CT Projekt: Raycasting engine (Hundefels 2D)

Christian Korn

20.10.2021 - 11.01.2022

Inhaltsverzeichnis

1 Ziele

1.1 Muss-Ziele

Wenn diese Ziele nicht erreicht werden, wird das Projekt als Fehlschlag angesehen.

- Anzeigen eines 2D Levels in 2,5D (Raycasting Methode) ✓
- Bewegungsfreiheit im Level (Translation und Rotation) ✓

1.2 Soll-Ziele

Diese Ziele müssen nicht unbedingt erreicht werden, sind aber für einen vollen Erfolg nötig.

- Laden von Leveln aus Dateien ✓
- Anzeigen von anderen Objekten im Level (z.B. Gegner, Items) ✓
- Kollisionserkennung ✓

1.3 Kann-Ziele

Diese Ziele sind nicht nötig, können aber nach Vollendung der Höheren Ziele in Angriff genommen werden.

- Gegner KI
- Schießen
- Sprites
- Texturen für Wände
- visuelle Effekte (view bobbing, Blutspritzer)

2 Verwendete Technologien

2.1 Python

Das Projekt wurde mit Python `3.9.7` erstellt, müsste aber auch in späteren Versionen funktionieren.

Externe Libraries

- Pygame: Installation mit "`pip install pygame`" Verwendet für Darstellung.
- Numba: Installation mit "`pip install numba`" Für 'magische' Leistungsverbesserungen von besonders aufwändigen Funktionen durch JIT-Compilierung.

IDE

Es wurde die PyCharm Community Edition verwendet.

2.2 Dokumentation

Die Projektdokumentation wurde mit LaTeXerstellt, UML Klassendiagramme wurden mit YUML erstellt.

2.3 Versionskontrollsystem

Das GIT Repository kann unter `https://github.com/MacAphon/hundefels2d` gefunden werden.

3 Mathematische Funktionsweise

Die Berechnungen werden 1 mal pro Frame ausgeführt. Idealerweise heißt das, dass sie 60 mal pro Sekunde erfolgen. wenn die Rechenleistung nicht ausreicht wird eine Warnung angezeigt.

3.1 Bewegung

Der aktuelle Bewegungszustand und die Position werden in den Variablen `_state`, `_movement` und `_position` gespeichert.

`_state` enthält die x- und y-Bewegungsgeschwindigkeit, sowie Rotationsgeschwindigkeit relativ zum Spieler, `_movement` enthält die absoluten Geschwindikeitswerte.

Für jeden Frame wird überprüft, welche tasten aktuell gedrückt sind und dementsprechend `_state` angepasst.

Aus `_state` wird dann `_movement` berechnet und diese Werte mit `_position` addiert.

3.2 Darstellung

Welt

Vom Spieler ausgehend werden Strahlen in Paaren ausgesendet:

Der horizontale Strahl überprüft bei vertikalen Linien, der vertikale Strahl bei horizontalen Linien. Mit dem Unterschied der Richtung funktionieren beide insgesamt gleich, die Erklärung konzentriert sich daher auf den horizontalen Strahl.

Zuerst wird der x-Wert des Strahls x_{ray} auf die nächste Linie gesetzt, danach der y-Wert y_{ray} mit folgender Formel bestimmt:

$$y_ray = x_{player} - x_{ray} * (-1/\tan(\alpha_{ray}) + y_{player}$$

Nun wird der Strahl wiederholt um 1 in x-Richtung und um

$$-block\_size * (-1/\tan(\alpha_{ray})$$

in y-Richtung verschoben. Nach jedem Verschieben wird überprüft, um es sich beim neuen Block um eine Wand handelt, wenn ja wird abgebrochen und die zurückgelegte Entfernung, sowie die x- und y-Koordinaten des Strahls zurückgegeben.

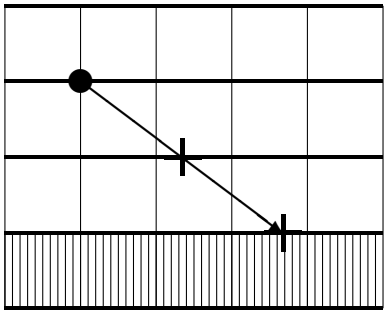
Von den zwei so berechneten Werten wird nun der kürzere Strahl ausgewählt, auf der Karte gezeichnet und als vertikale Linie der Länge

$$v\_offset = \frac{1}{dist + 0,001} * 9000$$

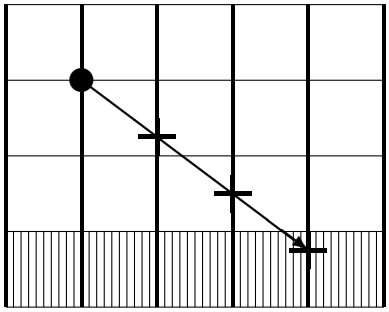
im Viewport angezeigt (*dist* ist die Länge des Strahls, die zwei festen Werte verhindern, dass durch 0 geteilt wird und erhöhen den Wert auf eine sichtbare Länge).

Beispiel

Horizontaler Check, vertikaler Strahl:



Vertikaler Check, horizontaler Strahl:



Entities

Die Formel für die Berechnung der x-Position einer Entity im Viewport wird mit folgender Formel berechnet:

$$x_{Fenster} = \frac{tan^{-1}\frac{\Delta y_{entity,player}}{\Delta x_{entity,player}} + \alpha_{player} + \beta_{player}}{\beta_{player}} * Fensterbreite$$

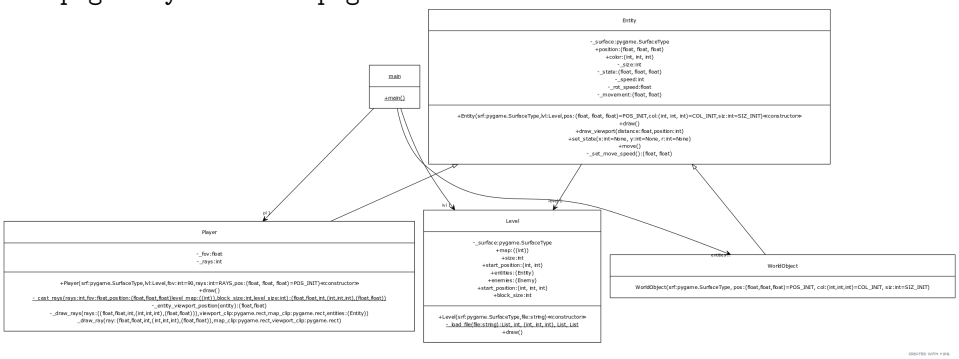
α_{player} ist die Blickrichtung des Spielers, β_{player} das Blickfeld.

Einige Korrekturen werden dabei noch vorgenommen, um die Position der Entity auf dem Bildschirm nicht umherspringen zu lassen.

Zuerst werden die Strahlen hinter der Entity angezeigt, dann die Entity selbst und zuletzt die Strahlen vor der Entity.

4 Programmaufbau

Die UML-Diagramme können in voller Größe im Git-Repo in `dokumentation/img` gefunden werden. Die relevanten Dateien sind `yuml-1.png`, `yuml-2.png`, `yuml-3.png`, `yuml-4.png` und `yuml-final.png`.



`Main` enthält die main-funktion: Sie ist für das Setup am Anfang verantwortlich und enthält die Gameloop, die Spielereingaben etc. verarbeitet.

Für `Player` wäre der Name Camera wahrscheinlich insgesamt angemessener. Sie ist zwar auch für Die Bewegung des Spielers verantwortlich, enthält aber den gesamten Anzeige-Code für den Viewport.

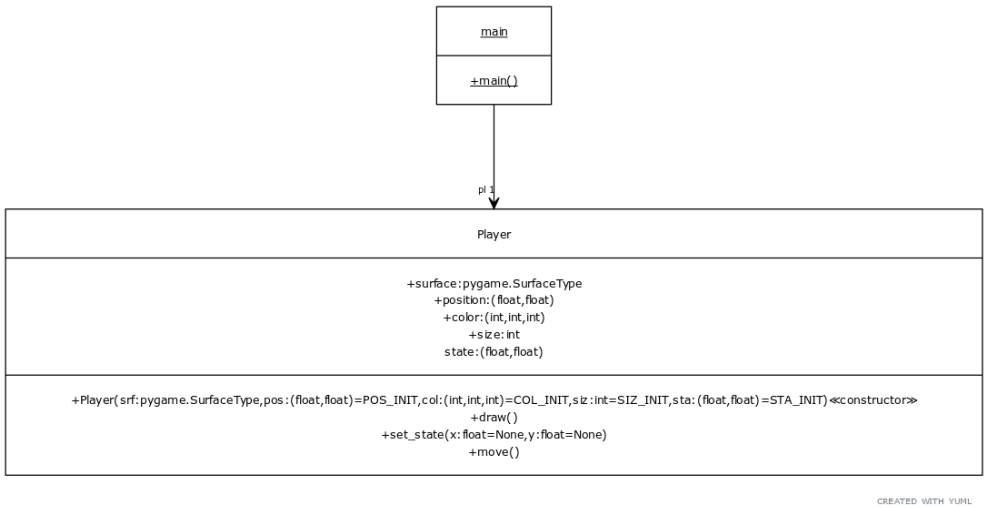
`Entity` enthält den tatsächlichen Bewegungscode und Code um sich selbst anzuzeigen.

`WorldObject` ist für bewegungslose Dinge, mit denen der Spieler nicht interagiert.

`Level` enthält die Daten der Welt unde die Kartenanzeigefunktion.

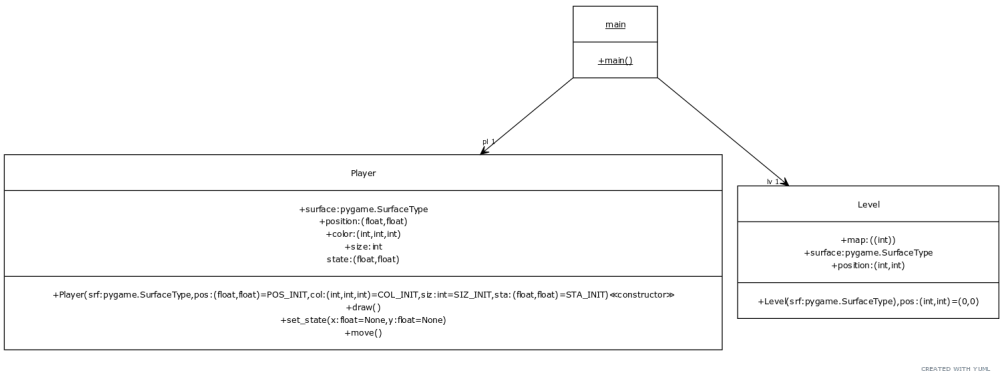
5 Entwicklungsprozess

5.1 Stand 18.11.2021



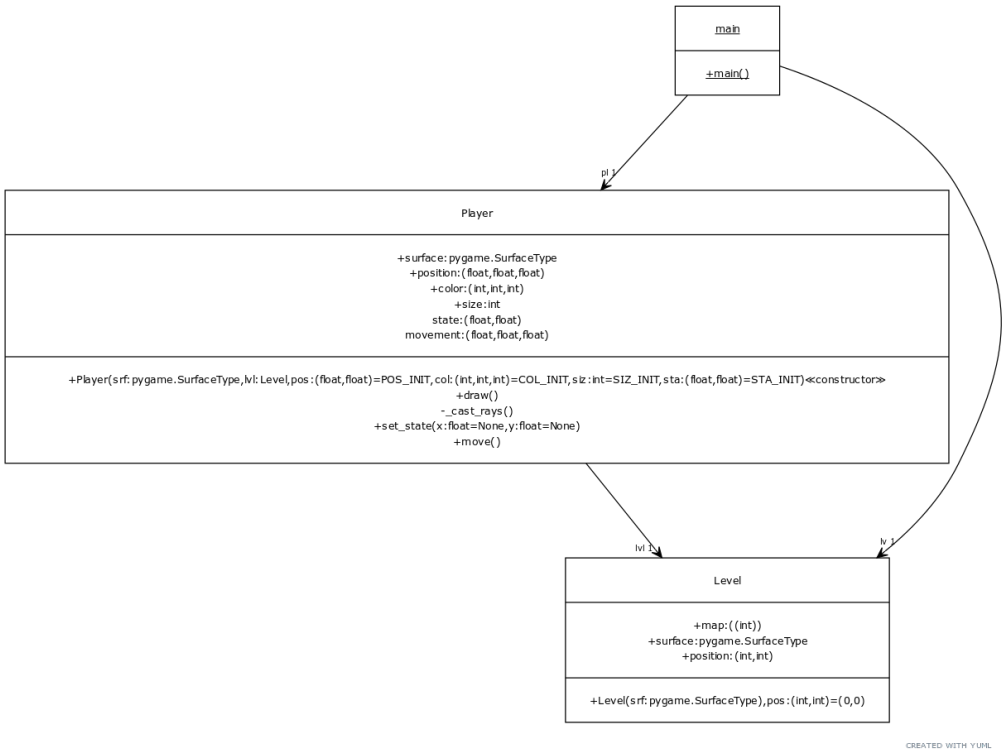
Die erste halbwegs funktionierende Version: Ein gelber Punkt bewegt sich auf einem grauen Bildschirm

5.2 Stand 23.11.2021



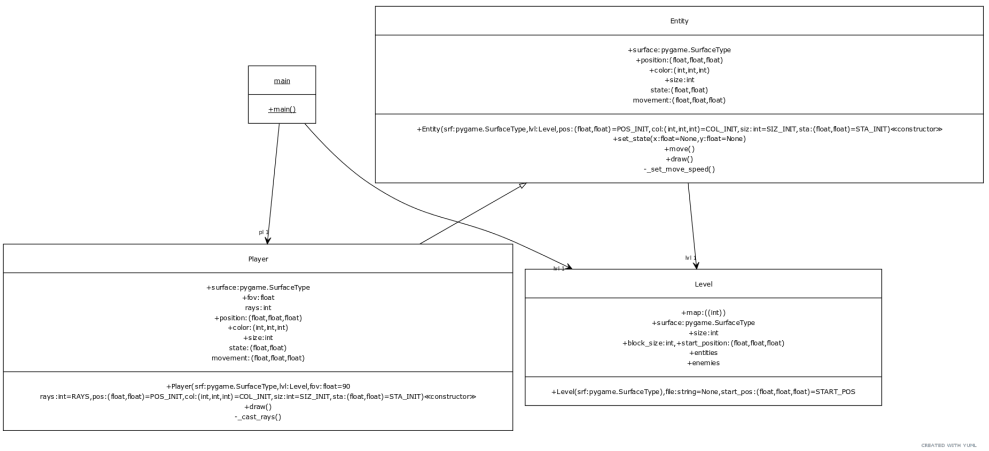
Nun wird eine Karte auf der linken Hälfte des Fensters angezeigt.

5.3 Stand 02.12.2021



Das Bewegungssystem wurde fast vollständig erneuert, damit sich der Spieler drehen kann und die erste Version Raycasting wird nun auf der Karte angezeigt.

5.4 Stand 09.12.2021



In der rechten Hälfte des Fensters wird jetzt der First-Person Viewport angezeigt, ausserdem unterstützt das Programm jetzt Command-line Argumente.

5.5 Stand 07.01.2022

Dies ist die finale Version, sie entspricht dem UML-Diagramm in Abschnitt 4.

Jetzt können auch Objekte in der Karte und im Viewport angezeigt werden, zusätzlich gibt es sehr rudimentäre Kollision.

6 Steuerung

6.1 Start und Command-Line Argumente

Gestartet wird das Spiel mit `py main.py` bzw. `python3 main.py`.

- `-h --help` zeigt die CLI Argumente und beendet das Programm.
- `-l --level` lädt das angegebene Level oder die angegebene Level Datei.
- `--fov` ändert den Blickwinkel (angegeben in Grad) Standartwert ist 90°.
- `--rays` ändert die horizontale Auflösung (Anzahl der gesendeten Strahlen) Standartwert ist 90. Höhere Werte können die Leistung beeinträchtigen.

6.2 Levelerstellung

Level werden im JSON-Format gespeichert.

- `"map": [[int]]` Die Map: 1 entspricht einer Wand, 0 Leerraum. Die Map muss quadratisch sein (ansonsten crasht das Programm)
- `"size": int` Die Größe der Map. Muss dem tatsächlichen Wert entsprechen.
- `"start_pos": [x: int, y: int, r: int]` Die Startposition des Spielers. `x` und `y` sind Werte zwischen 0 und 512, sie geben die Position in Pixeln an. `r` ist zwischen 0 und 360 und ist die Drehung in Grad.
- `"entities": [], "enemies": []` Enthalten aktuell keine Werte und werden für zukünftigen Gebrauch freigehalten.

6.3 Bewegung

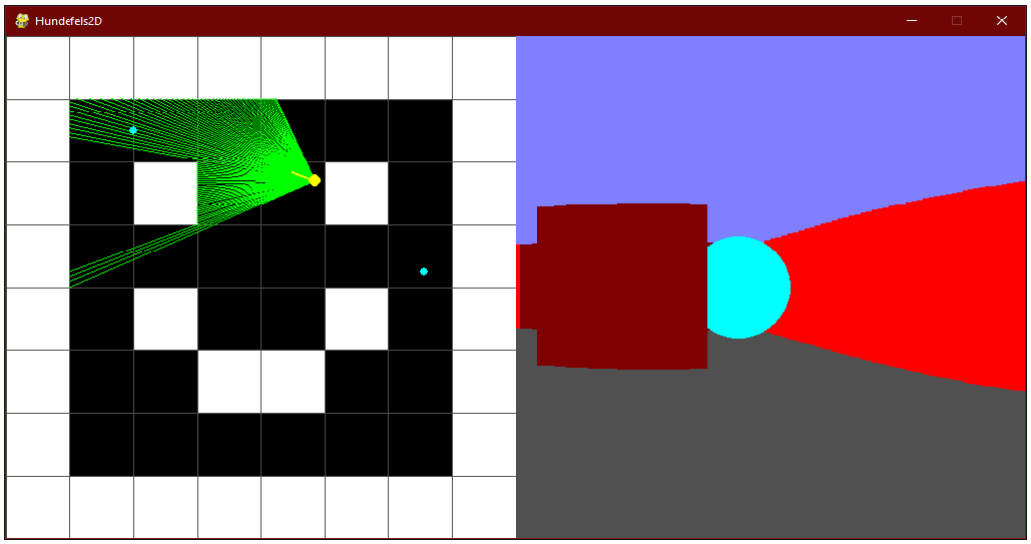
Translation (Laufen)

- Vorwärts: 'W'
- Links: 'A'
- Rückwärts: 'S'
- Rechts: 'D'

Rotation

- Links: linke Pfeiltaste (←)
- Rechts: rechte Pfeiltaste (→)

6.4 UI



Das Anzeigefenster ist 1024 auf 512 Pixel groß.

Die linke Hälfte enthält die Kartenansicht. Der Spieler wird mit einem gelben Punkt darestellt, Entities mit (standartmässig) blauen Punkten.

Die rechte Hälfte des Fensters ist der First-Person Viewport.

7 Quellen

- 3DSage: "Make Your Own Raycaster Part 1" `https://youtu.be/gYRrGTC7GtA`
  Quellcode verfügbar unter `https://github.com/3DSage/OpenGL-Raycaster\_v1`
  Für die Raycasting-Logik (Inhalt der Funktion `_cast_rays()` in `player.py`)
- Pygame tutorial: `https://www.pygame.org/docs/tut/MakeGames.html`