**Meine Änderungen für das Programm**

Als erstes habe ich mir über die Internetseite Mixamo einen neuen Character und neue Animationen importiert, welche ich später benutzt habe, um einen Neuen Gegner zu erstellen. Ich wollte einen „Endboss“ programmieren, welcher als letzter Gegner die Wasserschale verteidigt. Hierfür habe ich die Map um einen weiteren Raum erweitert, wo dann der neue Gegner die leere Schale bewacht. Ich hatte mir außerdem überlegt, dass sich die Schale noch einmal hinter einem Gatter befindet, welches man öffnen kann, wenn man den „Endboss“ besiegt hat.

Als erstes Habe ich den neu importierten Gegner in die Welt gezogen und ihm einen Capsule Collider hinzugefügt, um im späteren Verlauf keine Probleme damit zu haben ihn mit den Wurfsteinen zu treffen und er mir bei Berührung auch schaden hinzufügen kann. Danach habe ich ihn genau wie die Bat mit dem Tag Enemy versehen, da er genau wie die Bat ein Gegner des Spielers ist. Anschließend habe ich ihm ein Nav Mesh Agent hinzugefügt, damit er genau wie die Bat den Spieler über die Path Finding Funktion verfolgen kann. Außerdem konnte ich mir dadurch ein extra Script, von dem der neue Gegner gesteuert wird sparen. Den Nav Mesh Agent habe ich also passend zu dem neuen Character eingestellt. Dann habe ich ihm als Gerüst die Scripts der Bat hinzugefügt, welche ich jedoch duplizieren und umbenennen musste, da er auf andere Animationen zurückgreifen soll, als die Bat. Außerdem habe ich die Stats so eingestellt, und dementsprechend gegenüber der Bat so verbessert, dass er stärker als die Bat ist, und so auch einem „Endboss“ gerecht wird. Des weiteren Musste ich die Audioclips entfernen, da ich für den neuen Gegner keine zur verfügung hatte. Jetzt habe ich die Map um eine Weiteren Raum erweitert, in Welchem der „Endboss“ dann die Schalebeschützt , welche sich jedoch noch einmal in einem weiteren 1x1 Raum befindet, welches durch ein Falltor abgetrennt und von einem extra pointlight beleuchtet wird . In diesen Raum Habe ich dann vier Wegpunkte eingefügt, sodass der „Endboss“ beim Bewachen der Schale durch den ganzen Raum geht. Danach habe ich dem neuen Gegner einen Animator Controller hinzugefügt mit fünf verschiedenen animationszuständen. Einmal der ganz normale Laufzustand, welchen er benutzen soll, wenn er von Wegpunkt zu Wegpunkt geht, oder auf mich zu läuft. Dann einen Angriffszustand, welchen er ausführen soll, wenn er ich angreift. Einen Getroffen Zustand, welchen er ausführen soll, wenn ich ihn mit einem Stein getroffen habe. Einem Umgucken Zustand, welchen er benutzt, wenn er an einem Waypoint angekommen ist. Und einen Sterben Zustand, welchen er ausführen soll, wenn er stirbt. Dann habe ich noch vier Parameter erstellt, mit welchen die einzelnen Animationen später über die Scripts gestartet/gestoppt werden sollen. Denn Laufzustand habe ich als Startanimation gesetzt, da er diese direkt am Anfang verwenden soll, um zwischen den Waypoints her zu laufen und als „Hauptzustand“ dient. Diesen habe ich anschließend mit dem Angriffszustand und dem Umgucken Zustand beidseitig verbunden, da man sowohl aus dem Laufzustand in einen der Beiden wechseln können soll, als auch vom Angriffszustand/Umgucken Zustand zurück in den Laufzustand. Als Bedingung habe ich hierbei an die Transitions von Laufen nach Angriff und Umgucken attack/look = true geschrieben, damit er in diese zustände wechselt, wenn ich diese in einem Script auf true setze. Auf die entgegengesetzten Transitions habe ich dann dementsprechend attack/look = false geschrieben, damit ich diese ebenfalls per Script wechseln kann. Von jedem Zustand (Any State) aus soll man in den Getroffen Zustand und in den Sterben Zustand wechseln können. Hierbei habe ich an die Transition von Any State zu Dying mit der Bedingung die = true versehen und den Pfeil von Any State zu Hit mit dem trigger hit, damit ich biede Animationszustände später per Script ausführen kann. Als letzte Verbindung habe ich noch eine Transition ohne Bedingung von Hit zu Walking mit einer exit time gesetzt, damit der Gegner nachdem ich ihn getroffen habe kurz in einem Schockzustand gerät und danach weiter auf mich zu läuft. Jetzt musste ich noch auf die Einzelnen Zustände in den Scripts wechseln. Den Angriffparameter habe ich in dem IEnumerator SetReadyToHit von dem MonsterAi Script auf true, und danach direkt wieder auf false gesetzt, damit der Gegner einmal Schlägt und danach direkt weiter auf mich zu läuft. In dem selben Script habe ich in dem IEnumerator WaypointPause den Umguckenparameter auf true gesetzt und nach der zeile yield return new WaitForSeconds(seconds) wieder auf false gesetzt, wobei ich die Waypoint Pause auf 3,5 Sekunden gesetzt habe, damit der Character genau einmal die Umgucken Animation abspielt und danach weiter zum nächsten waypoint läuft. Den hit trigger habe ich in der override Methode Damaging aus dem MonsterHealth Script gesetzt, damit er die Getroffen Animation abspielt, wenn er schaden bekommt. In dem Selben Script habe ich in der override Methode Dying den Sterben Parameter auf true gesetzt, damit er die Sterben Animation abspielt, wenn ich ihn getötet habe. In der selben Methode habe ich auch noch das EndGate, hinter dem sich die Schale befindet entsperrt, sodass man dieses öffnen kann, sobald man den „Endboss“ getötet hat.

**Animationen in Unity**

Als aller erstes benötigt man eine Animation, welche man später für seinen Character oder andere Objekte benutzen kann. Hierbei kann man sich schon vorgefertigte Animationen aus dem Internet herunterladen und diese dann in sein Spiel importieren, oder sich seine eigenen Animationen erstellen. In Unity gibt es hierfür sogar einen speziellen Animation-Editor, mit dem man seine eigenen Animationen aufzeichnen und verändern kann. Hierbei arbeitet Unity mit Keyframes, welche aus einem Zeitpunkt auf der Zeitachse und einem Wert der zu animierenden Eigenschaft bestehen. Das heißt man kann einen Start und einen Endzustand festlegen (z.B. Geschlossenes Fallgatter und offenes Fallgatter), wobei alle Werte zwischen den definierten Zeitpunkten von Unity rechnerisch ermittelt werden. Außerdem bietet Unity die Möglichkeit solche Übergänge mithilfe von Kurven zu beschreiben. Jetzt muss die Animation über eine FBX-Datei in Unity hineingeladen werden. Hierzu muss man jedoch in den Import Settings die richtigen Rig- Model- und Animations-Einstellungen wählen. Der Reiter Model legt die Importeinstellungen für das 3D-Modell fest, welches gegebenenfalls mit importiert wurde. Der Reiter Rig definiert die Schnittstelle zischen dem Modell und möglicher Animationen. Und der Reiter Aimations legt die Importeinstellungen für die in der FBX-Datei enthaltenen Animation(en) fest. In Unity wird jedoch zwischen zwei generellen Animationsarten unterschieden. Einmal Generic, welche dem Generieren einer Animationsschnittstelle dient, die universell einsetzbar ist. Außerdem wird automatisch ein Avatar erstellt, welcher ein wichtiger Bestandteil des Rigs ist. Und den Animation Type Humanoid, welcher speziell für humanoide Wesen geeignet ist und es ermöglicht, einen Avatar, welcher hier auch automatisch erzeugt wird, aber manuell nachjustierbar ist, und seine Animationen im späteren Verlauf wieder zu verwenden. Hat man jetzt den Animation-Clip importiert kann man diesen noch bearbeiten. Man kann sowohl den Start- sowie End-Punkt der Animation verändern als auch verschieden Parametereinstellungen vornehmen. Ist der Parameter Loop Time aktiviert, führt dies zu automatischen Wiederholungen des Clips. Der Parameter Loop Pose passt die Animation automatisch an, damit Anfang und Ende nahtlos hintereinander passen. Cycle Offset fügt dem eigentlichen Animationsstart einen Offset zu. Des weiteren kann man noch über den Reiter Mask definieren, welche Teile der Animation tatsächlich abgespielt werden. Die deaktivierten Bereiche werden dabei von der Animation ignoriert. Um eine Animation auf ein Objekt anzuwenden zu können benötigt man zwei Dinge: eine Animator-Komponente und einen Animator Controller. Die Animator Komponente wird Objekt hinzugefügt und stellt die Verbindung zwischen dem Objekt und dem Animator Controller her. Der Animator Controller definiert bei mehreren Animationen, wann welche Animation abgespielt werden soll und wie zwischen diesen gewechselt wird. Zu Anfang besitzt jeder Animator Controller drei Einträge. Den grünen „Entry“- Node, der den Einstieg in den Animator Controller darstellt. Dieser zeigt mit einem orangenen Pfeil auf den Default-State, der als Allererstes ausgeführt werden soll. Den roten „Exit-Node, der einen möglichen Ausstieg aus dem Animator Controller darstellt. Dieser wird jedoch nur bei Animator Controllern benötigt, die für Wiederverwendungszwecke von anderen States entkoppelt werden sollen. Und dem türkisfarbenen State „Any State“. Dieser symbolisiert jeden Zustand des Animator Controllers. Dieser State dient also dem Wechseln in einen anderen Zustand, unabhängig davon, in welchem man sich momentan befindet. Neue Animation States fügt man dem Animator Controller hinzu, indem man einfach einen Animation-Clip in den Controller rein zieht. Der erste State, den man dem Controller hinzufügt, ist automatisch der Default-Zustand. Über Transitions (als Pfeil dargestellt) kann man Bedingungen festlegen, wann welcher Animation State aktiv ist. In den Eigenschaften der Transitions kann man jetzt im Bereich Conditions die Bedingung festlegen, wann vom einen in den anderen Zustand gewechselt werden soll. Im Animator Controller kann man auf dem Reiter Parameters beliebig viele neue Variablen definieren, welche für das Steuern dieser Wechsel zur Verfügung stehen. Außerdem kann man in den Transitions Eigenschaften über den Kenner Has Exit Time festlegen, ob die Transition eine feste Austrittszeit aus dem State besitzen soll. Hierbei wird dann nicht sofort in den nächsten zustand gewechselt, sobald die Bedingung erfüllt wurde, sondern noch die Animation zu Ende abgespielt. Um Animationen parallel abspielen zu können muss man einen zweiten Layer erstellen. Über die Blending Eigenschaft legt man fest, wie die Animation des Layers mit dem Standard Layer kombiniert werden soll. Override bedeutet hierbei, dass die anderen Animationen ignoriert, und somit nur diese dargestellt wird , Additive bedeutet, dass die Animationen gemischt werden und Weight beschreibt, wie stark der Einfluss dieses Layers ist. Um nur bestimmte Teile der Animation zu nutzen (z.B. nur Arme) gibt es eine Avatar Mask, die man in einem Layer hinterlegen kann. In dieser kann man dann bestimmen, welche teile animiert werden sollen und welche nicht. Um komplexe Vorgänge im Animator Controller geordneter darzustellen oder wiederverwendbar zu machen, kann man bestimmte Animation States mithilfe von Sub-State Machines ausgliedern, welche man jedoch durch einen Doppelklick auf die Sub-State Machine jederzeit einsehen kann. In der Sub-State Machine wird der obere State Machine durch einen orangenen Node „(Up) Base Layer“ dargestellt und alle Transitions, die von der oberen Ebene kommen gehen von „(Up) Base Layer“ zu den jeweiligen Animation States. Hat man jetzt den Animator Controller erstellt, muss man noch dafür sorgen, dass dieser auch das richtige Objekt steuert. Hierfür ziehen wir den Animator Controller in die Eigenschaft Controller des Animator des Objekts. Außerdem muss man noch in der Eigenschaft Avatar des Animators Festlegen, welcher Avatar animiert werden soll. Um die Parameter, welche zum Wechsel zwischen den einzelnen Animation States zuständig sind, mit Werten zu füllen, muss man in einem Script auf den Animator zugreifen. Hierfür speichert man die Referenz auf diesen in einer private- oder public-Variablen. Mit anim = GetComponent<Animator>() kann man dann auf die Parameter des Animators zugreifen. Um nun Den Wert des Parameters zu ändern gibt es zwei Möglichkeiten. Einmal mit anim.SetBool(„attack“,true), wobei man auf den Parameternamen „attack“ zurückgreift, oder man arbeitet mit Hash-Werten, welche lediglich eine Umwandlung Des Parameternamens Darstellt. Hierbei deklariert man in der Start Methode beispielsweise attackBool = Animator.StringToHash(„attack“) und kann dann im folgenden verlauf den Parameterwert mit anim.SetBool(attackBool,true) verändern. Mit GetCurrentAnimatorStateInfo kann man den aktuellen AnimationState ermitteln. Dies funktioniert jedoch ausschließlich, indem man den Hash-Wert ermittelt. Deshalb muss man hier den zu suchenden Animation State erst einmal in seinen Hash-Wert umwandeln und vor dem State-Namen auch noch den Layer angeben: runState = Animator.StringToHash(„Base Layer.Run“). Nun übergibt man noch den Layer-Index von dem man den aktuellen Animation State wissen möchte und dann kann man über die Methode nameHash den Hash-Wert von diesem erhalten: anim.GetCurrentAnimatorStateInfo(0).nameHash == runState