Hochschule Bielefeld

Programmiermethoden
Konzept
Zyklus 3

Game Over
Monster
Spiel speichern und laden

Matteo Antonuccio
Sascha Hahn
Moritz Lützkendorf

Abgabe: 18.05.2023

Inhaltsverzeichnis

1 Game Over

Wenn der Held stirbt, soll ”Game Over” auf dem UI angezeigt werden. Es soll zwei Buttons *Neustart* und *Beenden* geben, auf die man mit der linken Maustaste anklicken kann. Da man noch nicht sterben kann, soll der Held mit der Taste K sterben. Die `onDeath()` Funktion für den Hero muss definiert werden. Sie soll wie beim Pause-Menu alle ECS-Systeme stoppen und den GameOver-Screen auf *visible* stellen. Da es noch kein Schaden im Dungeon gibt muss eine Selbstmordfunktion eingebaut werden. Im Playersystem muss die Taste K hinzugefügt werden, mit der dann die Healthpoints des Helden auf 0 gesetzt werden, wodurch dann die `onDeath()` Funktion ausgelöst wird. Der Gameover-Screen soll zentriert in groß “GameOver” anzeigen. Unter diesem sind nebeneinander die zwei Buttons “*Restart*” und “*Exit Game*”. Der GameOver-Screen ist an sich erst mal eine Kopie von dem `PauseMenu`, dem noch Buttons hinzugefügt wurden. Die Buttons werden mit der bereits vorhanden Klasse `ScreenButton` implementiert. Beim klick auf den ExitGame Button wird das Spiel mit der Methode `System.exit(0)` beendet. Bei dem Restart Button wird der Hero in der `Game` Klasse neu initialisiert, das LevelAPI Objekt wird neu initialisiert, die Systems werden wieder aktiviert und anschließend die `onLevelLoad` Methode ausgeführt. Bei dem Attribut levelNummer in der `LevelAPI` Klasse wird das “static” entfernt, damit auch diese auch automatisch im Konstruktor auf den Startwert zurückgesetzt wird. (und da das static sowieso nicht da hingehört).

1.1 UML

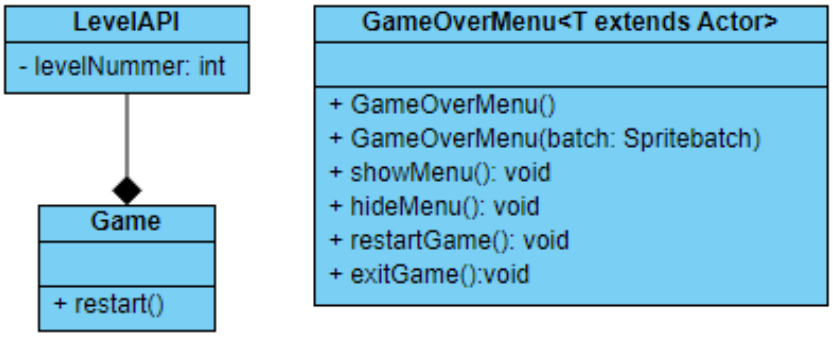

Abbildung 1: Klassendiagramm

2 Monster

Monster sollen das Spiel schwieriger, aber auch spannender und abwechslungsreicher gestalten. Die Monster können dem Spieler Schaden zufügen oder man die Monster bestehlen und ihre Items schauen sowie sie töten oder ihnen ebenso Schaden zufügen.

2.1 Codeanalyse

Die Monster sollen von einer AI gesteuert werden. Dafür stehen schon fertige Klassen und Interfaces bereit, die die Monster als Entitäten erben können und so von ihnen gebrauch gemacht werden kann.

2.1.1 Components

Die Monster brauchen Komponenten, die sie selbstständig im Dungeon laufen lassen, eigene Interaktionen durchführen können, beispielsweise den Spieler angreifen oder mit dem Angriff warten, sowie den Spieler verfolgen können beziehungsweise diesem im Blick haben.

2.1.2 AI-System

Im Programmquelltext lassen sich Interfaces und Klassen finden, die die Monster erben können, damit sie von der AI gesteuert werden können. In der Klasse `AISystem`, die sich im Pfad *game/src/ecs/systems* befindet, lässt sich beispielsweise die AI kontrollieren. In dem Pfad *game/src/ecs/components/ai* lassen sich verschiedene Unterordner finden mit denen man Entitäten, die mit einer AI verknüpft sind, laufen und kämpfen lassen kann sowie ihnen die Möglichkeit gibt zu entscheiden, ob der Spieler angegriffen werden soll oder nicht.

In dem Ordner *fight* findet man zwei Klassen (`CollideAI` und `MeleeAI`) sowie ein Interface (`IFightAI`). Die Klasse `CollideAI` sorgt dafür, dass der Spieler angegriffen wird, falls dieser sich in einer bestimmten Reichweite befindet beziehungsweise das Monster in Richtung des Spielers bewegt. Die Klasse `MeleeAI` kümmert sich ebenso um den Angriff, erweitert diesen aber mit der Möglichkeit Fähigkeiten mit einzubinden. Das Interface `IFightAI` implementiert das Kampfverhalten.

In dem Ordner *idle* gibt es drei Klassen (`PatrouilleWalk`, `RadiusWalk` und `StaticRadiusWalk`) und ein Interface (`IIdleAI`). Die Klasse `PatrouilleWalk` sorgt dafür, dass eine Art Patrouillengang erzeugt wird, den das Monster immer wieder läuft. Die Klasse `RadiusWalk` lässt das Monster zu einem bestimmten Punkt in einem bestimmten laufen. Nachdem der Punkt erreicht wurde wird ein neuer Punkt genertiert zu diesem das Monster läuft. Die Klasse `StaticRadiusWalk` gleicht der obigen Klasse, jedoch mit dem unterschied, dass der Punkt immer weit entfernt vom Zentrum des Radius liegt .Das Interface `IIdleAI` implementiert das Verhalten Nichtstun.

In dem Ordner *transition* befinden sich zwei Klassen (`RangeTransition` und `SelfDefendTransition`) und ein Interface (`ITransition`). Die Klasse

`RangeTransition` sorgt dafür, dass eine Entität in den Kampfmodus wechselt, wenn ein Spieler sich in der Nähe befindet. Die Klasse `SelfDefendTransition` sorgt für die Selbstverteidigung einer AI gesteuerten Entität.

Das Interface `ITransition` implementiert einen Entschluss, um zwischen den Modi Kampf und Nichtstun zu wechseln.

Außerhalb der Unterordner stehend gibt es noch zwei weitere Klassen. Diese sind `AIComponent` und `AITools`. Die Klasse `AIComponent` speichert das Verhalten Kampf und Nichtstun einer von der AI gesteuerten Entität. Die Klasse `AITools` setzt die Geschwindigkeit einer AI gesteuerten Entität

2.2 Monster im Dungeon

2.2.1 Riesenmaus

Die Riesenmaus ist schnell und flink. Sie kann dem Spieler schnell ausweichen, ist aber auch schnell beim Spieler.

Abbildung 2: Micky goes wild

2.2.2 Grottenolm

Der Grottenolm ist blind und kann somit den sich annähernden Spieler nicht mehr sehen.

Abbildung 3: Der Grottenolm

2.2.3 Feuersalamander

Der Feuersalamander macht seinen Namen alle Ehre. Wird er berührt, dann fühlt es sich für den Spieler so an, als ob seine Hand brennen würde. Er bekommt einen großen Schaden.

Abbildung 4: Feuersalamander. Mach Beine auseinander.

2.3 UML

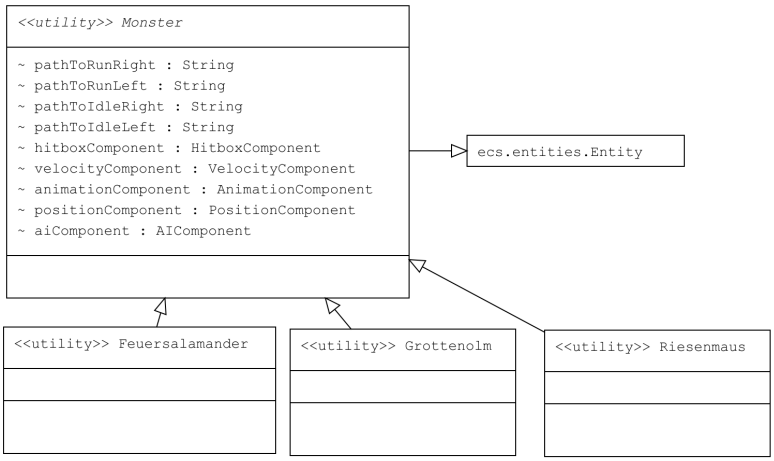

Abbildung 5: Klassendiagramm

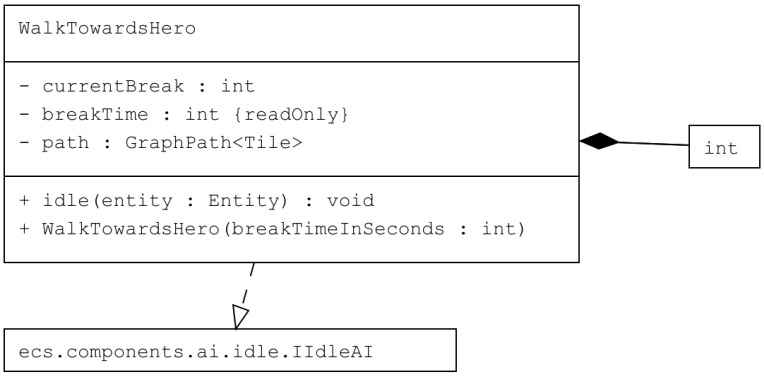

Abbildung 6: weitere IdleAI-Strategie

3 Spiel speichern und laden

Der Spieler hat die Möglichkeit den momentanen Spielstand zu speichern und diesen zu einem späteren Zeitpunkt, auch nach Beenden des Programmes, wieder zu laden. In dieser Aufgabe wird das Interface `Serializable` aus dem Paket `java.io` eingebunden. Nach Implementieren des Interfaces kann von Methoden Gebrauch gemacht werden, die Daten speichern oder z.B. aus einer Datei laden können. Hierbei wird der momentane Spielstand gespeichert. Dazu gehören beispielsweise die Position des Spielers, die Lebenspunkte oder die Items im Inventar.

3.1 UML

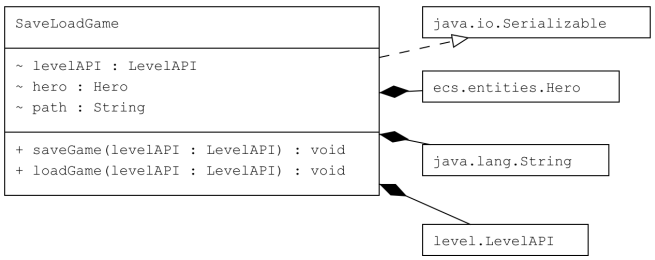

Abbildung 7: Klassendiagramm