Of Monsters And Men

Designkonzept und Ausarbeitung

Nico Klein, 256219

16.02.2020

Spielidee:

Holde Maiden und fiese Gestalten so Weit das Auge reicht. In „Of Monsters And Men“ begleiten wir einen namenlosen Ritter auf seinem Streifzug durch fremde Lande auf dass er eines Tages seine Angebetete Alise wiederfinden möge. Jedoch ist das Land einer Armee von magischen Wesen anheimgefallen und nun ist es an ihm das Land von diesen Kreaturen zu befreien und seine holde Maid wieder zu finden.

Steuerung:

Nach links gehen: A

Nach rechts gehen: D

Springen: W halten

Schlagen: H halten

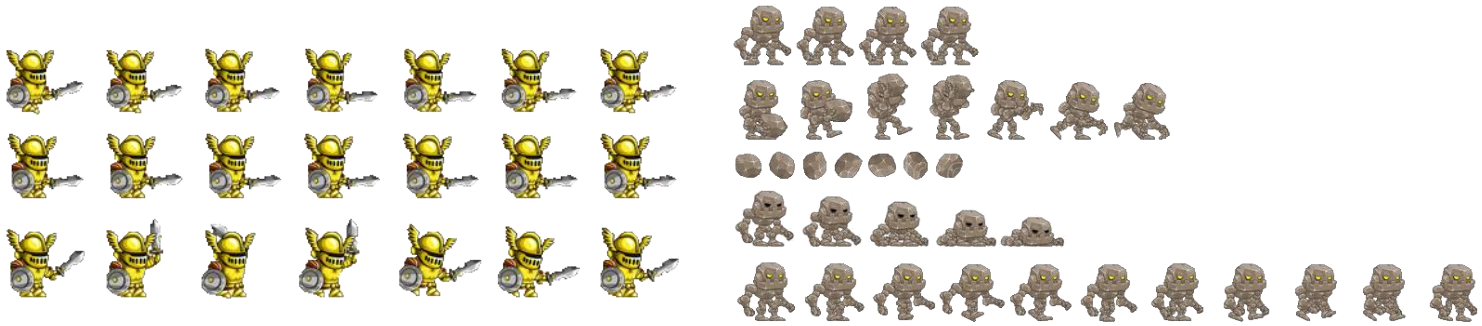
Features, welche das Spiel haben soll (und auch hat):

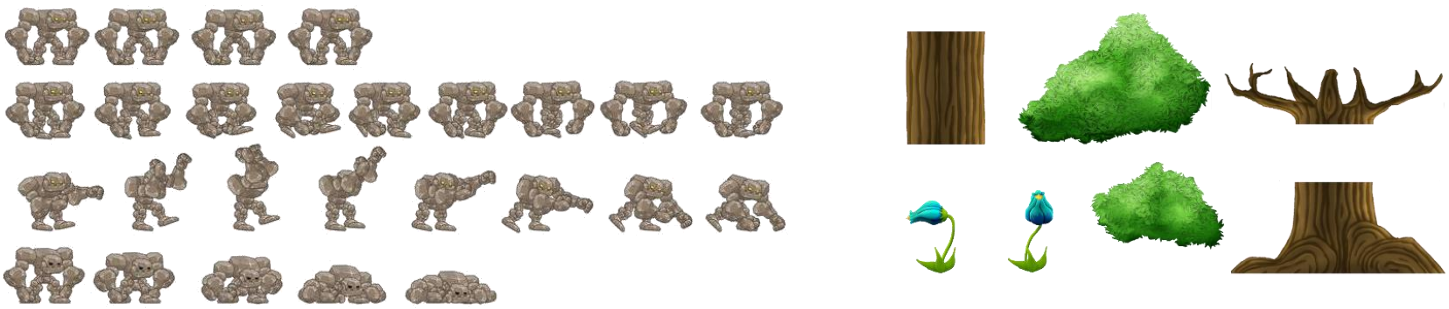
1. Schwertkampf: Der Spieler soll in der Lage sein sich per Schwert gegen Feinte behaupten zu können.

2. Items: Items wie tränke sollen in der Welt verteilt sein, um den Spieler zu unterstützen.
3. Gegner: In dem Spiel soll es verschiedene Arten von Gegnern geben.
    a. Fernkampf
    b. Nahkampf
4. Lebensleiste: Der Spieler soll immer sein Leben im Blick haben können.
5. Menü: Es soll ein Menü mit Einstellungen geben.
6. Paralax-Scrolling

Artstyle:

Die Sprites sind alle aus dem Internet geholt worden. Ich habe mich für einen relativ Zeitlosen Comic-Stil entschieden.

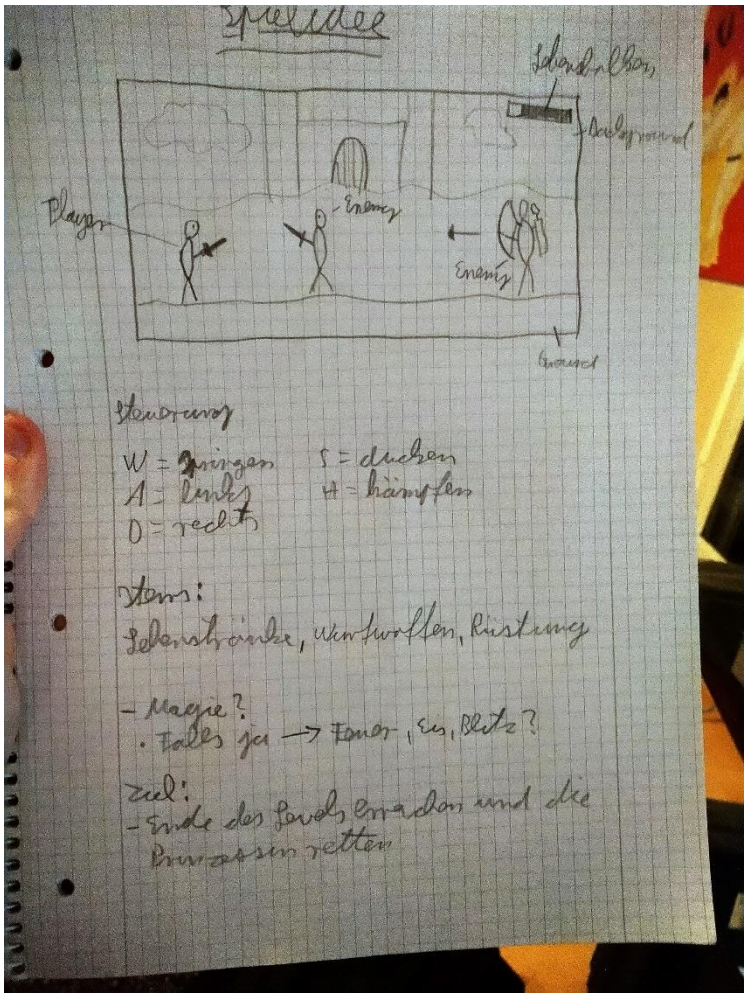


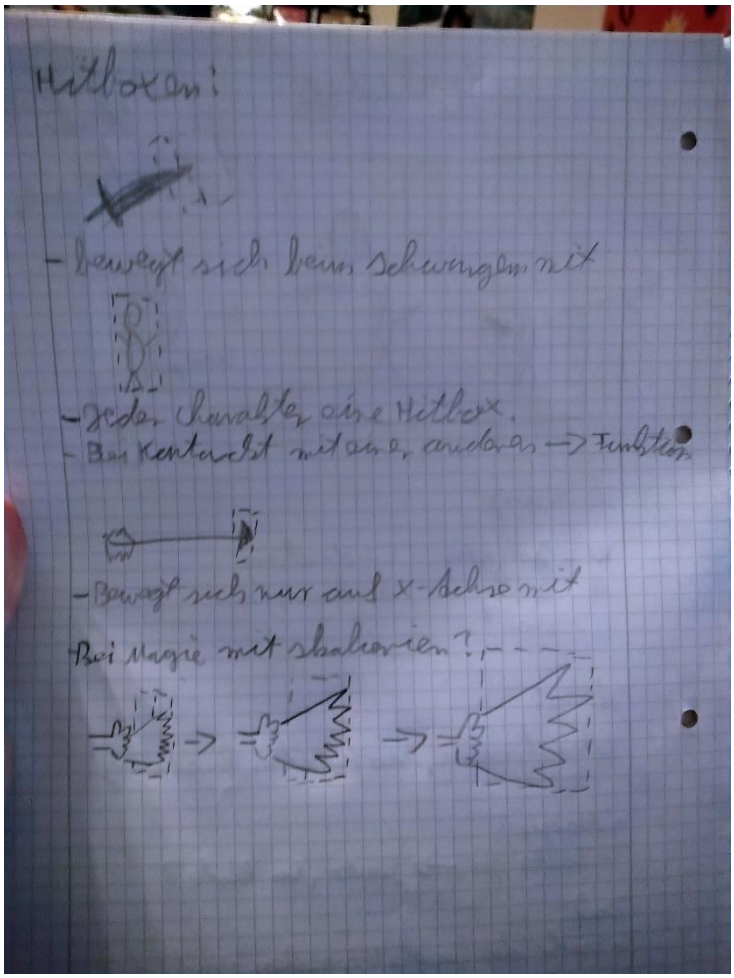


Ziel des Spiels:

Weil es sich bei dem momentanen Spiel nur um einen Prototyp handelt und dieser nur aufzeigen soll, wie sich das Spiel später anfühlen kann, gibt es momentan noch kein Ziel. Später jedoch, soll es mehrere Level geben, deren Ende der Spieler erreichen muss, um seine Prinzessin zu retten.

Erstes Konzept:





Ausarbeitung:

Nr.	Bezeichnung	Inhalt
1	Nutzerinteraktion	Der Nutzer kann die Figur mithilfe von A und D nach links und rechts bewegen. Zusätzlich ist es möglich durch das Gedrückt halten von W und H zu springen und zu schlagen.

		Weitere Interaktionsmöglichkeiten sind die Knöpfe im Startbildschirm, mit denen sich das Spiel starten lässt und der Steuerungsbildschirm aufrufen lässt. Zudem kann man jederzeit während des Spiels das Spiel anhalten indem man das Menü in der linken oberen Ecke öffnet. Hier kann der Spieler zu jeder Zeit die Lautstärke verschiedener Sound-Kategorien anpassen.
2	Objektinteraktion	Die präsenteste Objektinteraktion ist die Abfrage der „ground kollision“. Hier wird abgefragt, ob sich zwei definierte Punkte des Charakters (ein Punkt links und ein Punkt rechts seines Pivots) innerhalb eines Rechtecks befinden, welches den Boden bildet. Ist dies der Fall, wird der Charakter zurück auf die obere Kante des Rechtecks gesetzt. Zudem verfügt jeder Charakter (Spieler und Gegner) über eigene Hitboxen, welche in jedem Frame abfragen, ob sich in ihnen einer der Eckpunkte einer anderen Hitbox befindet. Falls ja, wird je nachdem welchen Typ die Hitbox hat (Gegner, Item, Spieler, Schwert) Schaden ausgeteilt, Leben aufgefüllt usw.…
3	Objektanzahl variabel	Sämtliche Objekte im Spiel (Items, Platformen, Bäume, Hügel, Gegner, Boden oder Hintergrund) werden bei Start des Spieles generiert und in das Level eingefügt. Welche Objekte das genau sind und wie viele es sind, Hängt komplett von der eingespeisten JSON-Datei ab, in der sich alle Informationen zu den zu erstellenden Objekten befinden. Nach dem Klick auf den Startknopf, wird die „startGame“-Methode aufgerufen, welche wiederrum die „createLevel“-Methode aufruft. Diese geht die Einträge der JSON-Datei durch und ruft immer die passenden Methoden zum Erschaffen der Objekte mit den angegebenen Koordinaten und sonstigen Informationen auf. Diese werden dann als Kind an das Level gehängt. Komplett während des laufenden Spieles erschaffene Objekte sind die Geschosse des Fernkämpfers (Gegner). Betritt der Protagonist seinen Sichtbereich, beginnt er in seiner „createProjectile“-Methode neue Objekte der Klasse „Stone“ zu generieren, welche in Blickrichtung des Fernkämpfers fliegen. Diese Aktion führt er so lange aus, bis sich der Spieler weit genug von dem Gegner entfernt.
4	Szenenhierarchie	Die Szenenhierarchie wird in Abbildung 1 dargestellt. Wie dort zusehen ist, gehen von dem Knoten „Game“ zwei weitere Knoten aus. In „Level“ befinden sich alle für eben dieses eine Level nötigen Knoten. Alle Gegner, Dekorationen Hitboxen usw….

		Wenn man nun ein zweites, dritte oder viertes Level hinzufügen möchte, wäre das durch das einfache hinzufügen eines neuen Levelknotens ohne Probleme möglich. Ich habe diese Hierarchieform gewählt, weil so die Abgrenzung der wichtigsten Spielbestandteile, der Level, sehr leicht und übersichtlich ist und eben das Implementieren eines neuen Levels sehr einfach ist. Der Charakter ist vom Level losgelöst, weil er immer derselbe sein soll. So muss der Charakter nicht bei jedem neuen Level neu erstellt werden und zukünftige Statusveränderungen müssten nicht in jedem neuen Level neu auf den Charakter angewendet werden.
5	Sound	Die eingebauten Sounds lassen sich unterteilen in 4 Kategorien: 1. Musik: Die Hintergrundmusik wurde aus dem Spiel „Skyrim“ entliehen und soll den Spieler etwas anpeitschen und in Kampflaune bringen. 2. Effekte: Unter Effekte fallen Sounds wie der Hieb des Schwertes, das Trinken eines Lebenstrankes, die Laufgeräusche und der Sound beim treffen eines Gegners. 3. Stimmen: Unter Stimmen fallen in diesem Beispiel vor allem die Schmerzlaute des Protagonisten und der Monster. 4. Umwelt: In diesem Teil des Spieles handelt es sich hierbei um das Säuseln des Windes.
6	GUI	Es gibt drei verschiedene GUI-Bereiche: 1. Startbildschirm: Hier hat der User die Möglichkeit, durch einen Klick auf den dementsprechenden Knopf, das Spiel zu starten oder sich die Steuerung anzeigen zu lassen. Im Falle des Anzeigens der Steuerung, wird einfach das Div „controllScreen“ von „hidden“ auf „visible“ geschaltet. Das kann natürlich wieder rückgängig gemacht werden mit einem Knopf auf dem Steuerungsbildschirm. Beim starten des Spiels wird der Startbilschirm auf „hidden“ gestellt und der fudge.loop des Spieles gestartet. 2. Menü: In der linken oberen Ecke des Spieles befindet sich das Menü. Hier kann der Spieler die Lautstärker der einzelnen Soundkategorien separat einstellen und im laufenden Spiel anpassen. Wobei laufendes Spiel nicht ganz richtig ist, weil beim Aufruf des Menüs der loop angehalten wird. Zusätzlich kann der Spieler auch hier auf den Steuerungsbildschirm zugreifen und sich die Steuerung anzeigen lassen. 3. Todesbildschirm: Hier kann der Spieler das Spiel neu starten. Dabei wird der Tab neu geladen.

7	Externe Daten	Das Spiel erhält sämtliche Informationen darüber, welche Objekte, also Gegner, Items, Hügel usw… sich wo in der Welt zu befinden haben aus einer externen JSON-Datei. Hier sind sämtliche Objekte mit deren entsprechenden Positionen in Form von X- und Y-Koordinaten aufgelistet. Möchte man also neue Objekte von irgendwas in der Welt erzeugen lassen, muss man einfach das entsprechende Objekt in die JSON-Datei eintragen. Abbildung 2 zeigt einen kleinen Ausschnitt aus der JSON-Datei.
8	Verhaltensklassen	Die hierfür wohl interessantesten Klassen sind die der im Spiel vorkommenden Charaktere, also die des Ritters und der Gegner. In einigen Funktionalitäten überschneiden sich diese Klassen, hier liegt ein Unterschied dann meist im Detail. Nun folgt eine Auflistung der einzelnen Methoden, sortiert nach gemeinsamen und unterschiedlichen Methoden der Gegner- und Ritter-Klasse 1. Gemeinsames: Die erste „Fähigkeit“, welche sich beide Parteien teilen ist das Nehmen von Schaden. Wird der Ritter von einem Gegner oder ein Gegner von dem Schwert des Ritters erwischt, wir die Funktion “receiveHit“ aufgerufen und stößt zum einen die Figuren zurück und zieht ihnen zum anderen Schaden ab. Als zweites können alle Charaktere eine eigene Hitbox erzeugen mithilfe von „createHitbox“. Zudem verfügen beide über eine „act“- und eine „show“-methode, die zum einen, die aktuelle Aktion des Charakters bestimmt und zum anderen die entsprechenden Sprites aktiviert. Auch verfügen sie über update Methoden, welche jeden Frame die Hitboxen der Charaktere anpassen, sie in die gewählte Richtung bewegen und eine Funktion namens „checkGroundCollision“ aufruft, welche Checkt, ob der Character sich noch auf dem Boden befindet. Weicht er zu sehr von dem festgelegten Wert ab, wird seine Position korrigiert. Zuletzt verfügt jeder Charakter über die Funktion „deleteThis“, welche den Charakter aus dem Spiel nimmt, indem er seinem Parent-Knoten entfernt wird. Zusätzlich wird die zu diesem Charakter gehörende update Funktion aus der Loop entfernt. 2. Funktionen der Gegner: checkDistanceToPlayer() achtet darauf, ob sich der Spieler in einer bestimmten Entfernung zum Gegner befindet. Falls ja, wird die Aktion des Gegners dauerhaft auf Angriff gesetzt. 3. Funktionen des Ritters: Der Ritter verfügt neben der Fähigkeit eine Hitbox für sich selbst, auch noch eine für sein Schwert zu erschaffen, mit welcher dann

		später erfragt wird, ob er einen Gegner getroffen hat. Mit updateHealthbar(), wird je nachdem ob der Ritter Schaden erhalten hat oder sich heilt die angezeigte Lebensanzeige aktualisiert. Die Funktion „checkInteractions“ fragt jeden Frame ab, ob der Ritter mit einem Item oder Gegner in Berührung gekommen ist, falls ja, wird updateHealthbar() aufgerufen. Zusätzlich wird auch die Hitbox des Schwertes berücksichtigt und auf Kollisionen untersucht.
9	Subklassen	Insgesamt erben 6 Klassen von fudgeCore: Node. Diese sind Characters, Floor, Healthpoints, Hitbox, Items und Flora. Des Weiteren erben 2 Klassen von Characters, nämlich Knight und Enemy. Knight beinhaltet teilweise ähnliche Funktionen wie die späteren Kinder der Klasse Enemy, jedoch sind die Unterschiede im Detail dann doch zu groß, um alle Funktionen einfach in Characters auslagern zu können. Von der Klasse Enemy wiederum erben EnemyMelee und EnemyRanged. Die einzigen Klassen, welche von niemandem erben, sind die Klassen Sound und SpriteGenerator. Eine Detaillierte Übersicht zu den Klassen in Form eines Klassendiagrammes sind unter Abbildung 3 zu finden.
10	Maßen & Positionen	Die Skalierung aller Objekte in der Spielwelt wurde dem „Knight“ also dem Protagonisten angepasst. Er wurde immer als Referenzgröße verwendet, um nachvollziehbare Größenverhältnisse darzustellen. Bei der Platzierung jeglicher Objekte in der Welt, wird immer von dem Nullpunkt der „mtxWorld“ ausgegangen. Von dort aus werden alle per alle Objekte per translate-Funktion an die entsprechende Position verschoben. Die Hitboxen wiederrum orientieren sich dann in jedem Frame an der „mtxWorld“-Position des zugehörigen Charakters. Bewegt dieser sich also auf der Weltachse, macht es die Hitbox ebenso.
11	Event-System	Das Events-System wir für eine Vielzahl an Aktionen verwendet: 1. HTML-Elemente: Bei der Interaktion des Users mit bestimmten HTML-Elementen werden onclick-Events ausgelöst, welche beispielsweise Screens ein- und ausblenden. 2. Sound: Im Menü befinden sich einige Regler, mit denen der User im aktiven Spiel die Lautstärke erhöhen oder verringern kann. Hierbei wird ein onclick-Event verwendet, welches die Volume-Eigenschaft von einem Sound um den aktuellen Wert des Reglers verändert.

		3. Restart: Durch einen Klick auf den Restart-Button, wird der Tab, durch einen location.reload, neu geladen. 4. Steuerung: Mithilfe von Keydown-Events wird dem Charakter die Fähigkeit zu springen verliehen. Die vom Event angesteuerte „handleJump“-Funktion lässt den Charakter, wenn die W-Taste gedrückt ist, seine Sprung-Aktion ausführen. 5. Loop: Der letzte Verwendungszweck der Events ist bei den Spielinternen Loops. Sobald der Loop gestartet wird, werden jeden Frame die update Methoden der Main und der Charakter-Klassen aufgerufen.

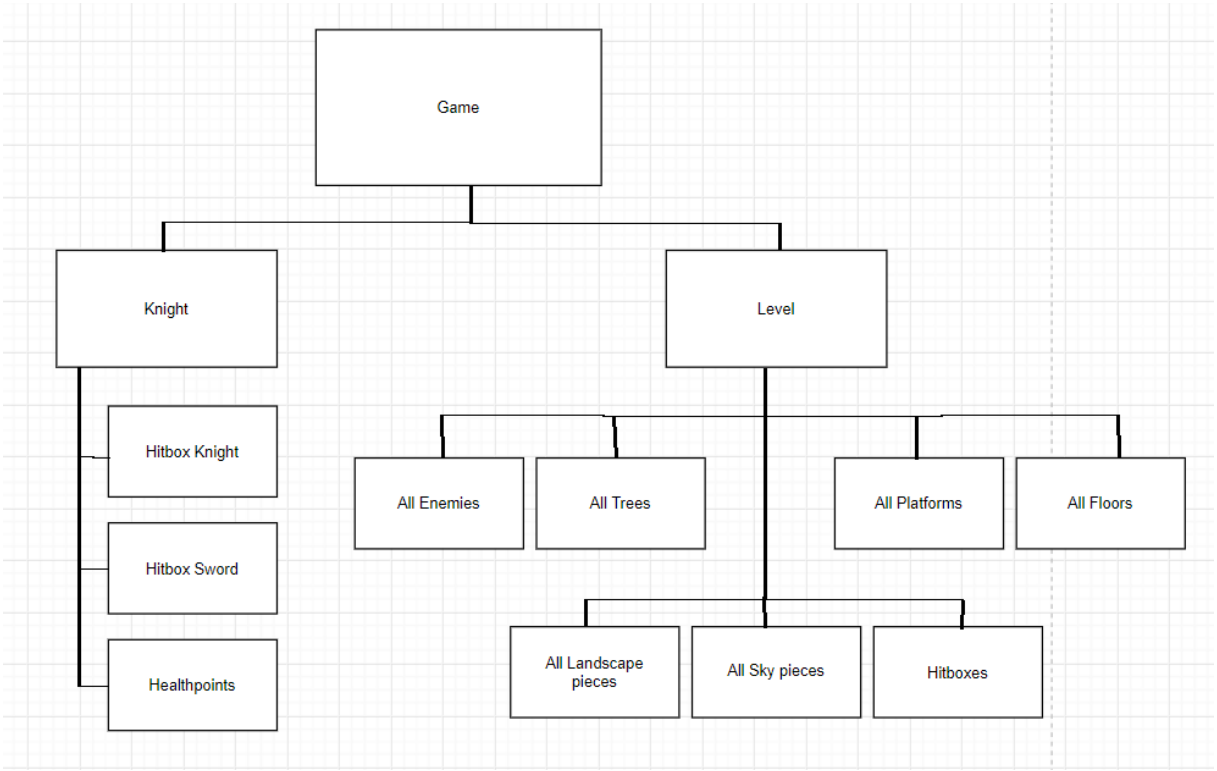


Abbildung 1: Szenenhierarchie

```
[{
    "level1":{
        "levelObjects":[
        {"objectName" : "Enemy", "type" : "Ranged", "posX" : 12, "posY" : 1, "scaleX": null, "scaleY": null},
        {"objectName" : "Enemy", "type" : "Melee", "posX" : 3, "posY" : 1, "scaleX": null, "scaleY": null},
        {"objectName" : "Enemy", "type" : "Melee", "posX" : 16, "posY" : 1, "scaleX": null, "scaleY": null},
        {"objectName" : "Floor", "type" : "Grass", "posX" : null, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Item", "type" : "Potion", "posX" : 5, "posY" : 1.5, "scaleX": null, "scaleY": null},
        {"objectName" : "Item", "type" : "Potion", "posX" : 19.5, "posY" : 1.5, "scaleX": null, "scaleY": null},
        {"objectName" : "Hill", "type" : "Small", "posX" : 2.5, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Hill", "type" : "Small", "posX" : 20, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Hill", "type" : "Small", "posX" : 42, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Hill", "type" : "Big", "posX" : 9, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Hill", "type" : "Big", "posX" : 27, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Platform", "type" : "Small", "posX" : 1, "posY" : null, "scaleX": null, "scaleY": null},
        {"objectName" : "Platform", "type" : "Small", "posX" : 4.6, "posY" : null, "scaleX": null, "scaleY": null},
```


Abbildung 2 Auszug aus der JSON-Datei

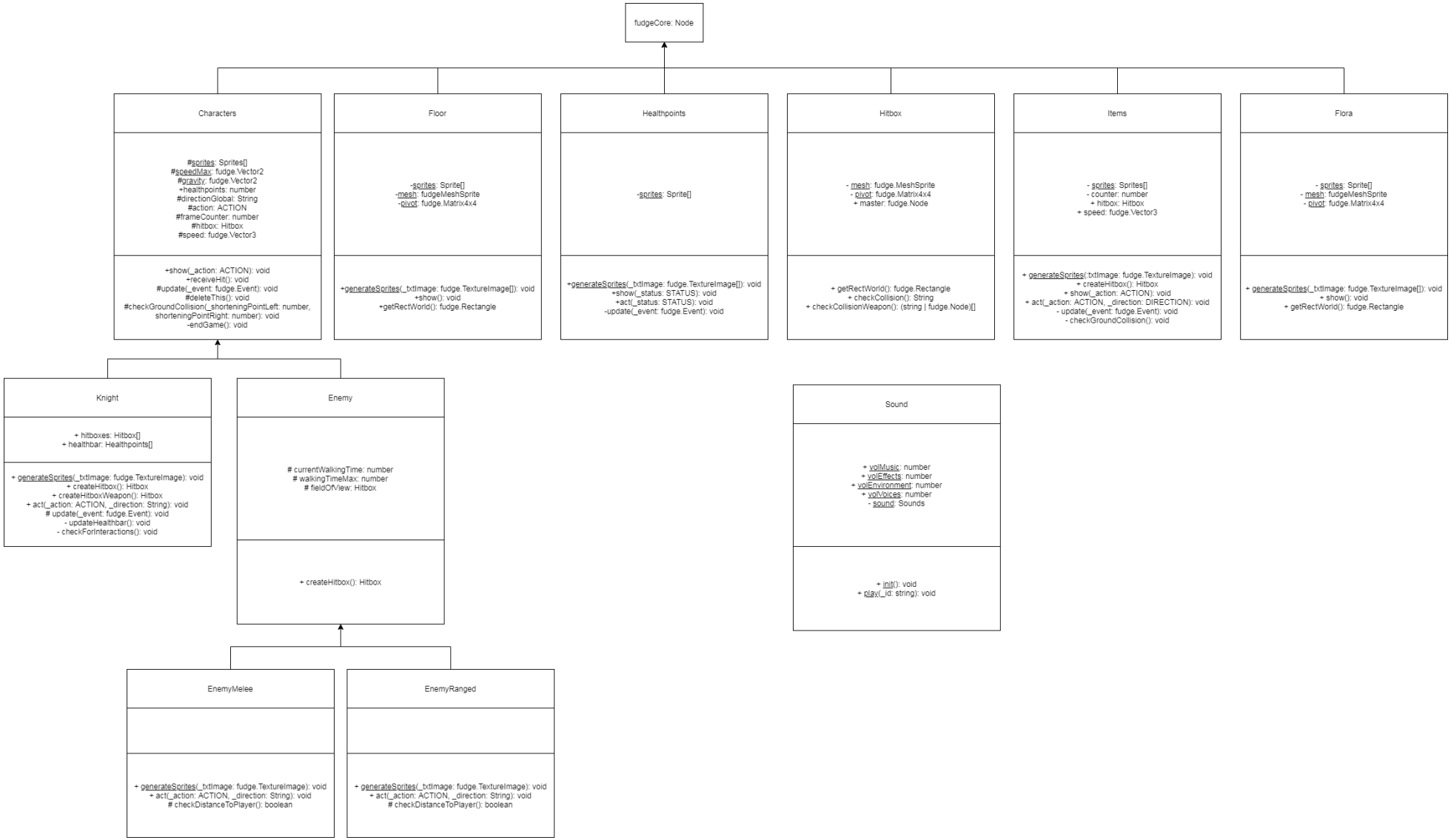


Abbildung 3 Klassendiagramm