objekte und Variablen an verschiedenen Stellen unnötig oft gespeichert wurden, was es Fehlern sehr leicht machte sich einzuschleichen.

Durch modulares Entwickeln beispielsweise durch Abstraktion mit Events, Listenern oder Interfaces an den entsprechenden Stellen, hätten vielen Fehlern vorbeugt werden können. Erwähnenswert ist auch der Unterschied zwischen Code der vor Ort mit den Teilnehmern entwickelt wurde, gegenüber Code, der im "Home Office" entwickelt wurde. Code der in Teamsitzungen entstanden ist, machte hierbei seltenst Probleme, da sich über mögliche Schnittstellen unterhalten werden konnte, welche Variablen wo bereits vorhanden werden konnten, und allgemein ein besserer Einblick über den Code von anderen entstanden ist.