Übungsblatt 1: 3D-Labyrinth

1. Wahl der Engine und Eingabeformat

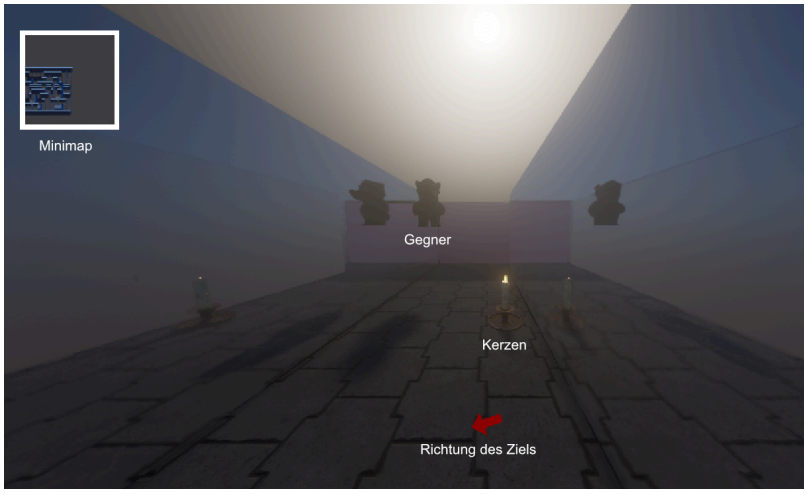
Ich beschloss recht früh, die Unity Engine (mit der Ray-Tracing Pipeline HDRP) für die Bewältigung dieser Aufgabe zu nutzen. Dafür sprachen für mich vor allem die folgenden drei Punkte:

- Neben der Unreal Engine ist die Unity Engine die am weitesten verbreitete Game Engine, welche Ray-Tracing (ohne großen Aufwand, z. B. eine Selbstimplementierung als Compute Shader) unterstützt. Durch die große Verbreitung ist die Engine gut dokumentiert und es gibt viele Tutorials, die den Einstieg erleichtern.
- C# bietet gegenüber C++ (Unreal Engine) mehr Comfort und etwas einsteigerfreundliche Programmierung.
- Obwohl die Unreal Engine zur Zeit das Flagschiff in Sachen Grafikqualität darstellt (wenn auch nicht unumstritten[1]), ist die Unity Engine deutlich leichtgewichtiger, was vor allem für ein kleines Projekt wie dieses sehr angenehm ist.

Zur Eingabe des Levels/Labyrinths verwende ich eine normale Textdatei. Mein Ziel war es, ASCII-Art zu verwenden, um den Grundriss des Labyrinths zu bestimmen. Daher habe ich für die meistverwendeten Zeichen (`-|+/\#@%&*.'()[]^<>=_~xo0v`) simple 3D-Modelle erstellt, welche an den entsprechenden Stellen in der Szene platziert werden. In meinem Projekt sind einige Beispiele vom ASCII Art Archive[2] enthalten, welche im Unity Editor beim `LevelGenerator` ausgewählt werden können.

2. Das Spiel

Auf Basis einer beliebigen ASCII-Textdatei wird ein Labyrinth mit zufällig platziertem Zielblock generiert. Der Spieler steuert eine Figur, die sich durch das Labyrinth bewegen kann. Das Ziel des Spiels ist es, das Ziel zu erreichen, bevor der Gegner den Spieler erreicht. Dazu stehen dem Spieler neben einem grundlegenden Bewegungsarsenal (Laufen, Springen) eine Taschenlampe (Taste `F`), eine Minimap und ein Pfeil, der die Richtung des Ziels anzeigt, zur Verfügung. Aufgrund der Sprungmechanik wurden die Wände absichtlich niedrig genug gehalten, dass an manchen Stellen über sie gesprungen werden kann.

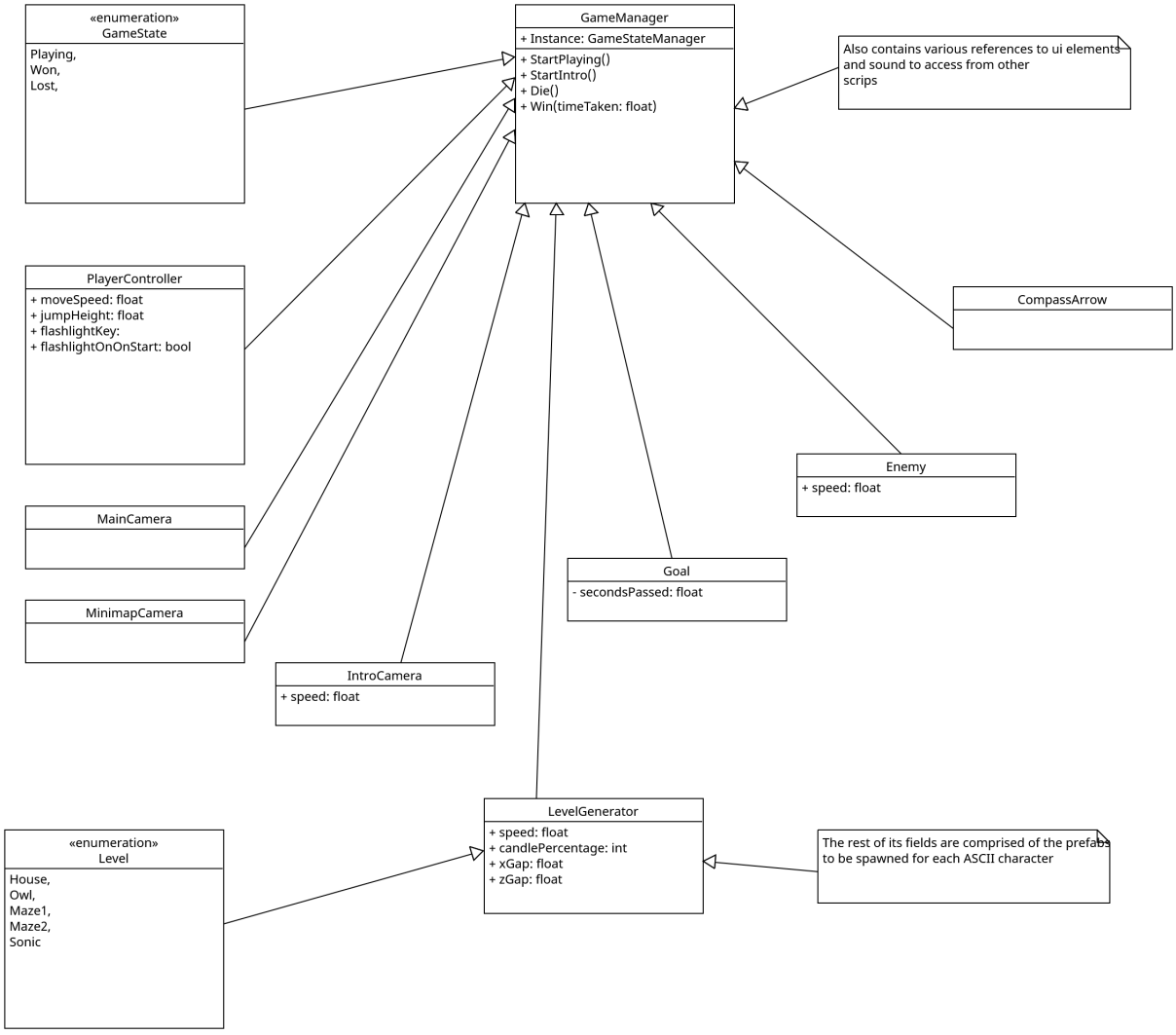


[1] siehe z. B. https://www.youtube.com/watch?v=lJu_DgCHfx4
[2] https://www.asciiart.eu

3. Umsetzung

Beim Start des Spiels wird der `LevelGenerator` aufgerufen, welcher die Textdatei einliest und die den Zeichen entsprechenden Prefabs (bestehend aus 3D-Modell, Material und Collider) an den richtigen Stellen in der Szene platziert. Ein Ziel (`Goal`) wird zufällig im Labyrinth platziert.

Nachdem das Level so aufgebaut wurde, ruft das `LevelGenerator` die `IntroCamera` auf, welche in einer Kamerafahrt das Labyrinth zeigt. Danach wird die `MainCamera` aktiviert und das Spiel in den Zustand „Playing“ versetzt, wodurch der PlayerController und der Gegner aktiviert werden. Zur besseren Übersicht ist eine dritte Kamera dafür zuständig, eine Top-Down-Ansicht des Labyrinths als Minimap zu zeigen. Zusätzlich wird am unteren Bildschirmrand ein Pfeil (`CompassArrow`) angezeigt, welcher in Richtung des Ziels zeigt. Auch diese beiden Features werden erst vom `LevelGenerator` aktiviert, nachdem das Labyrinth generiert wurde.



Nachfolgend möchte ich die wichtigsten Klassen in meinem Code kurz erläutern. Hinweis: Das Leistungsoverlay ist eine fertige UI-Komponente, welche keinen Code meinerseits benötigt. Die Minimap funktioniert so, dass die orthographische Minimap-Kamera ihr Bild in eine RenderTexture (Ordner: `UI`) rendert, welcher dann in der UI als `RawImage` angezeigt wird. Auch hier ist kein zusätzlicher Code erforderlich.

3.1. GameManager (Singleton)

Wie bereits im UML-Diagramm in Abbildung Abbildung 2 zu sehen, ist der GameManager das „Hauptdrehkreuz“, welches alle wichtigen Referenzen verwaltet und objekt-/skriptübergreifenden Zugriff es sehr simpel macht. Der `GameManager` ist als Singleton implementiert, wodurch man sich sparen kann, den GameManager an jedes Objekt in der Szene zu übergeben. Er verwaltet zudem den Spielzustand, anhand dessen gewisse Objekte aktiviert oder deaktiviert werden. Dazu existieren die Funktionen `StartPlaying`, `Die` und `Win`.

3.2. LevelGenerator

Der LevelGenerator ist die erste Komponente, welche beim Start des Spiels laufen muss. Sie ließt zeilenweise eine Textdatei ein und überträgt Zeichen für Zeichen die Struktur der ASCII-Zeichen in den Aufbau eines Labyrinths in der Szene. Dabei werden die maximale x- und z-Koordinate gespeichert, um an zufälligen Positionen innerhalb der Labyrinth-Begrenzungen Kerzen, sowie das Ziel zu platzieren. Schließlich entscheidet sich der LevelGenerator zufällig für „Tag“ oder „Nacht“ und passt entsprechend die Intensität der Sonne an. Dadurch lassen sich die Ray-Tracing Effekte bei stärkerem direkten/indirekten Licht vergleichen. Nach dem der LevelGenerator die Struktur des Labyrinths erstellt hat, wechselt der Spielzustand zu `Intro`.

3.3. IntroCamera

Die `IntroCamera` ist dafür zuständig, das Labyrinth in einer Kamerafahrt zu zeigen. Von einem Startpunkt etwa 100 Einheiten über dem Labyrinth, wird die Kamera mittels `Slerp` langsam zum Startpunkt des Spielers bewegt. So kann der Spieler sich ein Bild von der Struktur des Labyrinths und der Zielposition machen, bevor das Spiel in den Zustand `Playing` wechselt und der Spieler selbst die Kontrolle übernehmen kann. Den Wechsel ruft `IntroCamera` über den `GameManager` auf, nachdem die Animation abgeschlossen ist.

3.4. PlayerController

Der `PlayerController` ist dafür zuständig, die Eingaben des Spielers zu verarbeiten und die Figur zu bewegen. Zusätzlich spielt er bei Bewegung Schrittgeräusche ab. Die Bedingungen zu Sieg oder Niederlage des aktuellen Levels sind in die `Goal` und `Enemy`-Klassen ausgelagert.

3.5. Enemy

Die Klasse `Enemy` umfasst die sehr rudimentäre KI des Gegners. Der Gegner richtet sich dabei bei jedem Update nach dem Spieler aus und bewegt sich mit einer konstanten Geschwindigkeit in die Richtung des Spielers. Dabei schwebt er durch Wände hindurch, wordurch aufwändiges Pathfinding nicht nötig ist. Ursprünglich war geplant, Unitys `NavMesh` zwecks Pathfinding zu verwenden, jedoch erfordert dies eine Vorberechnung des NavMesh, was in einem dynamischen Labyrinth, was erst zur Runtime entsteht, nicht möglich ist.

4. Abschließende Hinweise

Sämtliche im Spiel verwendeten Ressourcen (Modelle, Texturen, Sounds) sind in der Datei `Credits.md` aufgelistet.