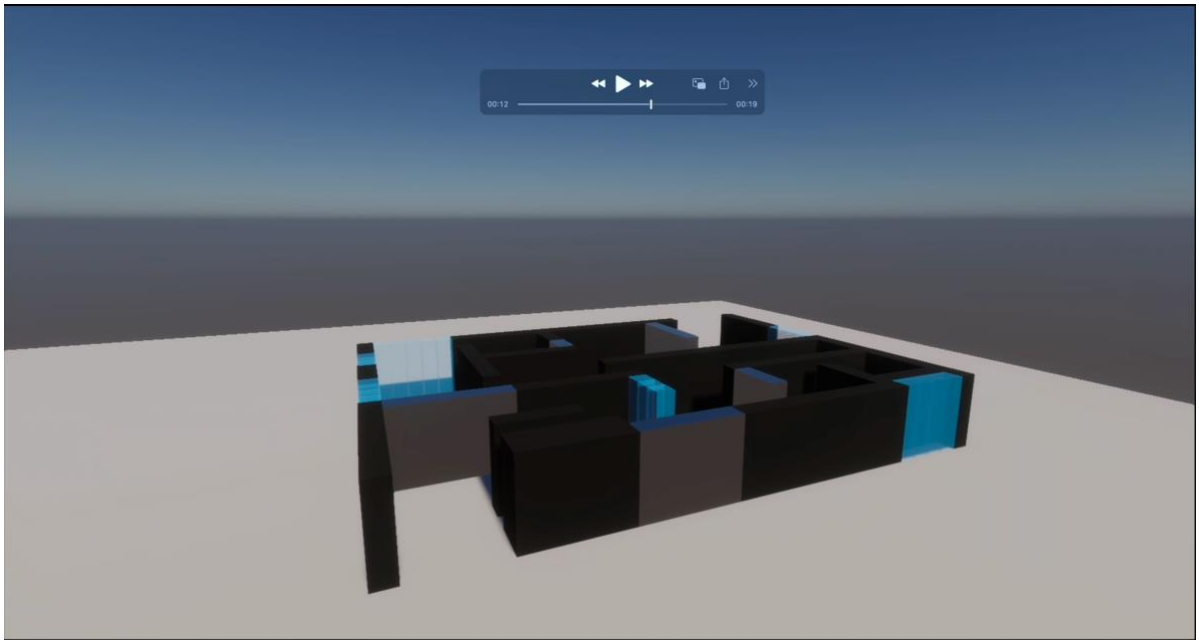
Bericht zur 3D-Labyrinth mittels Ray-Tracing

Narek Grigoryan
Matrikelnummer: 1547616
Kurs: Spielprogrammierung– Übung 1

Inhalt



Einleitung & Projektstart

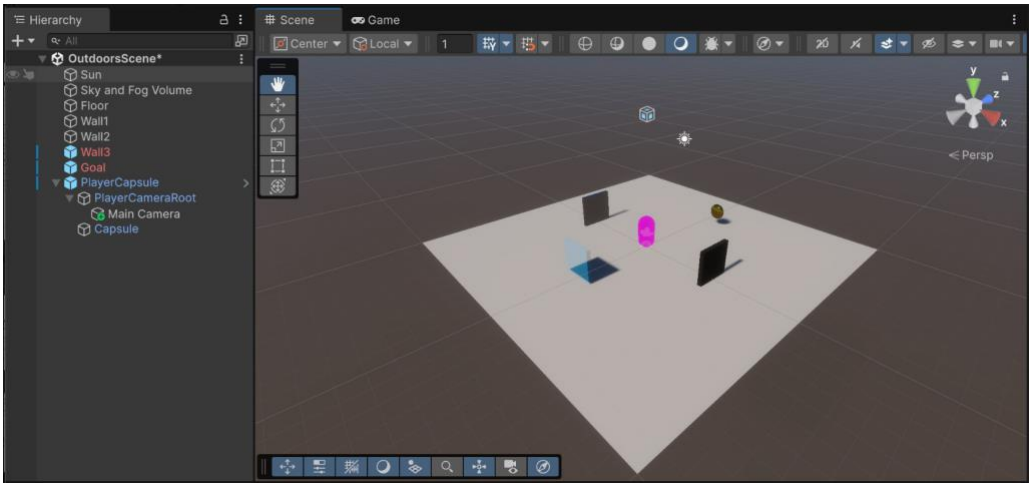
Im Rahmen der Übung 1 der Vorlesung „Spieleprogrammierung“ bestand die Aufgabe darin, ein begehbares 3D-Labyrinth zu erstellen, das mithilfe von Ray Tracing realistisch ausgeleuchtet ist und verschiedene Materialtypen wie reflektierende, absorbierende und transparente Oberflächen enthält. Die Interaktion mit der Szene sollte über eine First-Person-Steuerung möglich sein. Der Projektansatz war von Anfang an darauf ausgelegt, die Anforderungen schrittweise umzusetzen und eine realistische, moderne Spielumgebung zu schaffen.

Da ich zuvor noch keine tiefere Erfahrung mit hatte, habe ich mich zu Beginn der Aufgabe intensiv mit der Enginen auseinandergesetzt. Ich habe mich dazu entschieden, mein 3D-Labyrinth mit Unity umzusetzen, da die Engine eine breite Dokumentation, viele Community-Ressourcen und eine leistungsfähige HDRP-Pipeline für physikalisch korrektes Licht bietet. Besonders hilfreich waren dabei zwei Tutorials, auf die ich mich gestützt habe:

- Einführung in Unity HDRP mit Raytracing
- 3D First Person Projekt mit Materialien & Licht

Anhand dieser Tutorials habe ich mir Schritt für Schritt die wichtigsten Funktionen in Unity erschlossen. Das Projekt wurde auf einem Apple M4 Mac mit macOS eingerichtet. Die Unity-Version war 6000.1.4f1 (Unity 6). Die ersten Schritte bestanden darin, eine neue HDRP-Szene zu erstellen, wobei automatisch einige grundlegende Komponenten vorkonfiguriert wurden – z. B. Sky and Fog Volume, Sun und eine HDRP-Volume mit voreingestellter Lichtumgebung.

Daraufhin habe ich ein Bodenobjekt (Floor) über ein Plane erstellt und in der Szene zentriert platziert. In der Hierarchie habe ich dann über das Rechtsklickmenü 3D-Objekte hinzugefügt, um die Wände und das Zielobjekt zu erstellen. In einem frühen Stadium meiner Arbeit sah meine Szene folgendermaßen aus:



Man erkennt darauf:

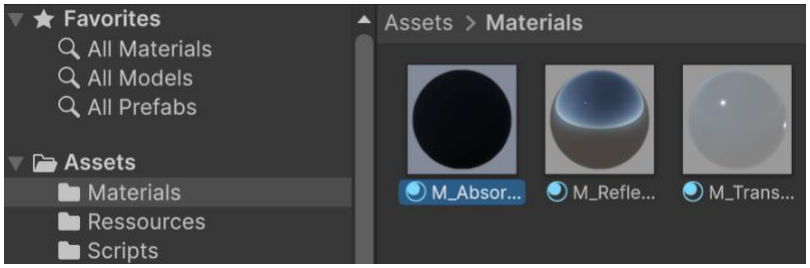
- Drei unterschiedliche Wände, jeweils mit reflektierendem, absorbierendem oder transparentem Material
- Ein Zielobjekt („Goal“)
- Einen First-Person-Controller („PlayerCapsule“)
- Eine einfache, begehbare Fläche (Floor)

Erstellung der Materialien

Um den Anforderungen an unterschiedliche physikalische Eigenschaften gerecht zu werden, habe ich im Projektverzeichnis unter Assets > Materials drei neue HDRP-Materialien erstellt:

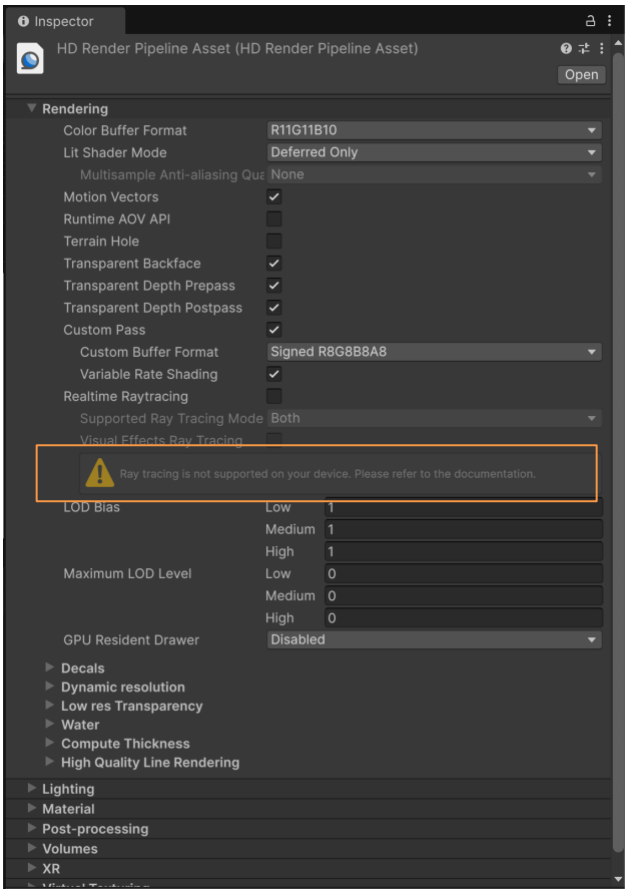
1. M_Absorbent – extrem dunkel, kein Glanz, niedrigste Smoothness
2. M_Reflective – metallisch glänzend, hohe Smoothness, aktiviertes SSR
3. M_Transparent – halbtransparent mit Blending Mode „Alpha“

Diese Materialien wurden jeweils über „Create > Material“ erzeugt und im Inspector mit den passenden Werten versehen.



Raytracing auf Apple M4

Ein zentraler Punkt der Aufgabenstellung war die Nutzung von Ray Tracing. Auf meinem Apple M4 Mac musste ich jedoch feststellen, dass Raytracing laut HDRP-Dokumentation nicht unterstützt wird. Dies wird auch im Unity Editor entsprechend angezeigt:



Trotzdem wurden alle anderen Anforderungen der Aufgabenstellung technisch korrekt umgesetzt:

- Die Materialien sind korrekt für Ray Tracing vorbereitet (z. B. „Receive SSR“, „Transparent Depth Prepass“).
- Die Lichter sind physikalisch korrekt als Punktlichter mit Volumetrics implementiert.
- Der Aufbau der Szene erfüllt die Struktur- und Lichtanforderungen, sodass der Gesamteindruck einem Ray-Traced-Setup sehr nahekommt.

Die Limitierung durch die Apple-Hardware war in diesem Fall technisch nicht umgehbar, wurde aber durch präzise Materialdefinition, realistisches Lichtverhalten und passende HDRP-Einstellungen ausgeglichen.

Implementierung

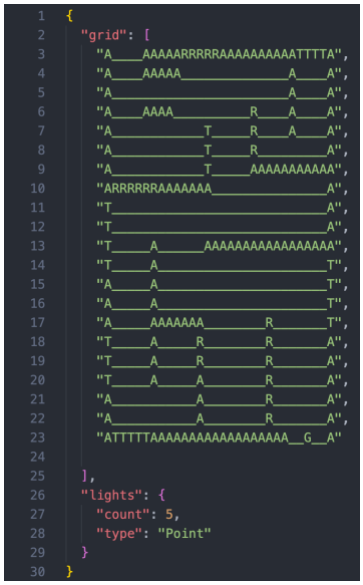
Automatisierte Levelgenerierung via JSON

Ein zentraler Bestandteil der Projektarbeit war die Ablösung des manuellen Aufbaus durch eine automatisierte Generierung des gesamten Labyrinths. Dazu habe ich mich entschieden, die komplette Struktur über eine JSON-Datei zu steuern. Diese Datei (labyrinth.json) enthält ein zweidimensionales Zeichenraster, das das Labyrinth beschreibt. Jedes Zeichen steht dabei für ein bestimmtes Objekt oder Verhalten in der Szene. Zusätzlich lassen sich in dieser Datei Lichteinstellungen wie Anzahl und Typ der Lichtquellen definieren.

Gridstruktur und Zeichenbedeutung

Die "grid"-Eigenschaft der JSON-Datei besteht aus einem Array von Strings, in dem jede Zeichenkette eine Zeile im Labyrinth darstellt. Insgesamt ergibt sich so ein klassisches 2D-Raster. Anhand der Zeichen lässt sich klar zuordnen, welches GameObject später erzeugt werden soll. Die verwendeten Symbole sind:

- "A" → Absorbierende Wand (dunkel, lichtschluckend)
- "R" → Reflektierende Wand (glänzend, spiegelnd)
- "T" → Transparente Wand (durchscheinend, Glas)
- "_" → Begehbarer Bereich
- "G" → Zielobjekt (END-Zelle)



Das Besondere an diesem Ansatz ist die Trennung von Logik und Inhalt: Die JSON-Datei bildet eine reine Datenstruktur, die über den Code des Spiels interpretiert wird. Dadurch lässt sich das Level beliebig anpassen.

LabyrinthManager.cs

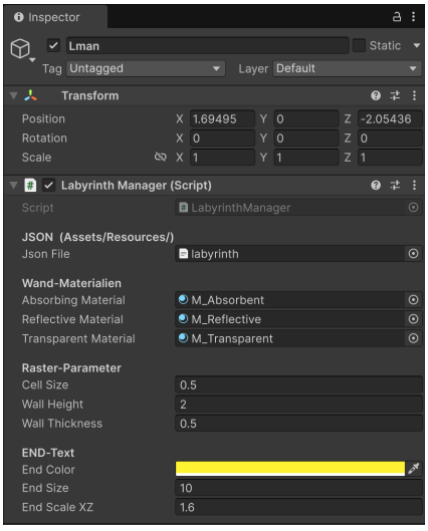
Die gesamte Verarbeitung der JSON-Daten übernimmt das zentrale Script LabyrinthManager.cs, das einem leeren GameObject namens Lman zugewiesen ist. Beim Start des Spiels lädt das Script die JSON-Datei (vom Typ TextAsset) und analysiert deren Inhalte. Danach werden alle Elemente der Szene dynamisch instanziiert.

Die wichtigsten Aufgaben, die dieses Script übernimmt, sind:

1. Initialisierung der Parameter über den Unity-Inspector (z. B. Cell-Größe, Materialien, Farben, etc.)
2. Erzeugung der Wände basierend auf dem Zeichenraster
3. Platzierung der Lichtquellen
4. Erstellung des Zielbereichs („END“)
5. Zuweisung von Triggern zur Erfolgserkennung

Der Ablauf ist dabei klar gegliedert: In der Start()-Methode erfolgt zuerst der JSON-Import, anschließend werden über verschachtelte Schleifen die Wände gebaut und abschließend das Licht und das Ziel generiert. Bei jedem Wandzeichen wird ein PrimitiveType.Cube mit den passenden Maßen erstellt und mit dem entsprechenden Material ausgestattet. Die Objekte werden auf Basis der cellSize-Variable im Raum korrekt positioniert.

Durch die Konfigurationsmöglichkeiten im Inspector (siehe Screenshot unten) lassen sich verschiedene Materialverhalten, Größenparameter und sogar das Farbschema des „END“-Ziels direkt in der Szene anpassen, ohne dass der Code verändert werden muss.



Intelligente Lichtplatzierung

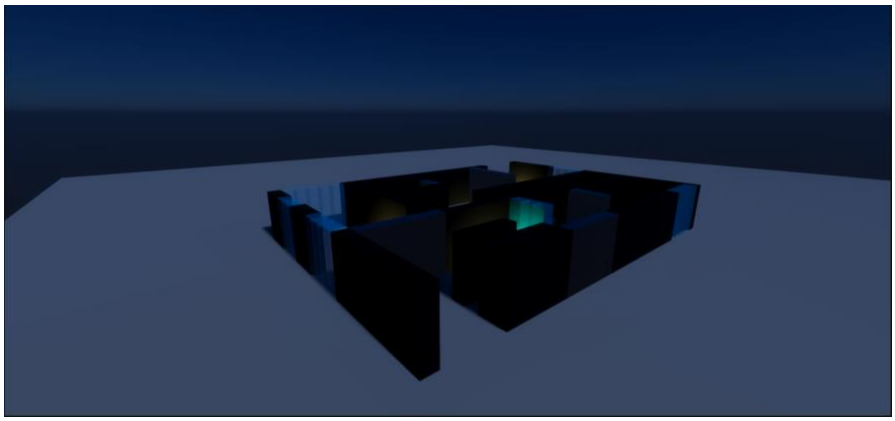
Ein weiteres Highlight des Projekts ist die durchdachte Platzierung von Lichtquellen: In der JSON-Datei wird festgelegt, wie viele Lichter erzeugt werden sollen und welchen Typ sie haben. In der Praxis habe ich mich für punktförmige HDRP-Lichter entschieden, die direkt im Unity-Script als LightType.Point instanziiert werden.

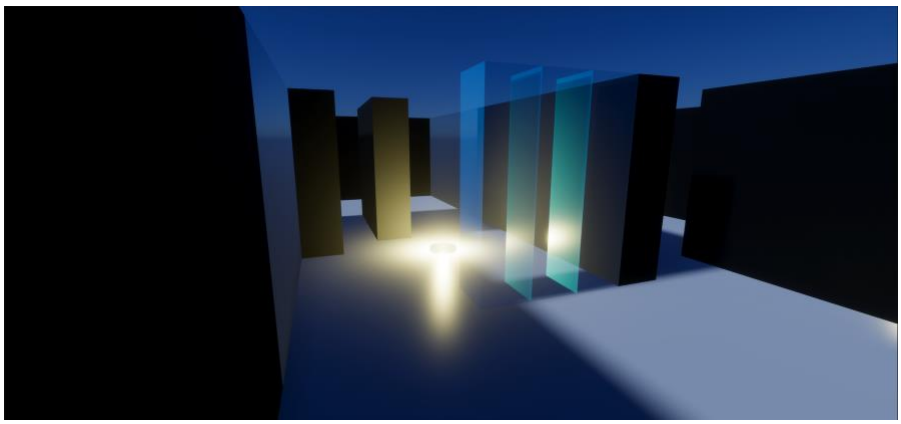
Besonderes Augenmerk wurde darauf gelegt, dass die Lichter nicht willkürlich im Labyrinth platziert werden. Über eine Hilfsfunktion namens IsCentral() prüft das Script, ob das aktuelle Feld vollständig von freien Feldern umgeben ist – also quasi in der Mitte eines Ganges liegt. Nur dann wird dort ein Licht erzeugt. Dadurch entstehen realistische Lichtinseln in zentralen Passagen, ohne dass die Wände überleuchtet werden.

Jede Lichtquelle besteht aus:

- einem volumetrischen Punktlicht
- einem leuchtenden Zylinder (Disk), der optisch als „Lampe“ dient

Die Emission wird über ein spezielles HDRP-Material erzeugt, das eine starke Leuchtkraft (_EmissiveColor) besitzt. So entsteht der Eindruck von eingelassenen Bodenleuchten, die nicht nur Licht spenden, sondern auch visuell zur Atmosphäre beitragen.





Zielerkennung mit „END“-Bereich

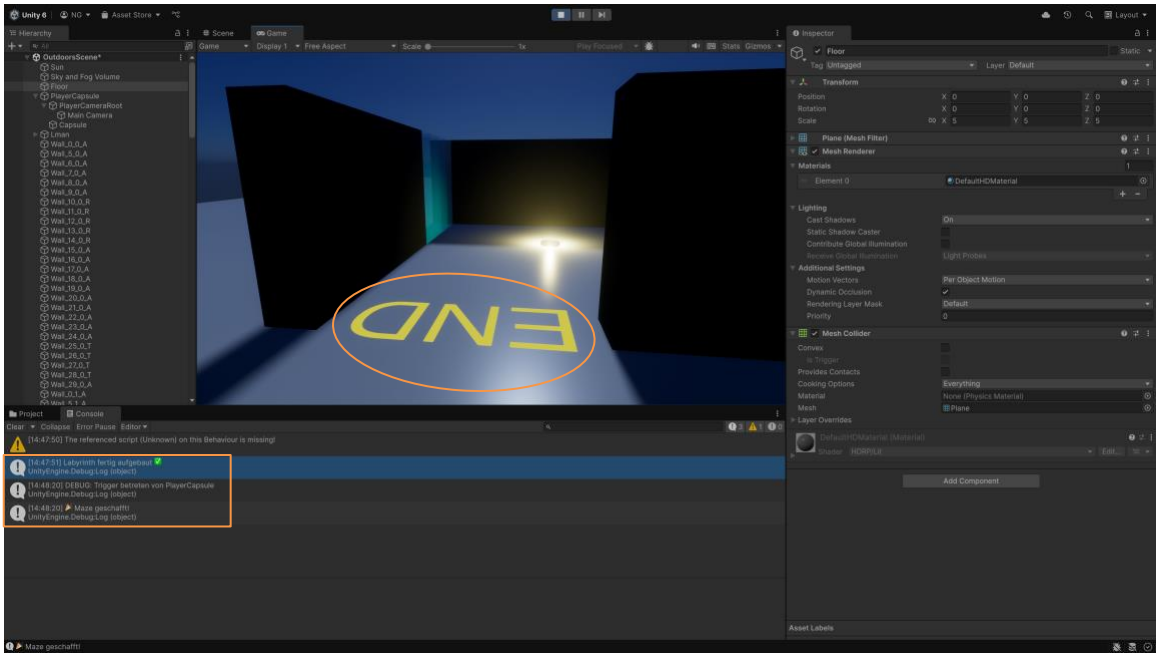
Um dem Labyrinth eine Spielmechanik zu verleihen, wurde ein Zielpunkt integriert, der durch den Buchstaben "G" im Grid markiert wird. Erkennt das Script dieses Zeichen, wird über die Methode SpawnEnd() ein flacher, gelber 3D-Text mit dem Wort END erzeugt. Dieser liegt direkt auf dem Boden der Szene.

Zusätzlich wird dem Text ein unsichtbarer BoxCollider (isTrigger) zugewiesen, sodass Unity erkennen kann, wenn der Spieler diesen Bereich betritt.

EndTrigger.cs

Um auf das Betreten des Ziels zu reagieren, wurde ein einfaches, aber effektives Script namens EndTrigger.cs geschrieben. Dieses wird dem END-GameObject automatisch zugewiesen. Die Aufgabe: Wenn der Spieler – also ein GameObject mit dem Tag "Player" – den Trigger betritt, wird eine Erfolgsmeldung auf der Konsole ausgegeben.

Das Script verwendet OnTriggerEnter(Collider other) und prüft explizit, ob der andere Collider der Spieler ist. Ein bool done sorgt dafür, dass die Nachricht nur einmalig erscheint.



So wird klar signalisiert, dass das Ziel erreicht wurde – eine einfache, aber effektive Rückmeldung im Kontext dieser Übung.

Fazit und Rückblick

Die Umsetzung des 3D-Labyrinths mit Unity und HDRP war für mich eine sehr lehrreiche Erfahrung, insbesondere weil ich vor diesem Projekt kaum praktische Erfahrung mit Unity hatte. Viele grundlegende Konzepte wie Materials, Lighting oder Trigger-Events habe ich mir Schritt für Schritt mithilfe von Tutorials und Experimentieren erarbeitet.

Selbstkritisch muss ich sagen, dass ich stellenweise zu viel Zeit damit verbracht habe, Fehler durch Ausprobieren zu lösen, statt systematisch in der Dokumentation oder im Debugging vorzugehen. Gerade bei HDRP war es nicht immer leicht, herauszufinden, warum etwas nicht sichtbar war oder warum die Lichter nicht korrekt wirkten – vor allem, weil Ray Tracing auf meinem Mac M4 nicht unterstützt wird. Trotzdem habe ich durch Workarounds mit punktuellem Licht, volumetrischen Effekten und gezielten Materialeinstellungen erreicht, dass der visuelle Eindruck den Anforderungen gerecht wird.