"LEAFY HOLLOWS"

Die schriftliche Ausarbeitung zur Klausurersatzleistung

Jakob Roehrborn
Robert-Havemann-Gymnasium
29.11.2023

Jakob Röhrborn
Robert-Havemann-Gymnasium
29.11.2023

Inhaltsverzeichnis

Jakob Röhrborn
Robert-Havemann-Gymnasium
29.11.2023

1. Einleitung

Als Klausurersatzleistung für die zweite Klausur in Q3 sollten wir ein Spiel in Python entwickeln. Es gab keine weiteren konkreten Vorgaben, weshalb wir uns dafür entschieden haben, ein Spiel zu programmieren, welches Spannend ist, einen hohen Wiederspielwert besitzt und somit möglicherweise auch nach Abschluss des Projekts veröffentlicht werden könnte. Um dies zu erreichen, sind viele Vorüberlegungen zum Spiel nötig. Dadurch war uns von Anfang an klar, dass dieses Projekt einen Umfang erreichen würde, welcher für uns beide neu wäre. Da wir für das Projekt fast ein halbes Jahr Zeit hatten, bot uns dies auch viele Möglichkeiten, um unsere eigenen Fähigkeiten zu verbessern und einen neuen Meilenstein in der unserer Entwicklung von Spielen zu erreichen. Dies bedeutete allerdings auch, dass wir eine persönliche Sicht auf das Spiel hatten, welche uns auch veranlasst hat sehr viel unserer Freizeit in dieses Projekt zu investieren

1.1. Motivation

Da wir dieses Projekt auch aus persönlichem Interesse verfolgt haben, fiel es uns generell leicht, uns für die Weiterarbeit am Projekt zu motivieren und neue Systeme, Spielmechaniken und Verbesserungen zu implementieren. Trotz diesem Ansporn hat uns das Projekt immer wieder aufs Neue gefordert, auch da unsere konkreten Vorstellungen teilweise zu ambitioniert waren und der Umfang des Projekts größer wurde, als alles, was wir bisher selbst programmiert haben, und uns somit auch immer wieder gezwungen erneut eine Motivation für die Weiterarbeit zu finden.

2. Ziele

Das Spiel sollte ein 2D Action-Adventure-Roguelite werden, bei welchem wir uns unteranderem an den Spielen „Noita“, „Magicite“ und „Rain World“ orientieren wollten. Dabei wollten wir einen Fokus auf eine spannende, physikbasierte und fesselnde Spielmechanik legen. Auch die partikelbasierte Simulation von Fluiden wäre ein interessanter Baustein gewesen, welcher das Spielerlebnis positiv beeinflusst hätte.

2.1. Grober Zeitplan

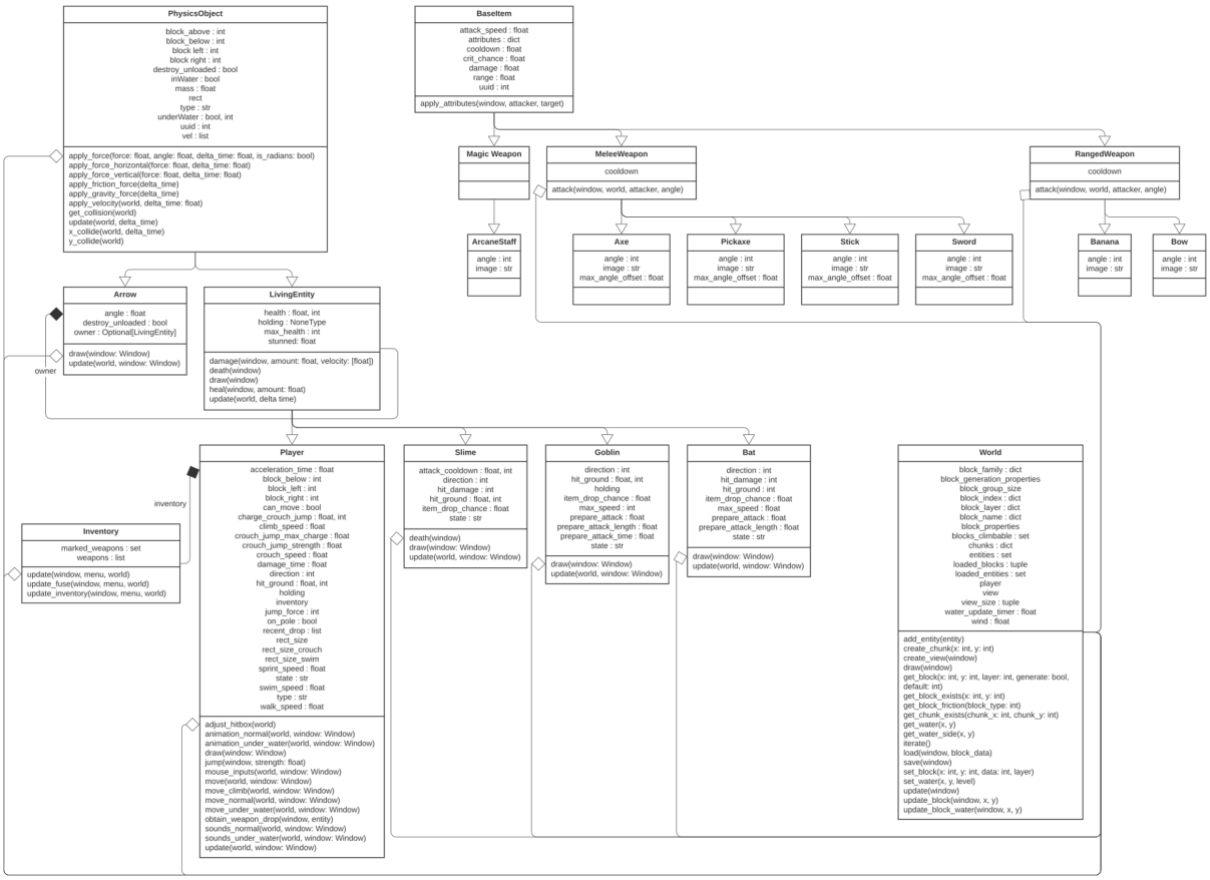
Ziel	*Geplante Erste Fertigstellung*	*Erste Fertigstellung*
Basisfunktionalitäten (z.B. öffnen eines Fensters / Eingaberegistrierung)	< 14.06.	31.05.
Physiksystem	14.06.	13.06.
Grundlegendes Spielerbewegungssystem	19.6.	11.06.
Partikelsystem	30.6.	29.08.
Weltgeneration	13.07.	29.06.
Gegner	25.8	11.11.
Gegenstände	25.8.	22.11.
Grafikverbesserungen (Sprites, Licht und Schatten) und Geräusche	25.8. – 28.9.	25.8

Jakob Röhrborn
Robert-Havemann-Gymnasium
29.11.2023

3. Ergebnis

Um zu beurteilen, inwiefern wir unsere Ziele erreicht haben, ist es auch notwendig, einen Blick auf die grundlegende Funktionsweise von bestimmten Systemen zu werfen, da diese das Fundament des endgültigen Spielgefühls bilden. So lässt sich zum Beispiel unter anderem auch unter Umständen die maßgebliche Entwicklungsmentalität erst daran erkennen. Natürlich ist der wichtigste Faktor für eine endgültige Beurteilung das tatsächliche Spielgefühl, sowie die Implementierung der innovativen Spielmechaniken, diese lassen sich allerdings in einer kurzen schriftlichen Ausarbeitung nicht analysieren und müssen bei der tatsächlichen Präsentation des Spiels beurteilt werden. Deshalb werden im Folgenden der Aufbau der Klassen, welche bei einer Objektorientierten Programmierung die Grundlage für die Spielmechaniken bilden, der konkrete, wenn auch stark versimpelte, Aufbau jener Klasse, welche die physikalischen Interaktionen regelt, sowie das Arbeitsprinzip einer Klasse, welche für die Wegfindung der Gegner zuständig ist, genauer erläutert.

3.1. Klassenbeziehungen



Eine Methode, um einen Überblick über den Aufbau eines komplexen Objektorientierten Programmes zu erlangen, ist die Darstellung der Beziehungen zwischen den wichtigsten Klassen in einem Klassendiagramm. In solch einem Diagramm lässt sich der Grundlegende Aufbau des Programms relativ einfach ablesen, da die Verbindungen

zwischen den Klassen visuell dargelegt werden und somit auch die Hierarchie der Klassen, falls sie vorhanden ist, sichtbar wird.

Dieses Diagramm zeigt die Beziehungen zwischen den Subklassen im Spiellogikordner („main\scripts\game“; Stand: 26.11.2023, 13:54 Uhr) und deren Superklassen. Die Beziehungen zwischen den Klassen werden mit gerichteten Pfeilen dargestellt. Vererbung wird durch Pfeile mit einer Dreiecksspitze dargestellt, wobei der Pfeil von der Super- zur Subklasse verläuft. Pfeile mit einer schwarzen Raute als Spitze zeigen an, dass eine Klasse als Attribut einer anderen verwendet wird, wobei der Text am Pfeil den Namen des Attributs angibt. Pfeile mit einer weißen Raute als Spitze zeigen an, dass eine Klasse einer Methode einer anderen Klasse als Parameter übergeben wird. Es ist deutlich zu erkennen, dass es drei relevante Basisklassen gibt („PhysicsObject“, „BaseItem“ und „World“), von welchen die restlichen Klassen im Spiellogikordner erben. So ist z.B. der Spieler eine Subklasse einer „LivingEntity“, welche wiederrum eine Subklasse der Basisklasse „PhysicsObject“ ist.

3.2. PhysicsObject

Eine der wichtigsten Klassen dieses Spiel ist die Klasse „PhysicsObject“. Wie man bereits im Klassendiagramm sehen kann, basieren die wichtigsten Akteure wie z.B. alle „Lebewesen“, insbesondere der Spieler, grundlegend auf dieser Klasse. Da sie nur das Fundament für andere Klassen darstellt, sind die Attribute und Methoden generell gehalten und ermöglichen viele grundlegendste Funktionen. Die Klasse „PhysicsObject“ regelt vor allem, wie sich Objekte mit einer Geschwindigkeit verhalten und wie diese Geschwindigkeit verändert wird. Der Aufbau der Klasse ist dabei den realen physikalischen Umständen nachempfunden, was auch dadurch bemerkbar wird, dass die Klasse grundsätzlich nur Krafteinwirkungen explizit akzeptiert und keine Methode für eine direkte, und somit sofortige, Änderung der Geschwindigkeit, was eine unendliche Beschleunigung voraussetzen würde, oder Position, was eine unendliche Geschwindigkeit (Teleportation) voraussetzen würde, besitzt. Außerdem erkennt und verarbeitet die Klasse Kollisionen mit der Umgebung. Eine sehr stark gekürzte und versimpelte Version dieser Klasse liegt hier im Folgenden mit Kommentaren zur Verständlichkeit vor:

```python
from scripts.utility import geometry
from scripts.constants import *
import math

class PhysicsObject:
    def __init__(self, mass: float, position: [float], size: [float]):
        self.mass: float = mass
        # custom rect class (similar to pygame's) with float precision
        self.rect: geometry.Rect = geometry.Rect(*position, *size)
        self.vel: [float] = [0.0, 0.0]

        self.inWater: bool = False
        self.underWater: bool = False

    # Applies force to the object
    def apply_force(self, force: float, angle: float, delta_time: float):
```

```
        if self.underWater:
            force /= 3

        acceleration = force / self.mass

        # Angle in degrees; 0 equals right; counterclockwise
        self.vel[0] += math.cos(angle) * acceleration * delta_time
        self.vel[1] += math.sin(angle) * acceleration * delta_time

    # Applies velocity to the object.
    def apply_velocity(self, world, delta_time: float):
        last_position = self.rect.center
        self.rect.x += self.vel[0] * delta_time
        self.x_collide(world)

        self.rect.y += self.vel[1] * delta_time
        self.y_collide(world)

        # Reset object position
        if self.get_collision(world):
            self.rect.center = last_position
            self.vel = [0, 0]


    def get_collision(self, world):

        for x in range(math.floor(self.rect.left), math.ceil(self.rect.right)):
            for y in range(math.floor(self.rect.top), math.ceil(self.rect.bottom)):
                if world.get_block(x, y):
                    return True
        return False

    # Applies gravity force to the object.
    def apply_gravity_force(self, delta_time):
        if self.inWater:
            gravity = PHYSICS_GRAVITY_CONSTANT_WATER
        else:
            gravity = PHYSICS_GRAVITY_CONSTANT

        self.apply_force(gravity * self.mass, 270, delta_time)

    # Resolves collisions on the y-axis; similar for x (omitted
    def y_collide(self, world):)
        for x in range(math.floor(self.rect.left), math.ceil(self.rect.right)):
            for y in range(math.floor(self.rect.top), math.ceil(self.rect.bottom)):

               # check if there's a block for every "tile" the player is touching
                if world.get_block(x, y):
                    if self.vel[1] > 0:
                        self.rect.bottom = y
                    if self.vel[1] < 0:
                        self.rect.top = y + 1
                    self.vel[1] = 0

    def update(self, world, delta_time):
        self.apply_velocity(world, delta_time)  # Add velocity to position
```

```
        self.apply_gravity_force(delta_time)    # Apply gravity and friction
```

3.3. Pathfinding

Eine notwendige Mechanik, um zu garantieren, dass sich das Verhalten der Gegner möglichst natürlich anfühlt, ist eine Wegfindungsmechanik (aus dem Englischen: „Pathfinding“). Ein Mensch sieht oft meist auf den ersten Blick, welcher Weg optimal ist. Diese Fähigkeit hat ein Computerprogramm nicht von allein, und somit ist die Implementierung eines Systems, welches dieses Vollbringt, zwingend notwendig. In diesem Spiel wird dies durch eine selbstprogrammierte Implementierung von einem als A* (auch AStar genannt) bekanntem Algorithmus, welcher eine Verbesserung des Dijkstra-Algorithmus ist. Dieser informierte Suchalgorithmus ist in der Lage immer den optimalen Weg durch einen Graph aus Knoten zu finden. Um den Suchaufwand zu verringern, besitzt dieser Algorithmus eine „heuristische“ Schätzfunktion, welche es dem Algorithmus ermöglicht abzuschätzen, wie nah ein noch nicht besuchter Knoten am Ziel liegt. Diese Eigenschaften machen es optimal für einen Wegfindungsalgorithmus in 2D-Spielen und findet daher auch Anwendung in unserem Spiel.

4. Fazit

Die Entwicklung eines so umfangreichen Spiels, sowie das im Team arbeiten, waren für uns beide neue und ungewohnte Erfahrungen. Wir mussten uns neue Kompetenzen erarbeiten und viel Zeit in die gemeinsame Planung investieren, damit das Spiel unseren Anforderungen gerecht wird. Dabei viel uns Anfangs vor allem die Kommunikation über den konkreten Aufbau von bestimmten Systemen schwer, da es für uns beide gewöhnungsbedürftig war, mit Code zu arbeiten, welchen wir nicht selbst geschrieben hatten. Nachdem wir diese Anfänglichen Schwierigkeiten allerdings überwunden hatten, war es immer wieder interessant zu sehen, was der jeweils Andere geschrieben hatte. Wir machten danach viele Fortschritte und hatten auch immer neue Ideen für Spielmechaniken, wodurch sich der Projektumfang immer weiter vergrößerte. Allerdings gab es auch einige Punkte, die wir nicht mehr implementieren konnten, sei es, weil die Zeit bis zur Abgabe zu wenig war oder weil dieser Punkt nicht mehr zu unserer Vorstellung vom Spiel passte bzw. nicht mit den anderen Spielmechaniken in Harmonie war. Ein Beispiel dafür war die bereits in den Zielen erwähnte Simulation von Fluiden, welche in der Finalen Version nur zu Teilen vertreten ist. Auch wenn uns solche nicht erfüllten Ziele manchmal an der Qualität unseres Spiels zweifeln ließen, kamen wir meist zu dem Schluss, dass es eine gute Entscheidung war dieses Ziel nicht weiter zu verfolgen.

Insgesamt kann ich sagen, dass ich froh bin dieses Projekt aus eigener Motivation verfolgt zu haben, da es sowohl meine Programmierfähigkeiten als auch meine Kompetenzen im Bereich der Teamarbeit verbessert hat und ich dabei auch Spaß hatte. Da ich mit dem Endprodukt dazu auch noch zufrieden bin, ist das Projekt für mich definitiv ein Erfolg.