QubeShoot

Ein 3D Top-Down Isometrisches Zombie Shooter Spiel

Schaubild 1: Das Logo des Spiel's

Auftraggeber	Kontakt
Ralf Kowalewski	kowalewski@bbs-me.de

Auftragnehmer	Kontakt
Viktor Legodzinski	viktor.legodzinski@bbs-me.org

Inhaltsverzeichnis

Abbildungsverzeichnis

Stichwortverzeichnis

Glossar

Wort	Bedeutung
GLTF	Ein Open Source Modellformat
DirectX	Eine Grafiktreiber Schnitstelle
API	Eine Schnitstelle
.NET	Ein Programminfrastruktur
Mono	Eine Open Source version von .NET
C#	Eine managed Programmiersprache
Realtime	Echtzeit
GameObject	Ein Spielobject in Unity
Szene	Eine Renderinstanz
Keyframe	Ein Animationszeitpunkt
Blending	Animationsinterpolierung
Knochen	Ein Knochen eines animierten Objekts
Locomotion	Fortbewegung
Loader	Ein Ladesystem
Map-Loader	Ein Spielkarten-Ladesystem
Mod-Loader	Ein Modifikations-Ladesystem
Model-Loader	Ein Modell-Ladesystem
Map	Eine Spielkarte
Parsen	Einlesung von Daten
Referenzpunkte	Ein 3D Punkt in einem Bereich
Animations Baking	Eine feste Animation (nicht Dynamisch)
XSD Definition	Typsicherheits Definitionsdatei
PreFab	Voraberstellte Ressource
Transform	Position, Rotation und Skalierung eines Objekts
NavMesh	Ein Navigationsraster
NavMeshTag	Ein Navigationsraster Marker
NavMeshBuilder	Ein Navigationsraster Erbauer
RPM	Schüsse pro Minute
Projektil	Ein fliegendes Projektil
HitScan	Laser Zielerfassung
Tracer	Temporärer Laser
AOE	Zu abdeckende Fläche
IDE	Entwicklungsumgebung
VS	Visual Studio
Hub	Ein ort mit vielen Funktionen
Viewport	Ein Renderbildschirm
Quaternion	Ein zahlenbereich (-1;-1;-1;-1)
Vector3	Eine 3D Position (-x;-x;-x)
MS	Microsoft
Managed	Verwaltet (Virtuelle Maschine unterliegend)
VM	Virtuelle Maschine
RPC	Remote Procedure Call

Command	Befehl zum Server
Sync	Synchronisiert/Synchronisation

1. Kurzbeschreibung des Projektes

Das Ziel dieses Projekt‘s war es ein Videospiel zu machen, welches ein Top-Down isometrisches Zombie Spiel darstellen soll. Welches eigenerstellte 3D Modelle, Animationen und Benutzerdefinierte Levels beeinhalten sollte.

2. Abstract

The Main goal of this project was to make a videogame, that resembled a top-down isometric zombie game. Which had to involve custom 3D Models, Animations and Custom Levels.

3. Hauptteil

3.1. Einleitung

Dieses Projekt wird dafür genutzt, um die schüler der BFI auf management, können und teamarbeit zu testen. Gefördert wird der nutzen von verschiedenen Arbeitsumgebungen, und Arbeiten.

3.2. Methode/Vorgehensweise

Die Vorgehensweise verlangt das die Erforderlichen Dokumente (Lastenheft, Pflichtenheft) fertig sind, damit die Projektarbeit gründlich angefangen werden kann. Nur dann ist Sichergestellt das der Arbeitgeber mit dem Resultat klarkommen muss.

3.3. Komplikation während der Projekt Bearbeitung

3.3.1. Nicht Erreichte/Vollständige Vorgaben

- Multiplayer: Der Multiplayer aspekt des Spieles wurde in der ersten Version ausgehalten, aus Zeitmangel.

  **Der Multiplayer wurde in der Erweiterten Version innerhalb der Work-from-Home Zeit zusätzlich Hinzugefügt.

- Eigene Engine: Dieser aspekt wurde schon sehr früh wervorfen, da nach gründlicher einlesung in die thematik, wir festgestellt haben das es uns massiv and Zeit fehlt, eine richtige 3D Engine zu erstellen.

- Benutzerdefinierte Waffen und Charaktere: Das Nutzen von Benutzererstellten Modellen, sowie den zugehörigen Animationen wurde etwas später (Nach dem umstieg auf Unity) verworfen. Es war technisch möglich, allerdings gab es komplikationen mit dem GLTF Modell-Loader, deshalb wollten wir den nutzen von Runtime geladenen modellen aufs minimum beschränken.

- Datenbank Interaktion: Das Verbinden mit und das Hinzufügen von neuen Datensätzen du einer Datenbank konnte nicht rechtzeitig implementiert werden, hierbei allerdings nur durch Zeitliche gründe. Dies ist allerdings eine Komponente die definitv noch Reingepatcht wird in den kommenden Updates.

  **: Die Datenbank Interaktion wurde samt Multiplayer ebenso Hinzugefügt innerhalb der Erweitern Version.

3.4. Hardware

Die hardware angaben sind als mindestwerte für den Konsumer gedacht:

- **Betriebssystem:** Windows 7 SP 1; 64 bit
- **Prozessor:** Core i3 @2.xx GHz
- **Arbeitsspeicher:** 3 GB RAM
- **Grafik:** GeForce GT 720 1GB VRAM; DX 11 Benötigt.
- **DirectX:** Version 11
- **Speicherplatz:** 4 GB verfügbarer Speicherplatz

3.5. Software

3.5.1. Betriebssytem

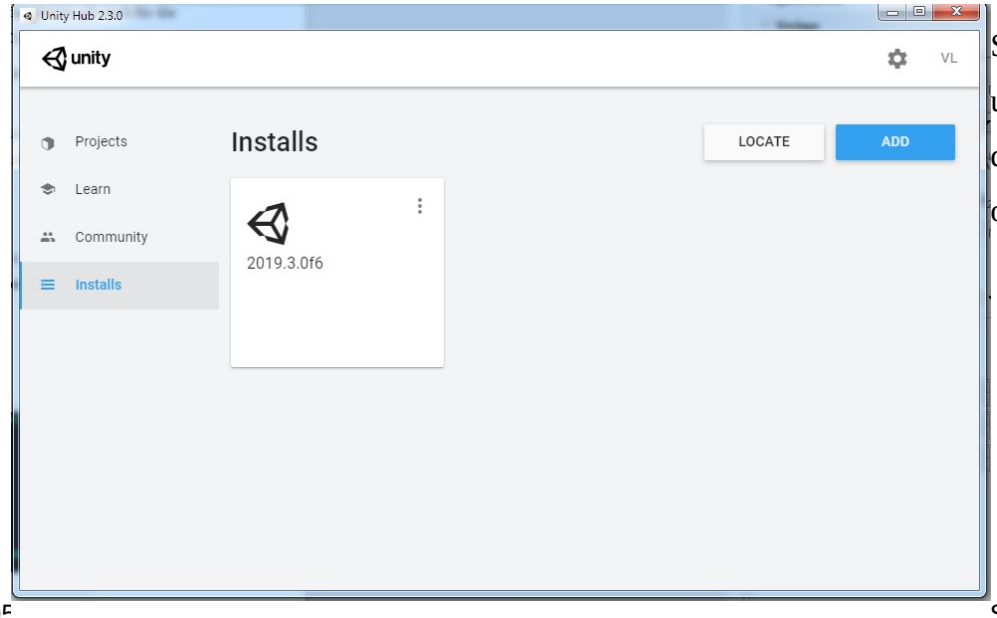
Das Betriebssystem zum Arbeiten sowie zum Anwenden des Produktes war Windows.

Dafür gibt es verschiedene gründe die hier einmal aufgelistet sind:

- Windows bietet objektiven einfacheren Support für DirectX Anwendungen, da es auch eine API ist die von Microsoft selber erstellt wurde, das gleiche gilt auch für die Implementierung von .NET/Mono (Welche beide in C# genutzt werden)
- Es gibt wesentlich weniger dokumentation für 3D Spiele Entwicklung für Linux als für Windows, auch wegen dem oben genannten grund.
- Für Linux gibt es keine nativ entwickelten Powertools wie z.B.: Photoshop(Texturing), AutoDesk 3DS Max (Modelling) und keine Leistungsstarke IDE wie Visual Studio, wessen in Verbindung mit Unity quasi benötigt wird.

3.5.2. Unity Konfiguration & Einrichtung

Die Konfiguration und erstmalige Installation von Unity geht schnell, dafür wird man genötigt einen „Hub“ Runterzuladen, welcher für einen die verschiedenen Unity Versionen und verschiedene Projekte verwalten kann.



Siehe: https://unity3d.com/de/get-unity/download

Schaubild 2: Unity Hub: Install Sektion

Nach dem erstmaligen starten vom Unity hub, muss man entweder eine vorhandene Unity Installation auswählen, oder eine neue Runterladen. Siehe Schaubild 2.

Wenn man eine Unity Installation hat, kann man ein neues Projekt anfangen oder ein altes Fortsetzen. Siehe Schaubild 3.

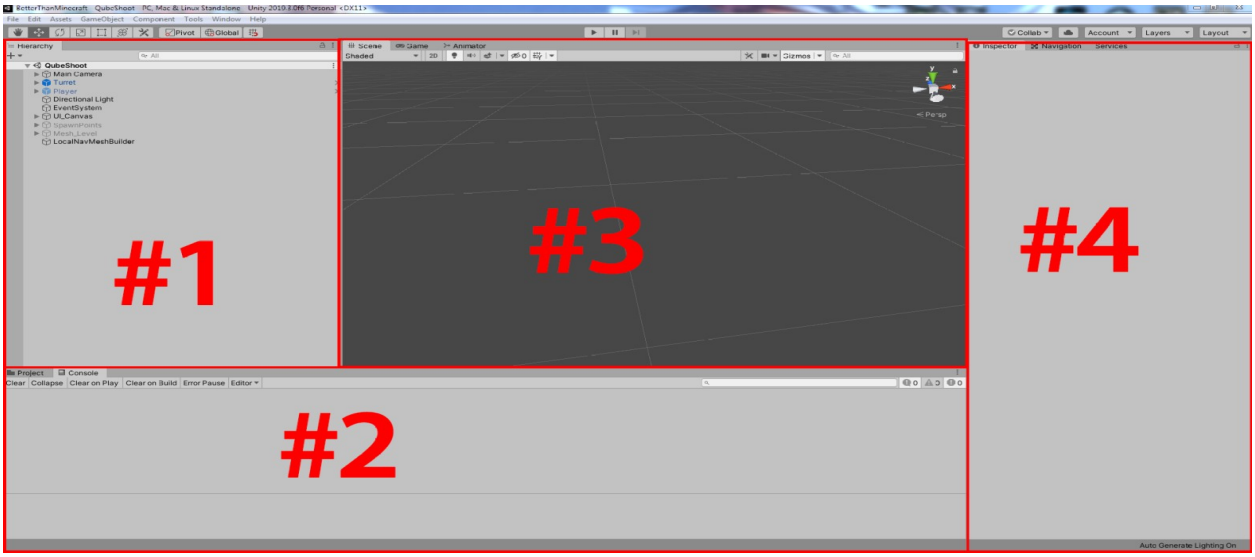


Schaubild 4: Unity

Nachdem man ein Unity Projekt geöffnet hat, wird man mit dem folgendem fenster begrüßt. Einige wichtige Komponenten von Unity hier beschriftet:

- #1: Projekt Hierarchy: Hier wird alles zur derzeitigen „Szene“ gezeigt, jedes einzelne „GameObject“ wird hier aufgelistet welches in der aktuellen Szene vorhanden ist.
- #2: Projekt View & Console Output: Die untere Leiste kann dafür genutzt werden um die Projektmappe zu durchforsten und womöglich Ausgaben vom Spiel zu lesen und fehler zu Erkennen.
- #3: Viewport: Dies ist die womöglich wichtigste Komponente, es ist die Anzeige der derzeitigen Szene in Echtzeit, dies ermöglicht auch die Echtzeit bearbeitung von 3D Modellen und vielem weiteren.
- #4: Inspektor: Der Inspektor ist einer der wichtigsten werkzeuge dieser Engine, er kann dafür genutzt werden um Werte von GameObjects zu ändern, und Komponenten anzupassen, ohne weitere schwierigkeiten.

3.5.3. Modellierung: 3DS MAX 2020

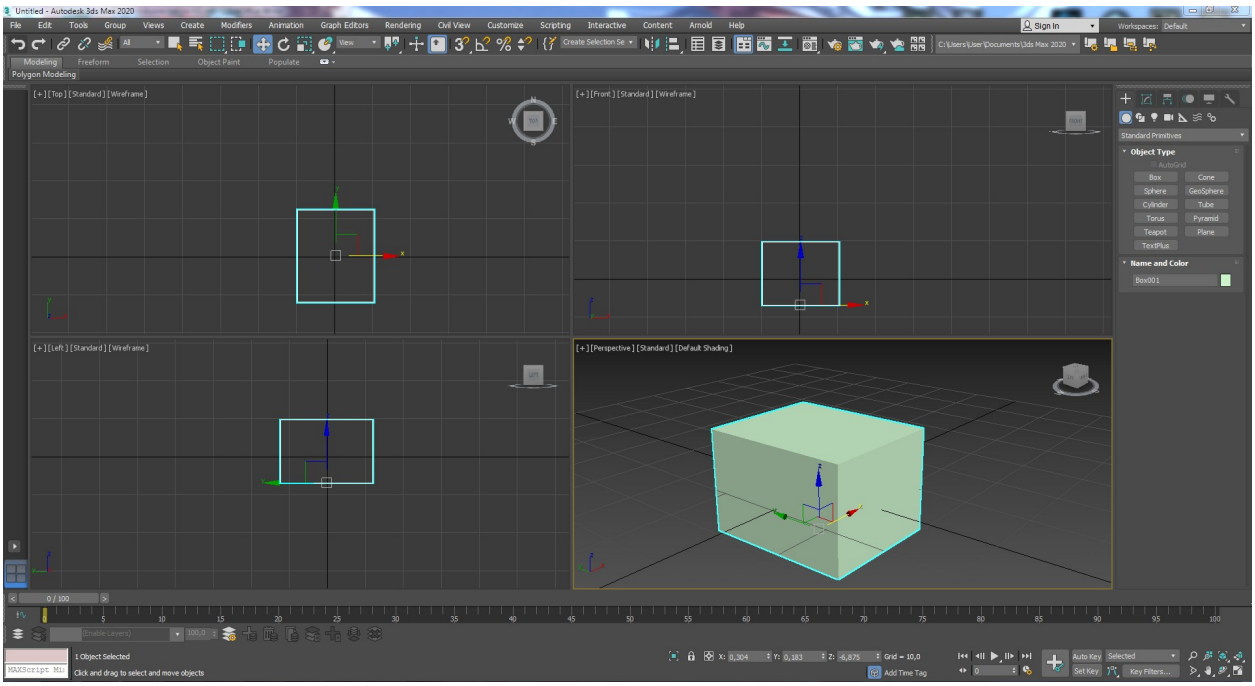


Schaubild 5: 3DS Max und ein kleiner Quader

Zum erstellen und Animieren von 3D Modellen habe ich 3Ds Max 2020 genutzt. 3Ds Max ist in der Industrie eine sehr angesehene Modellierungs Software, und da Autodesk (Die Hersteller) eine Studentenversion anbieten, habe ich mich für dieses Programm entschieden.

Dieses Programm bietet viel mehr anwendungsmöglichkeiten als man zu nutzen braucht, dies ist allerdings keineswegs eine hinderung. Es gab sehr viel Hilfreiche lernmittel für dieses Programm, und selbst Aufgaben wie die Animation waren ein reines „Kinderspiel“.

Im obigen Bild ist ein kleines Quader zu sehen, dafür bietet das Programm alleine ohne voreinstellungen, Knöpfe zur verfügung, welche die Erstellung einfach machen.

3.5.4. Animation

Animationen in der 3D Modellierung sind grundsätzlich einzelne Keyframes welche einem Bone und der dazugehörigen Körperteile sagen, wo und an welcher Stelle sie zu bestimmten Zeiten sein sollten.

Um dies zu Visualisieren wird hirzu eines der Monster aus dem Spiel genutzt:

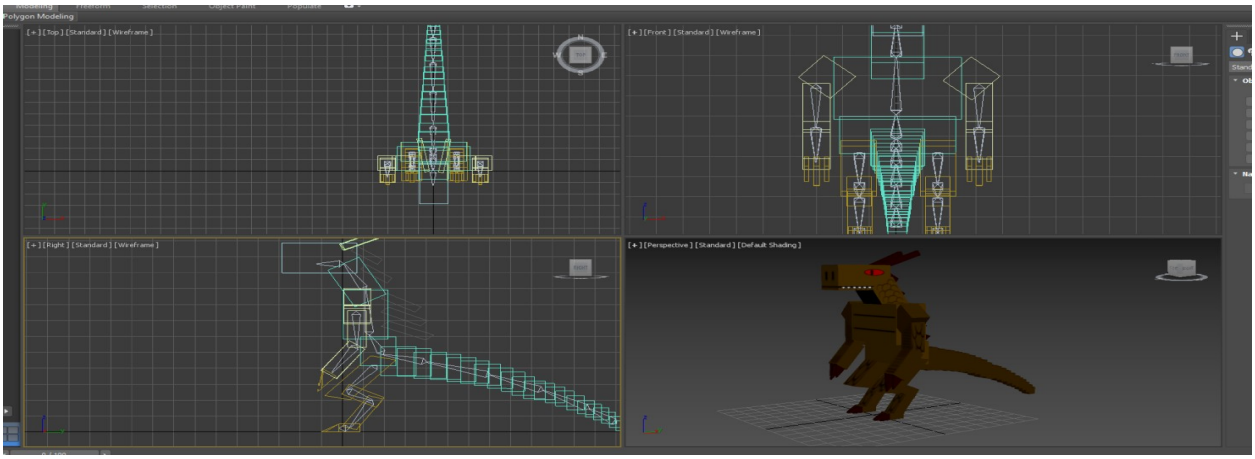


Schaubild 6: QubeShoot Monster: "Tyrant" AkA "Lucy"

Wenn man nun die Bones der Beine des Modell‘s auswählt, kann man unten sofort sehen mit was für Keyframes die einzelnen Knochen versehen sind.

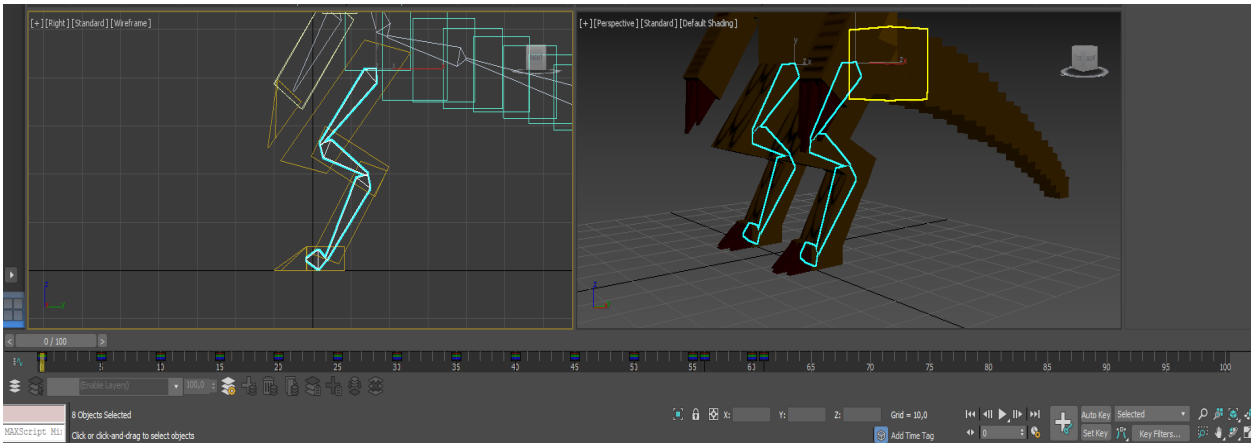


Schaubild 7: Keyframes = Die Rot/Grün/Blau Marker

Wenn man die Zeit voranschiebt auf Keyframe 5, wird man sehen das sich das Modell sofort mit den Knochen mitbewegt. Dies geschieht hierbei rein über Parenting, wobei jedes 3D Modell einem Knochen unterliegt. Es gibt andere methoden wie z.B: Das Rigging mit Skinning, allerdings ist diese Anwendung für unser Spiel nicht nötig gewesen, wegem dem Block Design.

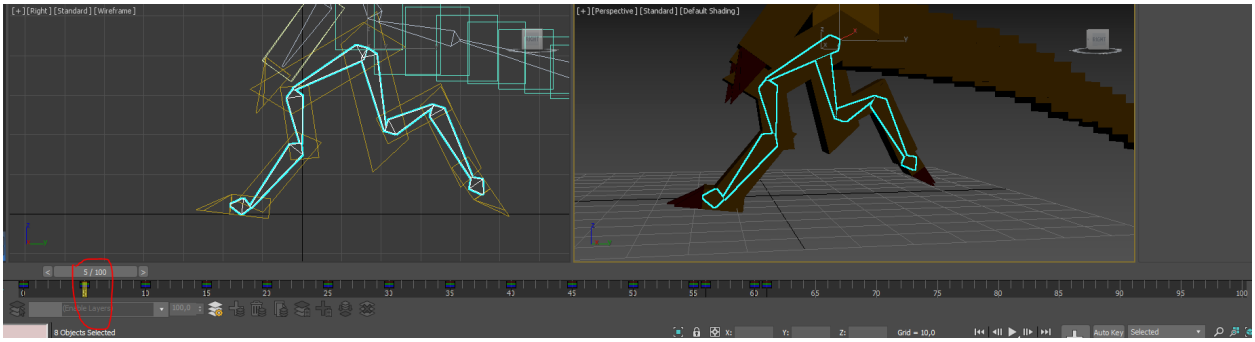


Schaubild 8: Knochen mit Keyframes

Animation wird mit dem Programm zwar leicht angeboten, allerdings ist dieser Prozess sehr Zeitfressend, und kann einfach 2 – 4 Stunden dauern bis man eine gute Laufanimation erstellt hat.

Das ganze hat mit dem Thema der Locomotion zutun, welches vieles mit dem eigentlichen Objekt zutun hat, welches man Animieren will. Aspekte wie die anzahl der Beine, größe des Körpers, etc, können allesamt die Animation anders aussehen lassen, da die Fortbewegung davon abhängt.

3.5.5. Bestandteile des Endprojekts

3.5.6. Mod Support (Maps)

Der Mod-Support beeinhaltet zum Zeitpunkt der Abgabe einen vollen funktionalen „Map-Loader“ für Benutzererstellte Maps. Dies geschieht über das einlesen eines 3D Objektes und das Parsen von Referenzpunkten in dem 3D Objekt.

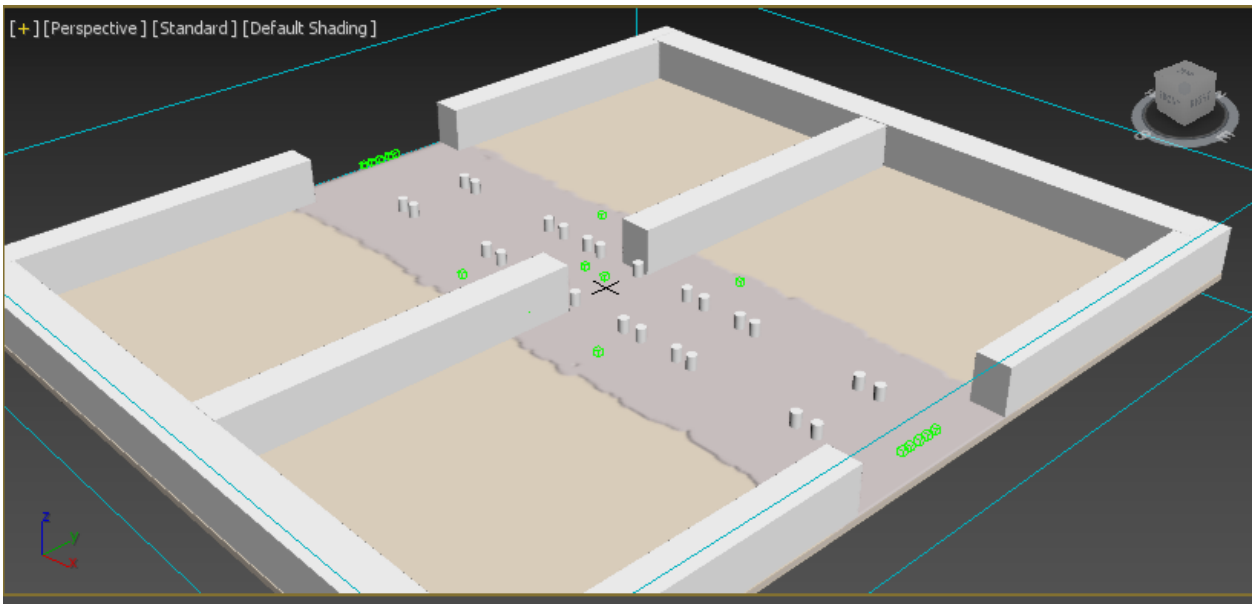
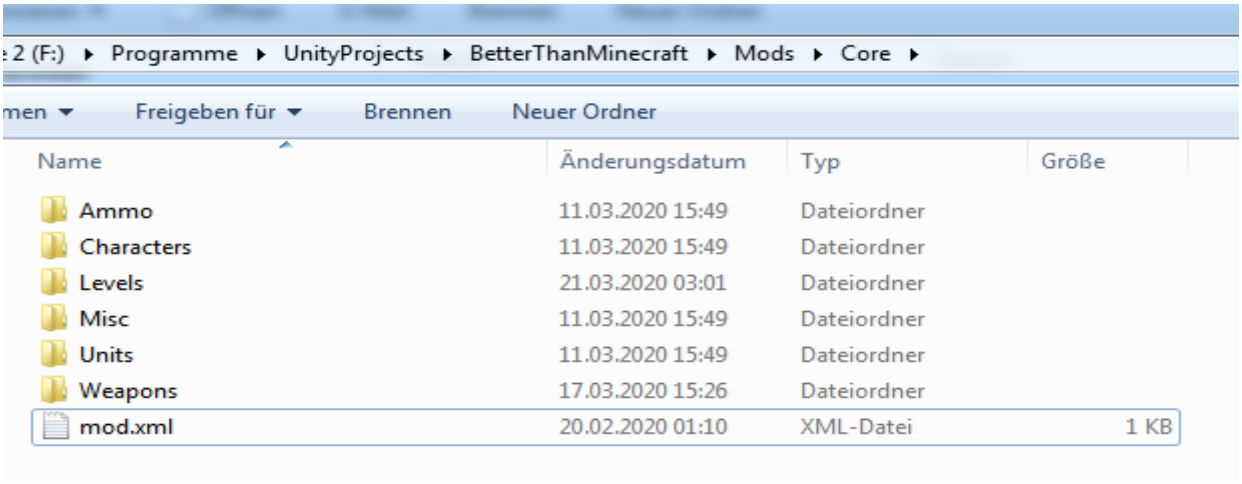


Schaubild 9: Map: Flat Lands

Das Format des Modellformates muss GLTF sein, da dieses eines der wenigen ist, welches man in Unity (dank Drittanbieter werkzeuge) zur Echtzeit laden kann. Ein weiterer grund für die auswahl von GLTF ist die option Animationen ebenfalls von Datei aus zu Laden, dies ist eine sehr Exklusive fähgikeit, da es vor GLTF sogut wie keine echten Alternativen zu Kommerziellen Dateiformaten gab, wie .FBX, .maya, .3ds, .max, etc.

Das Laden von den Maps wird von dem Internen „Mod-Loader“ übernommen. Hierbei wird zu Beginn des Spiels ein ordner namens „Mods“ gelesen und ausgewertet. Hierbei achtet der Loader auf folgende Dateistruktur:



Jede „Mod“ muss seinen eigenen Ordner im „Mods“ Ordner haben, und jede Mod braucht eine „mod.xml“ Datei welche wichtige Information zu dieser einzelnen Mod hat beeinhalten.

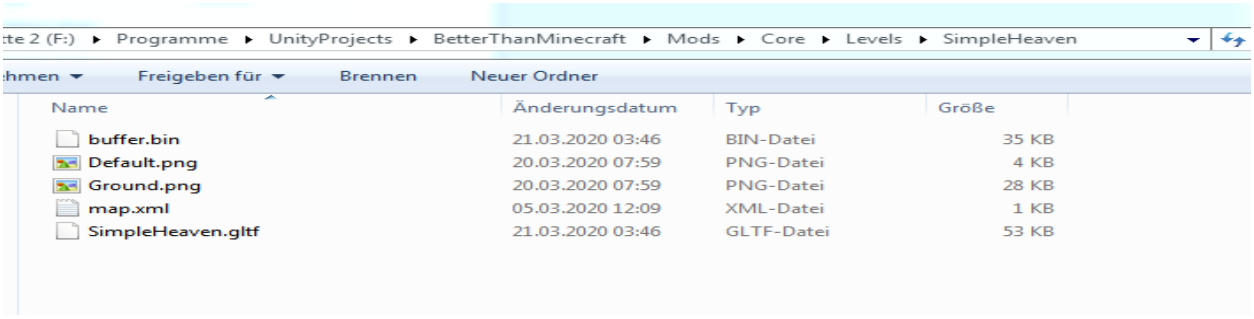
Inhalt der mod.xml:

```
<Mod>
	<Author>Viktor Legodzinski</Author>
	<Name>Core Mod</Name>
	<Description>The Core mod for the QubeShoot Game.</Description>
	<Version>1.0</Version>
</Mod>
```

Wir haben einen sehr simplistischen weg genommen, und unsere Hauptspieldateien auch als „Mod“ deklarifiziert, somit sollte es potenziellen Kunden einfacher sein, die Dateien zu ändern.

Nennenswert ist jedoch, das nur der „Levels“ ordner wirklich verwendbar ist, da alle anderen Aspekte in der ersten Abgabe leider rausgenommen werden mussten. Wir haben aber vor dies noch in späteren Updates wieder hinzuzufügen.

Folgend ist der Levelordner aufbau:



Hierbei hat jeder Levelordner seine 3D Modell datei: .gltf & buffer.bin, sowie eine map.xml.

Hier einmal die map.xml:

```
<Map>
	<internalName>SimpleHeaven</internalName>
	<Name>Simple Heaven</Name>
```

```
        <Desc>Don't focus too much on the last part of the name</Desc>
        <Players>4</Players>
        <TimeAccelFactor>1.0</TimeAccelFactor>
</Map>
```

Hier gibt es 3 wichtige Einträge: „internalName“, „Name“ und „Desc“.

- InternalName: Name welcher genutzt wird um das level Eindeutig zu erkennen.
- Name: Name vom Level, kann Leerzeichen enthalten.
- Desc: Beschreibung vom Level, kann aus mehreren Leerzeichen und Newlines bestehen.
- Players: Ungenutzter Eintrag, welcher für die Mehrspielerkomponente ausgelegt war.
- TimeAccelFactor: Ungenutzter Eintrag, welcher für die Schwierigkeit ausgelegt war.

3.5.7. Die Erstellung einer Benutzerdefinierten Map

Falls ein Nutzer eine Map Erstellen möchte, muss er zuerst eine Karte in einem 3D Modellierungs Programm Erstellen. Wir nehmen folgende Karte als Beispiel:

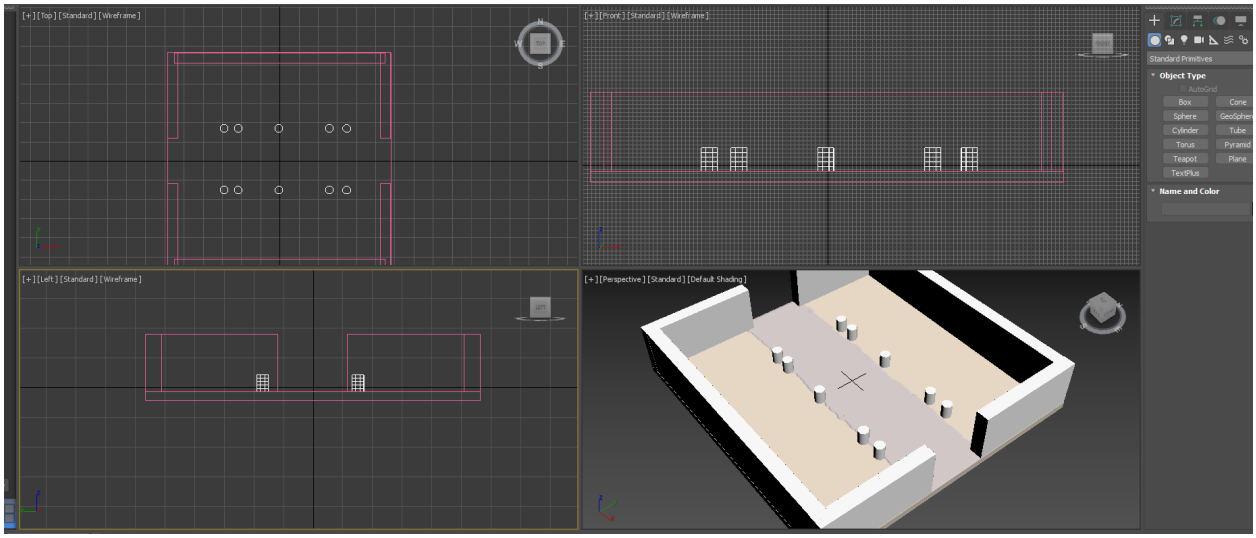


Schaubild 10: Simples Design für eine Spielkarte

Nachdem das Geometrische erstellt wurden ist, ist das Objektiv schwerste geschafft, was nun folgt sind Referenzpunkte, welche dem Spiel und der Engine sagen, wo sich was befinden sollte.

Referenzpunkte können wir einfach mit „Dummies“ Erstellen und sie positionieren wie normale Objekte.

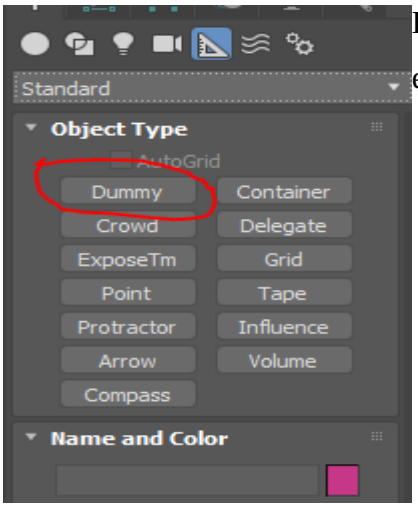


Schaubild 11: Dummy auswahl in 3DS Max

Es müssen folgende Referenzpunkte gesetzt werden, für jedes einzelne Level:

- RSpawnSP: Spieler Startposition
- RSpawnZ: Zombie Startposition
- RSpawnBOSS: Boss Startposition
- RSpawnAmmo: Munitionspäckchen Position

Folgend ist eine Fertige Map, welche Bereit zum Exportieren ist.
Bei der Exportierung der Karte gibt es auf folgendes acht zu geben:

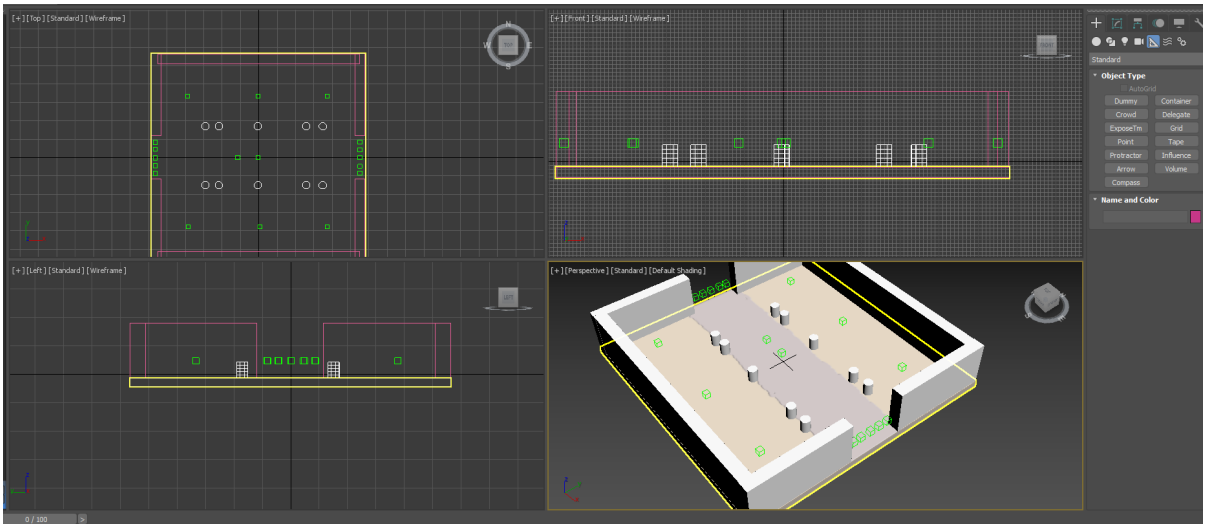


Schaubild 12: Eine Fertige Map

- Dateiformat MUSS .FBX (oder direkt GLTF) sein.
- Nach dem Abspeichern als .FBX muss es zu GLTF Konvertiert werden, mit einem Drittanbieter werkzeug wie FBXtoGLTF.
- Bei Animationen MUSS Animation-Baking aktiviert sein.
- Nach abspeicherung, müssen alle Texturen und die .gltf & .bin Datei in den Mod Levelordner enthalten sein

3.5.8. XML

XML ist ein Dateiformat für den Austausch von Daten zwischen Computern, welches wir für reine Konfigurationszwecke genutzt haben, da die Programmiersprache selber einen Leistungsstarken Parser für XML Dateien bereitstellt, und die Implementation somit einfach fällt.

XML Dateien sind Knoten basiert, und jeder Knoten kann zahlreiche Subknoten besitzen, hier ein Beispiel:

```
<Kunden>
	<Kunde name=“BBS-ME“>
		<Abgeschlossen>
			<Project name=“Router“>
				<Description>Das Routerprojekt der BFIA18</Description>
			</Project>
		</Abgeschlossen>
	</Kunde>
</Kunden>
```

XML Dateien sind typsicher, da man eine zweite Datei als „XSD Definitions“ Datei festlegen kann, welche so aussehen würde:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Kunden">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Kunde">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Abgeschlossen">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Project">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Description" type="xs:string" />
                        </xs:sequence>
                        <xs:attribute name="name" type="xs:string" use="required" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Diese Dateien werden allerdings meist nur genutzt, falls die Typsicherheit zu 100% garantiert werden muss, da dies extra Arbeit für den Parser ist, und ihn deswegen auch langsamer macht.

Im Projekt QubeShoot wurde keine XSD Definitionsdatei genutzt, da wir XML nur für die Konfiguration nutzen, und keine größeren Erweiterten features brauchten.

3.5.9. Unity Komponenten: GameObjects

Game Objects sind in Unity jedes Objekt, welches in der Szene im Spiel auftaucht. Dies kann ein unsichtbares Skript sein, oder gleich ein ganzes Modell mit mehreren Skripten welche verankert sind. Wir nehmen hierfür die Kamera der Szene als Beispiel:

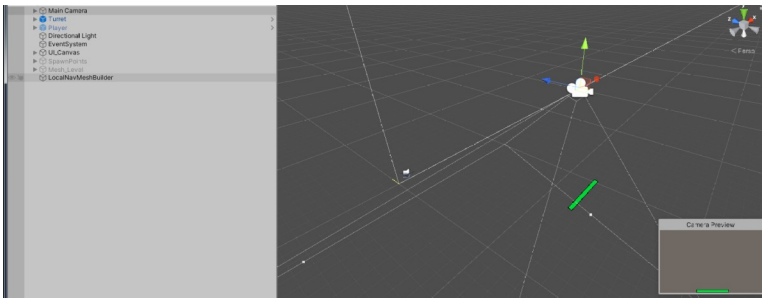


Schaubild 13: Die Spiel-Kamera als GameObject

Anhand der „Main Camera“, kann man hierbei auch die Komponenten ansehen, welche an dem Game Object „Main Camera“ verankert sind.

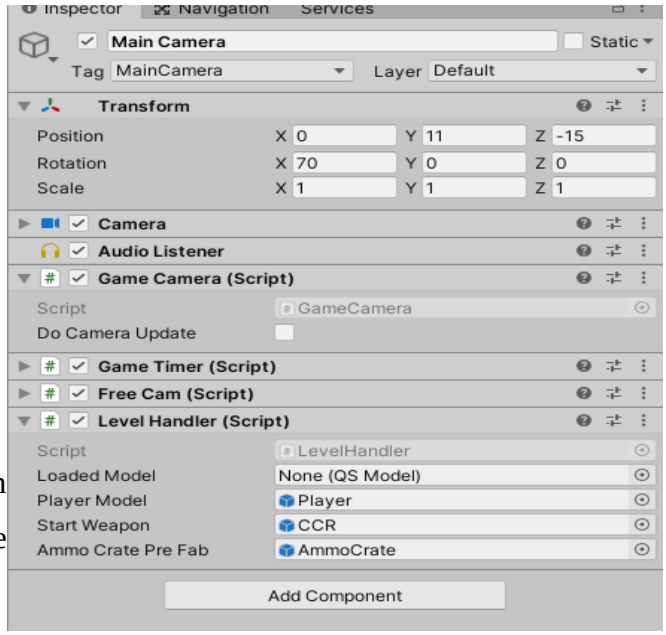


Schaubild 14: Unity Game Object: Main Camera

Hierbei haben wir an die Kamera, Skripte wie „Game Camera“, „Game Timer“, „Free Cam“ und den „Level Handler“ hinzugefügt, aus dem grund, weil die Kamera einer der fundamentalen Game Objects ist, welche wirklich immer da sind, ist.

Natürlich kann man „Game Objects“ auch im Code selber Herstellen und hinzufügen, wie folgt:

```
// Create the Player
private void CreatePlayer()
{
    GameObject newPlayer = Instantiate(
          playerModel,
          new Vector3(0, 0, 0),
          Quaternion.identity
   ) as GameObject;
}
```

Das wichtige schlagwort hierbei ist „Instantiate“, damit kann man Unity und noch wichtiger, der C++ unterliegenen internen Engine sagen das eine Neue komponente zum Spiel sofort hinzugefügt werden soll, mittels eines Bereits angefertigten GameObjects, an einer Vector3 Position, und einer Ausrichtung mithilfe eines Quaternions.

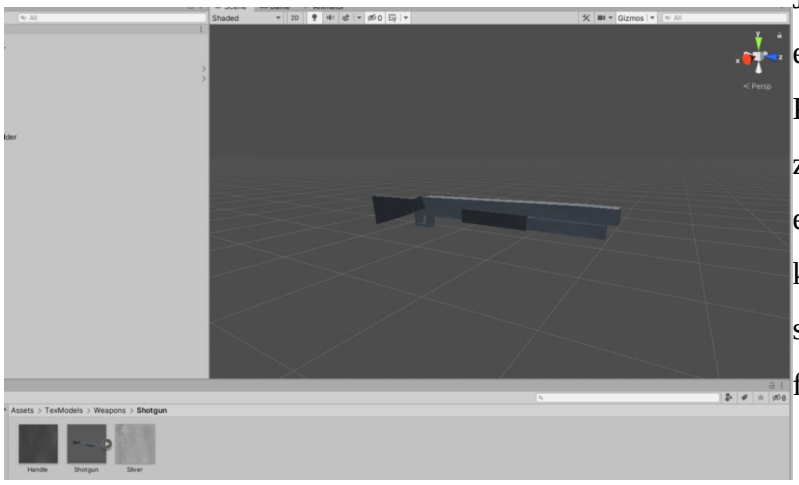
In dem fall kann man aber kein komplett neues GameObject einfach so Erstellen, dies ist einfach nur die Instanzierung von neuen GameObjects, mithilfe eines vorerstellten „PreFab“.

3.5.9.1 PreFabs

PreFabs sind in Unity vorerstellte Objekte von jeglicher Sorte. Es kann sich um einfache Skripte handeln, oder um Komplexe 3D Modelle mit voreingestellten Skripten und mehr.

Hierbei wird erklärt wie ein PreFab erstellt wird, und wie es genutzt werden kann, und auch was die Vorteile von PreFabs sind.

Für die PreFab Erstellung wird angenommen, dass man eine neue Waffe ins Spiel bringen möchte. Der Prozess beginnt damit das man ein 3D Modell in die Spiel-Szene reinzieht per Drag&Drop.



Jetzt kann man anfangen, dieses einfache 3D Modell mit Komponenten auszurüsten, wie z.B: Ein „BoxCollider“, welcher zu einem Trigger genutzt werden kann, um ein Skript auszuführen, sobald die Waffen auf den Boden fällt.

Schaubild 15: Eine Waffe in der Szene

Dies geschieht über „Add Component“ – „Box Collider“. Nach dem hinzufügen wird eine Box um die Waffe gezeigt, das ist die Grenze des Box Colliders. Wir werden einen Haken in „Is Trigger“ setzen, damit eine spezielle Funktion ausgeführt wird, sobald etwas innerhalb dieser Grenzen auftaucht.

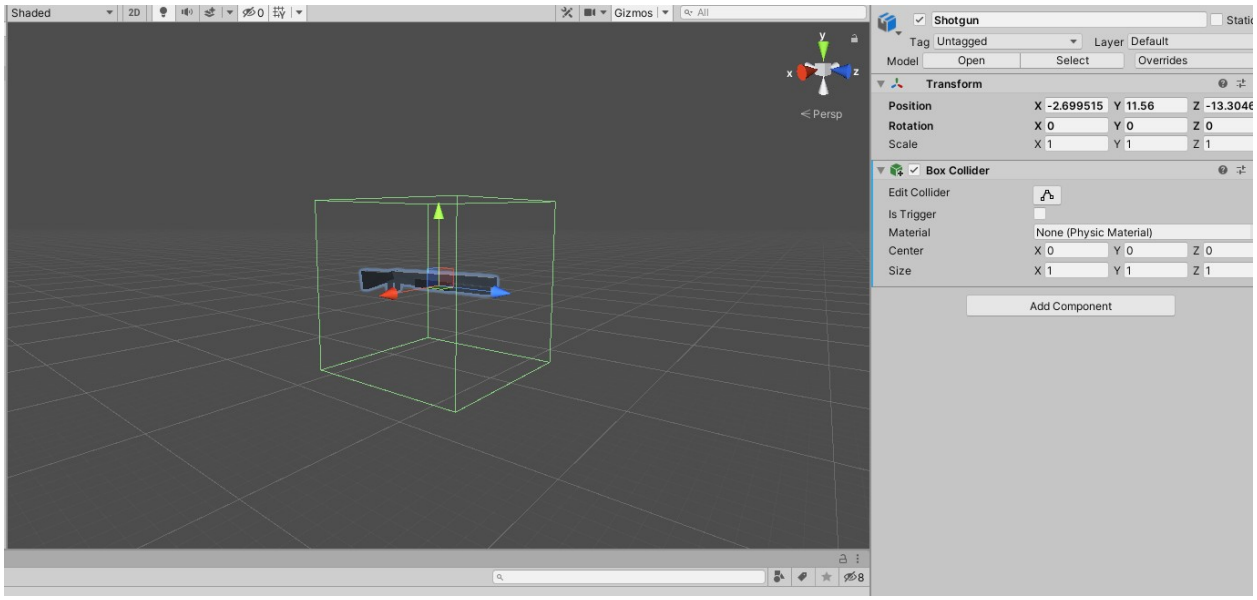


Schaubild 16: Waffe mit BoxCollider

Als nächstes muss man ein Skript einbinden, in welchen man die spezielle oben genannte Funktion definiert. Hierbei wird per „Add Component“ – „New Script“ eine neue Funktion namens „WaffenDrop“ hinzugefügt.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
    0 Verweise
5   public class WaffenDrop : MonoBehaviour
6   {
7       // Start is called before the first frame update
        0 Verweise
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
        0 Verweise
14      void Update()
15      {
16
17      }
18  }
19
```

Schaubild 17: Das WaffenDrop.cs Skript

So schaut jede von Unity Erstellte Skriptdatei anfangs aus, sie bietet die 2 Wichtigsten Funktionen von anfang an. „Start“ und „Update“.

- Start(): Diese Funktion wird beim erstmaligen Erstellen des Objektes (an dem dieses Skript angehängt ist) ausgeführt.
- Update(): Diese Funktion wird nach jedem einzelnen Frame ausgeführt.

Diese Beiden funktionen reichen uns nicht, wir brauchen eine neue namens „OnCollisionEnter“:

```
private void OnCollisionEnter(Collision collision)
{

}
```

Diese Funktion funktioniert nur wenn wir einen Box Collider oder ähnliches haben, welches andere „Collider“ erkennen kann. Um die Funktion jetzt dafür zu nutzen die Waffe zu zerstören sobald sie den Boden erreicht hat, muss man folgende änderungen unternehmen:

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.transform.name == "Ground")
        Destroy(gameObject);
}
```

Jedes GameObject in Unity ist meistens an einen „Transform“ gebunden, welches die Position, Rotation und Skalierung darstellt, meistens auch von einem 3D Modell. In dem fall hat der Boden in jedem Level „Ground“ zu heißen, und deshalb kann man den namen des Transforms einfach auf „Ground“ überprüfen. Dies würde auch über Koordinatenprüfung funktionieren, allerdings wurde hier aus Beispiel gehandelt.

Um dies Waffe jetzt von überall aus zu Nutzen, auch in anderen Unity Projekten, kann man, und sollte man dieses GameObject als PreFab speichern. Dafür Drag&Dropped man das GameObject aus der Linken Projekt Hierarchy Liste, runter zum Projektordner „PreFabs“.

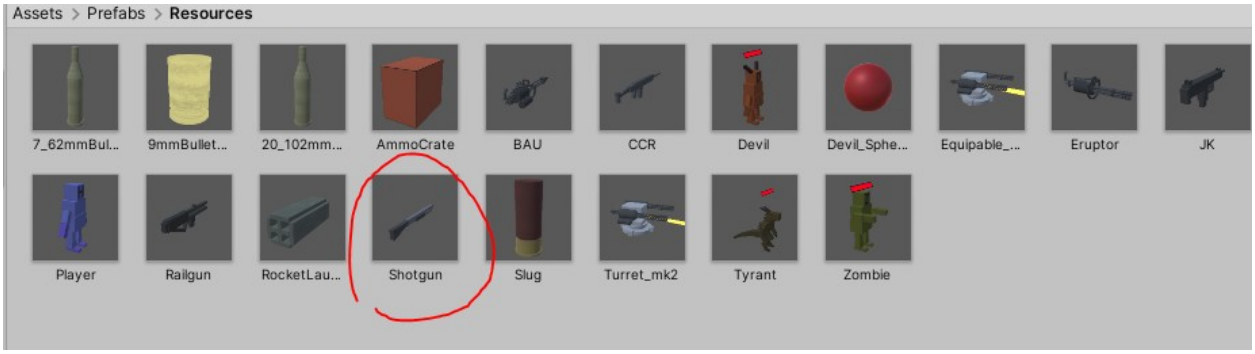


Schaubild 18: Shotgun PreFab

Nun kann man dieses PreFab beliebig per Skript instantieren, oder einzelnt in die Szene Droppen. Natürlich kann man das PreFab auch beliebig ändern und anpassen.

3.5.9.2 Spiele Funktion

Hier wird der *Essenzielle* Spiele Code dokumentiert, gezeigt und erklärt.

Zu beachten ist dass nur selten echte Funktionen gezeigt werden.

3.5.9.2.1 Initialisierung

Bei der Initialisierung des Spieles, wird erstmals eine bestimmte funktion namens „OnBeforeSceneLoadRuntimeMethod“ im Initialization.cs Skript ausgeführt.

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void OnBeforeSceneLoadRuntimeMethod()
{
    // Startup Cleanup
    Logger.PrepareLOG();

    // Load Mod Data
    ModLoader modLoader = new ModLoader();
    modLoader.LoadAllMods();

    Debug.Log("== Startup Complete ==");
    Debug.Log(System.Environment.CurrentDirectory);
}
```

Hierbei werden noch kurzerhand funktionen ausgeführt, bevor die Erste Szene im Spiel geladen wird. Dies hat den vorteil das man noch Externe sachen Laden kann, wie z.B.: Konfigurationsdateien, 3D Modelle, oder andere Dateien, und die nach dem laden nutzen kann.

Funktion:

- Das vorbereiten der LOG Datei (Alte wird gelöscht, und eine neue wird Erstellt)
- Der ModLoader wird instanziert und es werden alle Mods geladen.

3.5.9.2.2 Modloader: Level-Lade Skript

```
    // Load mod's Level
    private void LoadModLevel(string levelFolder, int modIndex)
    {
        Logger.LOG("[ModLoader]: Adding Level - " + levelFolder);
        XMLHandler xh = new XMLHandler(levelFolder + "\\map.xml");
        ModLevel newLevel = new ModLevel();
        newLevel.filePath = levelFolder + "\\";

        // General
        newLevel.internalName = xh.ReadXMLSingle("internalName");
        newLevel.name = xh.ReadXMLSingle("Name");
        newLevel.description = xh.ReadXMLSingle("Desc");
        newLevel.players = int.Parse(xh.ReadXMLSingle("Players"));
        newLevel.timeAccelerationFactor =
float.Parse(xh.ReadXMLSingle("TimeAccelFactor"), CultureInfo.InvariantCulture);

        // -- Finish
        ModCache.mods[modIndex].GetLevels().Add(newLevel);
    }
```

Funktion:

- Instanzierung des XMLHandlers für das Parsen der map.xml
- Hinzufügen der relevanten Daten zum Globalen Mod Cache.

3.5.9.2.3 LevelHandler: Level-Modell-Lade Skript

```
    // Load Level by Name
    public void LoadLevel(string levelName)
    {
        Logger.LOG("[LevelHandler]: Loading Level: " + levelName);
        load_timer.Start();
        this.loadedLevel = GetModLevelByName(levelName);
        Constants.gameTimer = GameObject.Find("Main Camera").GetComponent<GameTimer>();

        // Generate the map
        CreateMap(loadedLevel);

        // Finish
        load_timer.Stop();
        Logger.LOG("[LevelHandler]: took ~ " + load_timer.ElapsedMilliseconds + " ms!");
    }
```

Funktion:

- Das Laden einer Map anhand des „Namen“ der Map, fest definiert in der map.xml
- Das merken der Dauer des Ladevorgangs per StopWatch.
- Das festsetzen des Globalen GameTimers zu dem, der in der main Camera verankert ist.
- Das Erstellen der Map anhand laden des 3D Modells, sowie Erstellung von NavMesh Tags.

3.5.9.2.4 Spieler Controller

Der Spieler Controller regelt die Bewegung des Spielers, und ist somit eine unübersehbare Komponente des Spieles.

Hierbei gibt es einige Variablen, die zur Bewegung beitragen:

```
// Movement Variables
public float rotationSpeed = 950;
public float walkSpeed = 8;
public float runSpeed = 12;
private float acceleration = 5;
```

Es wird alles bis auf acceleration genutzt, da dies ein ausgelassenes Feature ist.

- RotationSpeed: Ausrichtungs geschwindigkeit, wie schnell man sich umschauen kann.
- WalkSpeed: Geh Geschwindigkeit
- RunSpeed: Renn Geschwindigkeit (Togglebar über LSHIFT)

3.5.9.2.5 Weapon Controller

Der Weapon Controller regelnt welche Waffen der Spieler zur verfügung hat, und ob er sie Freigeschaltet hat.

Wichtige Variablen:

```
private int weaponIndex = 0;
public GameObject selectedWeapon;
public GameObject instanceWeapon;
```

- weaponIndex: Die Index Nummer der aktuell ausgerüsteten Waffe.
- SelectedWeapon: Die aktuell ausgewählte Waffe (welche der Spieler am start hat)
- InstanceWeapon: Die aktuell instanzierte Waffen (welche Temporär ist)

3.5.9.2.6 Weapon

Das Weapon Script ist das sogenannte Herz jedes Waffen PreFab‘s.

Es regelnt alles von der RPM bis zu der maximalen Anzahl an womöglich durchgeschossenen Monstern.

Wichtige Variablen:

- wname: Der name der Waffe.
- Highscore: Was für einen Highscore man benötigt um diese Waffe freizuschalten.
- Waffen ID: Wird dafür genutzt um die Passende Animation für das halten der Waffe abzuspielen.
- Damage: Der gemachte Schaden der Waffe.
- AmmoCurrent: Der „derzeitige“ stand der Munition.
- AmmoDefault: Der anfangs stand der Munition.
- AmmoMax: Die Maximale anzahl an Munition.
- WeaponType: Der Schuss Waffentyp (Einzeln, Mehrfachschuss, Vollautomatisch).
- projectileType: Der Projektiltyp der Waffe (HitScan oder Projektil).
- roundsPerMinute: Schusszahl pro Minute.
- hasTracer: Ob die Waffe einen Tracer zeichen soll.
- TracerLifeTime: Lebenszeit des Tracers in Sekunden.
- TracerBeginWidth: Anfangsbreite des Tracers.
- TracerEndWidth: Endbreite des Tracers.
- HasSmoke: Ob die Waffe rauch an der Flinte zeichen soll.
- HasShellEject: Ob die Waffe leeere Kugeln droppen sollen.
- DoPierceAmount: Durch wie viele Objekte die Waffe durchschießen kann
- DoMultiShotAmount: Wie viele mehrere Geschosse die Waffe pro Schuss abgeben soll.
- IsPlaceable: Ob diese „Waffe“ ein Platzierbarer gegenstand ist, z.B.: Selbstschussanlage.
- PlacementSize: Wie größ das zu platzierende Objekt ist.
- PlacedPrefab: Welches PreFab für das Platzieren genutzt werden soll.
- ProjectileForce: Wie schnell und mit wie viel Kraft das geschossene Projektil vorwärts fliegt.
- ProjectilePrefab: Was für ein PreFab für das Geschoss genutzt wird.
- ProjectileDoAOEDamage: Ob das Projektil AOE Schaden machen soll.
- projectileAOEDamage: Der AOE Schaden.
- ProjectileAOERange: Die AOE Reichweite.
- ProjectileExplosion: Ob das Projektil beim Erwischen eines Objektes explodieren soll.
- ProjectileLifeTime: Nach welcher Zeit dieses Projektil aus dem Spiel verschwinden soll.

3.5.9.2.7 `ZombieAttack Skript`

Das ZombieAttack Skript wird jedem Zombie und jedem „Tyrant“ hinzugefügt, es ist das Skript welches die Monster angreifen lässt.

Wictigte Variablen:

```
public Transform player;
private Transform zombie;

public float attackRange = 20;
public float attackSpeedPerMin = 60;
private int zombieDamage = 5;
private float zombieHealth = 100.0f;
```

- player: Der Transform des Spielers
- zombie: Der Transform dieses Zombies
- attackRange: Angriffsreichweite
- attackSpeedPerMin: Angriffe pro Minute
- zombieDamage: Schaden

Folgend ist der Angriffs Enumerator:

```
private IEnumerator IsInRange() {
    if (Vector3.Distance(player.position, zombie.position) < attackRange) {
        canCallFunction = false;

        // Anim
        GetComponent<Animator>().SetTrigger("OnAttack");

        GameObject.Find("Main Camera/PlayerHealthBar").
          GetComponent<TakeDamage>().TakeZombieDamage(zombieDamage);

        yield return new WaitForSeconds(60 / attackSpeedPerMin);


        canCallFunction = true;
    }
}
```

Hierbei wird überprüft ob der Spieler in Reichweite ist, und falls er es ist, wird die „TakeDamage“ funktion vom Spieler aufgerufen.

Bei yield wird gesagt dass die Engine eine bestimmte anzahl an sekunden warten soll, bevor der untere Code ausgeführt werden soll.

CanCallFunction stellst sicher, dass diese Funktion nicht aufgerufen wird solange ein Angriff bereits stattfindet.

3.5.9.2.7 `DevilAttack Skript`

Das DevilAttack Skript wird von Devils genutzt. Es stellt folgendes sicher:

- Das Spieler in Reichweite sind.
- Das Spieler in Sichtweite sind.

Die variablen sind identisch mit dem ZombieAttack Skript, bis auf folgend neue:

```
public Rigidbody projectile;
public Transform projectileSpawn;
```

- projectile: Das Projektil PreFab zum instanzieren
- projectileSpawn: Die Position an welcher stelle das PreFab auftauchen soll.

Die folgende Funktion stellt sicher, dass Direkte sichtweite zwischen dem Devil und dem Spieler besteht.

```
private bool IsInLineOfSight() {
    // Raycast
    Ray ray = new Ray(projectileSpawn.position, projectileSpawn.forward);
    RaycastHit hit;

    // Raycast Events
    if (Physics.Raycast(ray, out hit, attackRange)) {
        Transform target = hit.transform;
        if (target != null) {
            if (target.tag == "Player")
                return true;
        }
    }
    return false;
}
```

Die folgende Funktion stellt sicher das der Spieler in Reichweite ist, und in Sichtweite. Falls beides zutreffen sollte, wird ein Energiekugel Instanziert.

```
private IEnumerator IsInRange() {
    if (Vector3.Distance(player.position, devil.position) < attackRange) {
        if (IsInLineOfSight()) {
            isAttacking = true;
            Rigidbody clone;
clone = Instantiate(projectile, projectileSpawn.position, projectile.rotation);
clone.velocity = gameObject.transform.TransformDirection(Vector3.forward *
projectileSpeed);
            clone.GetComponent<DevilSphere>().damage = damage;

            yield return new WaitForSeconds(60 / attackSpeedPerMin);
            isAttacking = false;
        }
    }
}
```

3.5.9.2.8 Multiplayer

Der Multiplayer wurde in der Erweiterten Version hinzugefügt. Hier wird erklärt wie der Multiplayer funktioniert, und aussieht.

3.5.9.2.8.1: MP: Synchronisation

Im Multiplayer hängt alles von der Synchronisation verschiedener Clienten ab, alles andere ist Optional. Dies wurde durch die Unity Networking Library „Mirror“ realisiert, da die Interne Library Bereits als „Deprecated“ gilt, und daher obsolete ist.

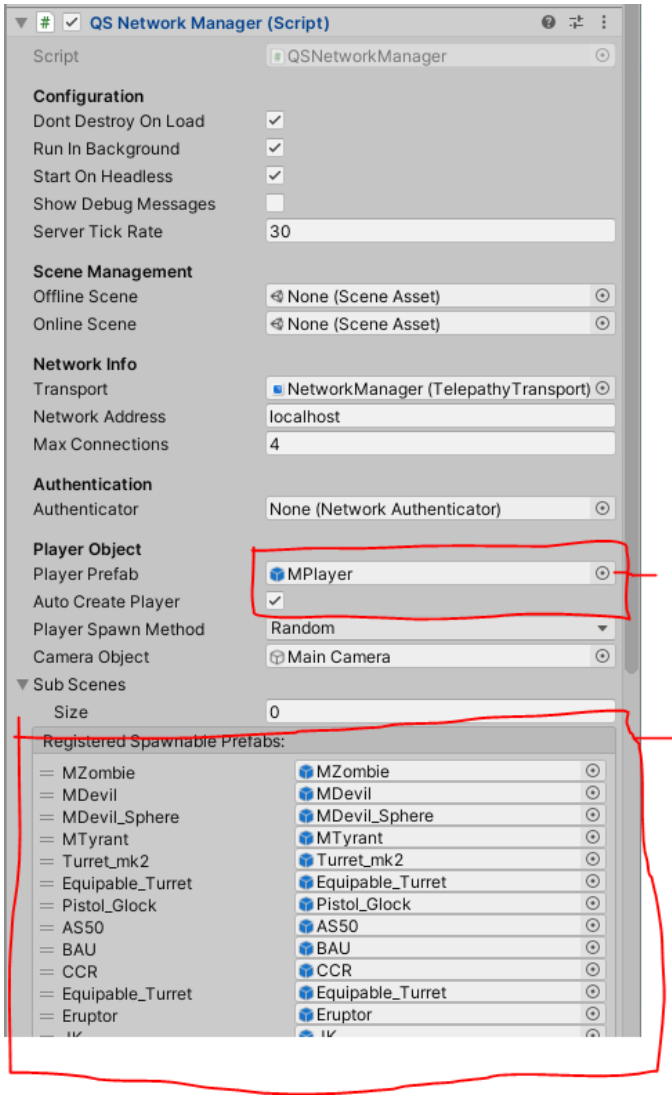


Schaubild 19: QubeShoot Netzwerk Manager

Beispiel: Nehme man einen Spieler, welcher sich Bewegen kann. Für das Instanzieren von einem Spieler, gibt Mirror eine Besondere Option: Das manuelle Auswählen eines PreFab‘s welcher Instanziert werden soll, falls ein Client dem Server Beitritt. Siehe Highlight #1.

In der Regel müssen alle möglichen Instanzierbaren PreFabs Registriert werden. Siehe Highlight #2.

Darüberhinaus erlaubt es die Mirror Library das einstellen der Tickrate, die Tickrate besagt wie oft der Server geänderte Objekte synchronisieren soll. 64 Tickrate ist eine Sekunde.

Das heißt eine Tickrate von 30 bedeutet das die Kalkulationen 30 mal innerhalb einer Sekunde durchgeführt werden.

3.5.9.2.8.2: MP: Commands, ClientRPC

Für das Synchronisieren brauchen alle Clients die möglichkei Befehle zum Server zu schicken, und auch vom Server zurückgegebene Befehle zu verarbeiten.

Commands sind Methoden welche auf dem Server ausgeführt werden, aber von Clients beliebig gerufen werden können.

ClientRPCs werden auf den Clients ausgeführt, wenn der Server nach einem Command dies implementiert hat.
Folgend ein Beispiel für eine Simple Kommunikation:

```
    // Equip a new Weapon
    // [ClientSide] EquipWeapon: Sender
    // Event: (Client)->(Server)
    [Client]
    public void EquipWeapon(int index)
    {
        if (!isLocalPlayer)
            return;

        DeEquipWeapon();

        // Get Weapon
        GameObject go_weapon = GetWeaponByIndex(index);
        Transform newWeaponREF = Common.GetModelReferencePath(gameObject,
"Bone_TopArm_R/Bone_BotArm_R/Bone_Hand_R/RWeapon").transform;
        Constants.weaponEquipped = go_weapon;

        Cmd_EquipWeapon(newWeaponREF.position, gameObject.transform.rotation * go_weapon.transform.rotation,
index, gameObject);
    }

    // [ServerSide] EquipWeapon: Receiver
    // Event: (Server)<-(Client)
    [Command]
private void Cmd_EquipWeapon(Vector3 position, Quaternion rotation, int id_WepToBeInstanced, GameObject parent)
    {
GameObject go_ReferencedWeapon = parent.GetComponent<WeaponController>().GetWeaponByIndex(id_WepToBeInstanced);

        instanceWeapon = Instantiate(go_ReferencedWeapon, position, rotation) as GameObject;
        NetworkServer.Spawn(instanceWeapon);
        instanceWeapon.GetComponent<NetworkIdentity>().AssignClientAuthority(connectionToClient);

        Rpc_EquipWeapon(position, rotation, instanceWeapon, parent);
    }

    // [ClientSide]: EquipWeapon: Reactor
    // Event.: (Client)<-(Server)
    [ClientRpc]
    private void Rpc_EquipWeapon(Vector3 localPos, Quaternion localRot, GameObject go_instancedObject,
GameObject go_parentObject)
    {
        go_instancedObject.transform.localPosition = localPos;
        go_instancedObject.transform.localRotation = localRot;
        go_instancedObject.transform.SetParent(go_parentObject.transform);

        // Assign Arms Animation
        gameObject.GetComponent<Animator>().SetFloat("Weapon ID",
         go_instancedObject.GetComponent<Weapon>().gunID); // Set Anim ID

        // Assign Ammo
        Weapon weaponScript = go_instancedObject.GetComponent<Weapon>();
        weaponScript.ammoCurrent = TryGetAmmo(weaponScript);

        // Assign GameObject
        instanceWeapon = go_instancedObject;
    }
```

Im vorherig gezeigtem Code, versucht eine Methode auf einem Client, „EquipWeapon()“ eine Methode auf der Serverseite auszuführen mittels Cmd_EquipWeapon()“, welcher dann weiteren Code ausführt, und letztlich an alle Clients einen Remote-procedure-Call schickt, über „Rpc_EquipWeapon()“.

Hier einmal eine Veranschaulichung dieses Systems über ein UML Diagramm.

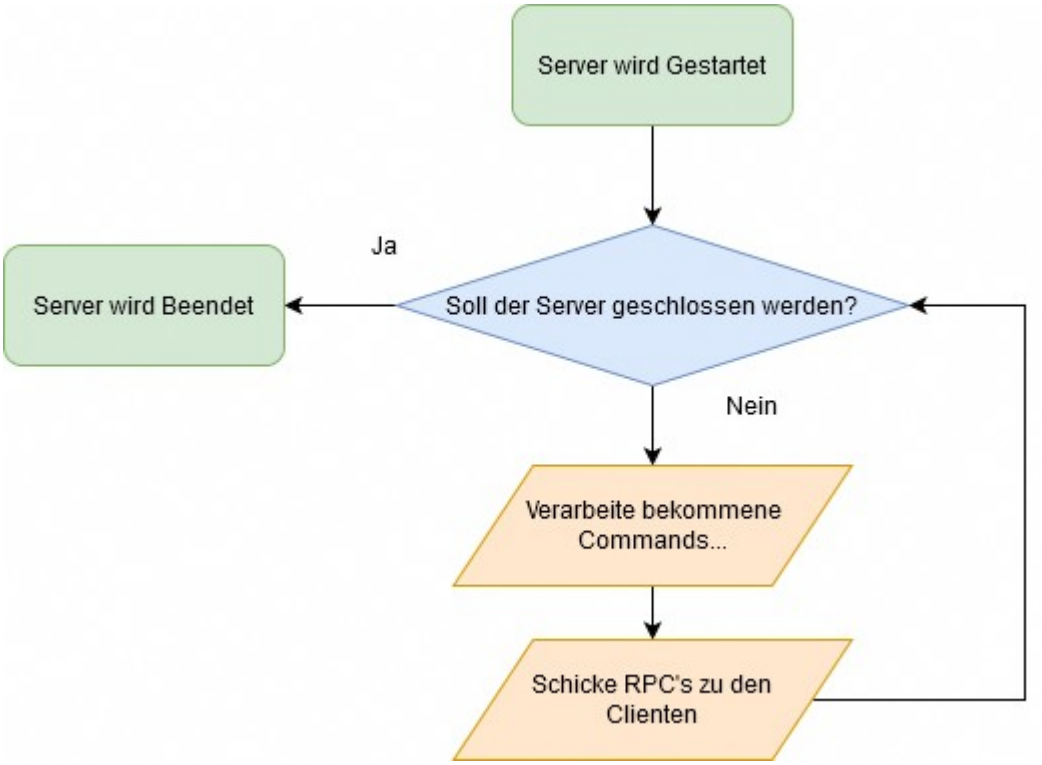


Schaubild 20: Server UML

3.5.9.2.8.3: MP: Lobby-System

Das Lobbysystem erlaubt es, mehreren Clienten sich gegenseitig einfacher zufinden. Jeder Client welcher einen Server erstellt, sendet automatisch einem „Masterserver“ welcher von uns gehostet wird Informationen wie z.B.: Server IPv4, Server Passwort, Server Name… etc

Somit muss kein Client mehr einem anderen Client seine IPv4 schicken, da dies vom Masterserver verarbeitet wird, und in einer Liste angezeigt wird.

Der MasterServer arbeitet mit einer MySQL Datenbank, und einer PHP API, welche es dem Spiel erlaubt mit der Datenbank zu Kommunizieren, indirekt.

Für das funktionierende Lobbysystem gibt es folgende PHP APIs für Events im Spiel:

- OnPlayerJoinedServer
- OnPlayerDisconnectServer
- OnRoomOpened
- OnRoomClosed

Mithilfe dieser Client-seitigen Methoden im Spiel, updaten wir Regelmäßig die Master-Server-Liste. Somit weiß der Server welche „Räume“ noch Platz für weitere Clients haben, und welche nicht. Welche mit einem Passwort versehen sind, und welche nicht.

Darüberhinaus wurde ein „Timeout-Level“ System implementiert, welches einen Server aus der Datenbank löscht, falls dieser ein bestimmtes Timeout Level erreicht. Jede 5 Minuten schickt der Client-Server einen Heartbeat an den Master-Server, falls der Master-Server diesen nicht bekommt, wird der Client-Server aus der Liste gelöscht, und folglich auch aus der Datenbank.

3.5.9.2 Spiel Akteure & Inhalte

Hier werden alle relevanten Spiel Akteure und Gegenstände gezeigt.

3.5.9.2.1 `Spiel Akteur: Zombie`

Der einfache „Zombie“ wird sofort nach Instanzierung zum Spieler laufen, und wird versuchen den Spieler zu „Schlagen“ sofern er in Reichtweite ist.

Daten	Werte
Trefferpunkte	100
Schaden	5
Angriffe pro Minute	60
Animationsgeschwindigkeit	0
Punkte fürs Besiegen	20
Lebenszeit nach dem Tot (Sek.)	2
Angriffsreichweite	1
Angriffsskript	0
Animationsskript	**ZombieAnimator**

Animationen	Enthalten
Laufen	Ja
Angreifen	Ja
Sterben	Ja
Erscheinung	**Nein**

Schaubild 21: Ein Zombie

Schaubild 22: Ein Zombie im Spiel

3.5.9.2.2 Spiel Akteur: Devil

Der „Devil“ wird nach der Erscheinung sofort zum Spieler laufen, ähnlich wie der Zombie, allerdings wird er ihnt nicht „Schlagen“, sondern er wird Energiekugeln zum Spieler werfen, die erheblichen Schaden machen.

Daten	Werte
Trefferpunkte	400
Schaden	20
Angriffe pro Minute	120
Animationsgeschwindigkeit	0
Punkte fürs Besiegen	100
Lebenszeit nach dem Tot (Sek.)	2
Angriffsreichweite	100
Angriffsskript	DevilAttack.cs
Animationsskript	**DevilAnimator**

Animationen	Enthalten
Laufen	Ja
Angreifen	Ja
Sterben	Ja
Erscheinung	**Nein**

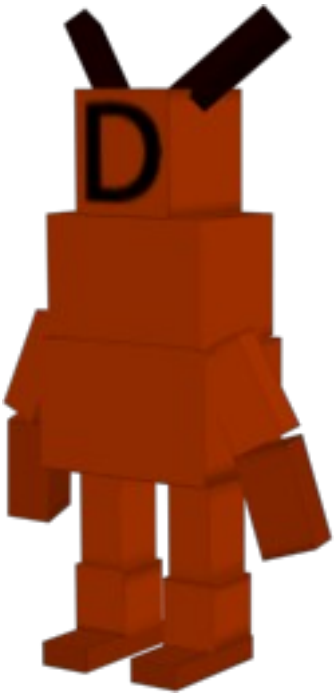


Schaubild 23: Ein Devil

Schaubild 24: Ein Devil im Spiel

3.5.9.2.3 Spiel Akteur: Tyrant AkA „Lucy“

Der „Tyrant“ oder auch „Lucy“ genannt (intern) ist der Boss des Spieles. Er hat wesentlich mehr Trefferpunkte und macht enormen Schaden.

Daten	Werte
Trefferpunkte	4000
Schaden	45
Angriffe pro Minute	60
Animationsgeschwindigkeit	0
Punkte fürs Besiegen	1000
Lebenszeit nach dem Tot (Sek.)	5
Angriffsreichweite	2
Angriffsskript	ZombieAttack.cs
Animationsskript	**TyrantAnimator**

Animationen	Enthalten
Laufen	Ja
Angreifen	Ja
Sterben	Ja
Erscheinung	**Nein**

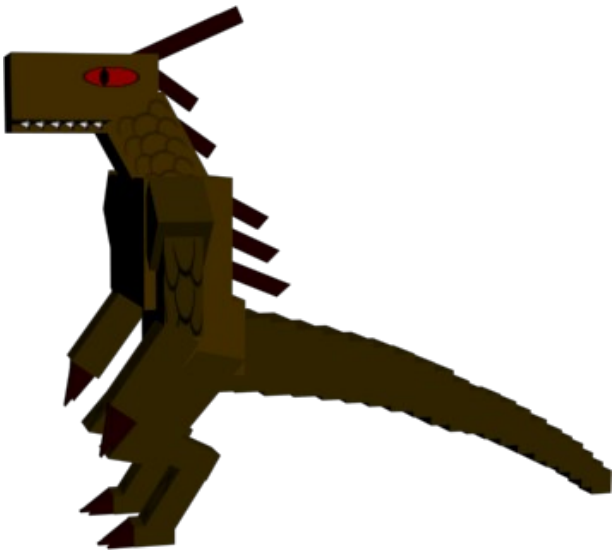


Schaubild 25: Der "Tyrant"

Schaubild 26: Der „Tyrant“ im Spiel

3.5.9.2.4 Spiel Akteur: Turret

Der Geschützturm wird von dem Spieler an einer komplett auswählbaren Position platziert, und nach der Platzierung wird die Selbstschussanlage auf den nächstnahen Feind schießen. Hierbei wird zwischen 2 Turrets unterschieden: Der Geschützturm welcher in der „Hand“ des Spielers ist, und der echte.

Das Geschützturm enthält eine leicht abgeänderte Version des Weapon Skripts, d.h.: Es können beliebige Kombinationen an Waffen an ein Geschützturm angebracht werden (In der Entwicklungsumgebung), das Modell hat eine feste Waffe. Welche man manuell ändern müsste.

Daten	Werte
Waffentyp	Vollautomatisch (Hitscan)
Schüsse pro Minute	1200
Schaden	100
Reichweite	20

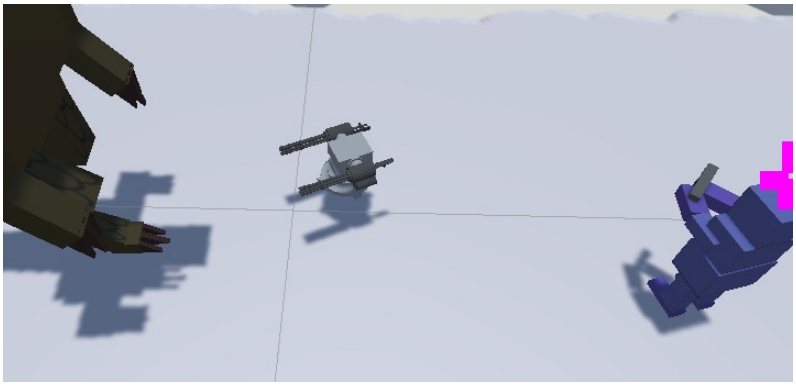


Schaubild 28: Ein Geschützturm in dem Spiel

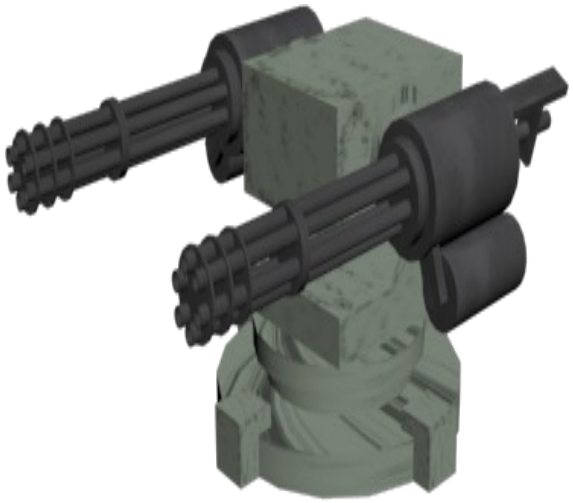


Schaubild 27: Ein Geschützturm

3.5.9.2.4 Spiel Akteur: Rocket Launcher Turret

Dies ist eine andere Variante des normalen Geschützturms, mit einem Raketenwerfer an der seite befestigt, anstatt der 2 Maschinengewehre. Dieser Geschützturm zeichnet sich durch seine längere Reichweite und des AOE Schadens aus.

Daten	Werte
Waffentyp	Semi (Projektil)
Schüsse pro Minute	20
Schaden	200 (AOE)
Reichweite	40

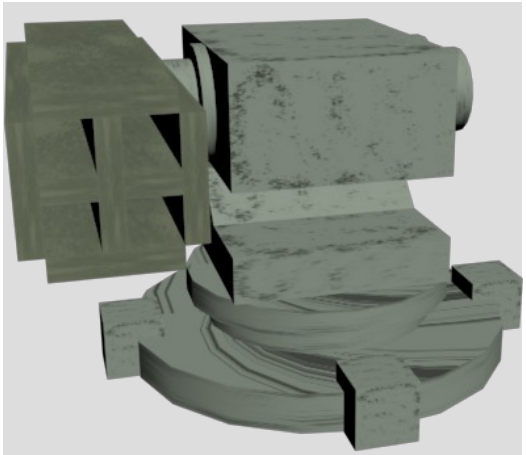


Schaubild 29: Ein Rocket Launcher Turret

3.5.9.2.5 Spiel Waffen: Pistole

Die Pistole ist die erste Waffe die der Spieler bekommt, und damit auch die schwächste.
Inspiration war hierbei die Glock 19.

Daten	Werte
Waffentyp	Semi (Hitscan)
Schüsse pro Minute	600
Schaden	40
Projektiltyp	Hitscan
Munition Maximum	360
Durchschlagsanzahl	1
Schussanzahl	1
Highscore Benötigt	0

Schaubild 30: Die Q.S: Pistole

3.5.9.2.6 Spiel Waffen: Uzi

Die Uzi ist die zweite Waffe die der Spieler bekommt.
Inspiration war die „Uzi Gal“.

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	600
Schaden	55
Projektiltyp	Hitscan
Munition Maximum	3200
Durchschlagsanzahl	1
Schussanzahl	1
Highscore Benötigt	500

Schaubild 31: Die Q.S: Uzi

3.5.9.2.7 Spiel Waffen: Schrotflinte

Die Shotgun ist die dritte Waffe die der Spieler bekommt.
Inspiration war die „Benelli Super 90-M4“

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	90
Schaden	80
Projektiltyp	Hitscan
Munition Maximum	600
Durchschlagsanzahl	2
Schussanzahl	5
Highscore Benötigt	13000

Schaubild 32: Die Q.S: Shotgun

3.5.9.2.8 Spiel Waffen: CCR

Die CCR ist die vierte Waffe die der Spieler bekommt.
Inspiration war die „Bushmaster ACR“

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	1000
Schaden	180
Projektiltyp	Hitscan
Munition Maximum	1000
Durchschlagsanzahl	2
Schussanzahl	1
Highscore Benötigt	2000

Schaubild 33: Die Q.S: CCR

3.5.9.2.9 Spiel Waffen: Railgun

Die Railgun ist die fünfte Waffe die der Spieler bekommt.
Inspiration war hierbei Concept Art für „Pantropy“

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	300
Schaden	300
Projektiltyp	Hitscan
Munition Maximum	120
Durchschlagsanzahl	7
Schussanzahl	1
Highscore Benötigt	40000

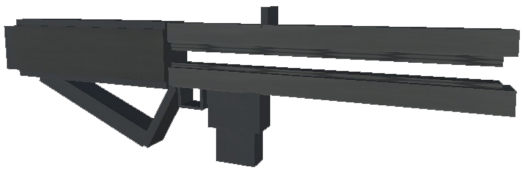


Schaubild 34: Die Q.S: Railgun

3.5.9.2.9.1 Spiel Waffen: Eruptor

Die Eruptor ist die sechste Waffe die der Spieler bekommt.
Inspiration war hierbei die „M61 Vulcan“

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	4000
Schaden	90
Projektiltyp	Hitscan
Munition Maximum	12000
Durchschlagsanzahl	2
Schussanzahl	1
Highscore Benötigt	100000

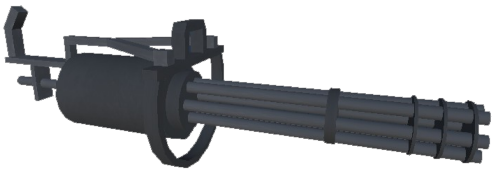


Schaubild 35: Die Q.S: B.A.U

3.5.9.2.9.2 Spiel Waffen: B.A.U

Die B.A.U ist die siebte Waffe die der Spieler bekommt.
Inspiration war hierbei die „GAU-19“

Daten	Werte
Waffentyp	Auto (Hitscan)
Schüsse pro Minute	6000
Schaden	120
Projektiltyp	Hitscan
Munition Maximum	36000
Durchschlagsanzahl	3
Schussanzahl	1
Highscore Benötigt	200000

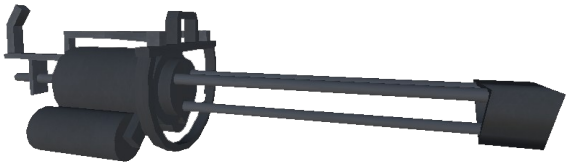


Schaubild 36: Die Q.S. B.A.U

3.5.9.2.9.2 Spiel Waffen: XA-1000

Die B.A.U ist die siebte Waffe die der Spieler bekommt.
Inspiration war hierbei die „AS-50“

Daten	Werte
Waffentyp	Semi (Hitscan)
Schüsse pro Minute	300
Schaden	600
Projektiltyp	Hitscan
Munition Maximum	260
Durchschlagsanzahl	12
Schussanzahl	1
Highscore Benötigt	10000

3.5.9.2.9.3 Spiel Waffen: Munitionsarten

Kaliber	Aussehen
7.62mm	
9mm	
Kartusche	
20_102mm	
M235 TPA	

3.5.9.3: Erstellung einer neuen Waffe als Video.

Hierbei wurde ein Video aufgenommen bei welchen eine Komplett neue Waffe: Erstellt wird und eingebunden wird. (Hierbei wird auf einem 2.ten Bildschirm ein Konzept der AS-50 gezeigt, Siehe: Literatur)

Es wird behandelt: 3D Modellierung, Texturing (Simple anwendungsform), Unity Einbindung (Über vorgaben Pistole) und Adjustierung der Werte.

Youtube Link: https://youtu.be/6OXO8a_ekTs

5. Literaturverzeichnis

Top Down Shooter: Unity tutorial series (01)
https://www.youtube.com/watch?v=pSN2x3dPgYw, 23.03.2020

Unity User Manual (2019.3)
https://docs.unity3d.com/Manual/index.html, 23.03.2020

Digitade Walk Cycle – MERNOLAN
https://i.pinimg.com/originals/36/ef/88/36ef88f998ecb4316c936078867a12d3.gif, 23.03.2020

Walk Cycle
https://1.bp.blogspot.com/-Qh-WczAqMFc/T22Ink7M1UI/AAAAAAAABMA/AJy640CGJSU/s1600/RW_modified_walk_lesson_07.jpg, 23.03.2020

Einige Waffen Sounds (Stark abgeändert)
https://store.steampowered.com/app/35300/Warfare/, 23.03.2020

AS-50 Concept
https://de.m.wikipedia.org/wiki/Datei:AS-50.JPG, 23.03.2020

6. Unterschriften

Unterschrift & Datum (Auftraggeber)

_____

Unterschrift & Datum (Auftragnehmer)

_____

7. Anlagen