Prof. Dr. Sturm, Spieleprogrammierung, Sommer 2025, Übungsblatt 1
Fabian Vecellio del Monego, 1442314

Ergebnisbericht

1. Einleitung

In dieser Arbeit beschreibe ich die Umsetzung eines dynamisch generierten 3D-Labyrinths in der Unreal Engine 5 (UE5), das zur Laufzeit vollständig aus einer externen ASCII-Textdatei geladen und aufgebaut wird. Mein Ziel war es, eine einfache, aber funktionale Spieleinstanz auf Basis des FPS-Templates der UE5 zu realisieren, die die folgenden Kernpunkte abdeckt:

- **Struktur des Labyrinths:** Ein zweidimensionales Gitter (32×32 Zellen) wird in einer externen Textdatei (`.txt`) als Matrix aus Zeichen kodiert, wobei „0“ für leere Felder und „1“, „2“ bzw. „3“ für drei unterschiedliche Wandmaterialien steht. Die vier Ecken (jeweils 2×2 Zellen) bleiben stets frei, um dort den Spieler platzieren zu können.
- **Dynamische Generierung:** Aufruf und Logik der Methoden `OnConstruction()` und `BeginPlay()` in der Haupt-Actor-Klasse `ALabyrinthGame` sorgen dafür, dass Änderungen an der Textdatei sofort im Editor sichtbar sind und beim Spielstart die Objekte (Wände, Kisten, Waffen, Schatz) erzeugt bzw. bei einem Restart erneuert werden.
- **Erweiterte Gameplay-Elemente**: Neben den Mauern umfasst das Spiel zufällige Kisten, mit denen Spieler rudimentär interagieren und die mit den randomisiert verteilten Bälle-Waffen abgeschossen werden können. Nach dem Fund eines Schatzes wird das Level neu gestartet, wobei Kisten, Waffen und der Schatz an neuen Positionen generiert werden.

Beigefügt habe ich diesem Bericht und dem Projekt drei UML-Diagramme und die Level1.txt, aus der das Labyrinth generiert wird. Außerdem habe ich ein Video aufgenommen, dass den „Spieldurchlauf“ und den Weg dahin kurz präsentiert.

2. Wahl der Engine und ihre Kerneigenschaften

Die Wahl der Unreal Engine 5 begründet sich in ihrer weitflächigen Verbreitung im Kontext aktueller Videospiele, der nativen Unterstützung von Raytracing und der Coding-Infrastruktur.

Die Engine bietet mit Nanite für hochdetaillierte Geometrie und Lumen für Global Illumination und Hardware-Raytracing eine moderne Rendering-Architektur. Im Labyrinth sorgt ein einzelnes *DirectionalLight* („Sonne“) in Kombination mit aktiviertem Raytracing für realistische Schattenwürfe jeder Wand und jedes Objekts. Besonders auffällig sind die Spiegelungen auf den metallenen Typ-2-Mauern, wenn Kisten gespiegelt werden oder die gelben Projektile dagegen prallen und Lichtstrahlen mehrfach reflektiert werden.

Als Basis des Projekts diente mir das *FirstPerson*-Template, das eine Spielfigur, die Kamerasteuerung und eine Basis-Map enthält. So konnte ein Großteil der grundlegenden Setup-Arbeit massiv beschleunigt werden. Mit Rider und dem *UnrealLink*-Plugin ließ sich der C++-Code per Live Coding (*STRG + ALT + F11*) unmittelbar in den laufenden Editor injizieren, wodurch Änderungen an OnConstruction() oder den Spawning-Methoden ohne vollständigen Rebuild getestet werden konnten.

3. Softwarearchitektur und Klassenübersicht

Die Kernlogik des Labyrinth-Prototypen ist in sechs C++-Klassen und ihren zugehörigen Komponenten organisiert. Im Zentrum steht der *Actor* `ALabyrinthGame`, der das Level aus einer externen Textdatei einliest, die Mauern erzeugt und alle interaktiven Objekte (Kisten, Waffen, Schatz) spawnt. Die übrigen Klassen kümmern sich um die Spielfigur, Spielregeln, Waffensysteme, *Pickup*-Trigger und das physikalisch simulierte Projektilverhalten.

1. `ALabyrinthGame` (*Actor*): Liest in `OnConstruction()` das 32×32-ASCII-Maze ein, erzeugt *Instanced Static Meshes* für die Wände und in `BeginPlay()` spawnt es kistenweise Crates, Waffen-Pickups und die finale Statue – jeweils inklusive Aufräum-Logik für Neustarts.

2. `ALabyrinthCharacter` (*Actor*): Stellt die First-Person-Pawn-Logik bereit, bindet Input-Aktionen für Bewegung und Blicksteuerung und versorgt Kamera- und Mesh-Subcomponents mit den Spielerbefehlen.

3. `ALabyrinthGameMode` (*GameMode*): Initialisiert beim Spielstart die grundlegenden Game-Settings (*DefaultPawnClass*, HUD, etc.) und übergibt so den Rahmen für das Labyrinth-Gameplay.

4. `ULabyrinthWeaponComponent` (*ActorComponent*): Kapselt das Abfeuern von Projektil-*Actors* (`ALabyrinthProjectile`) und räumt beim Spielende alle offenen Waffenvorgänge auf.

5. `ULabyrinthPickUpComponent` (*ActorComponent*): Verwendet eine *Sphere-Overlap*-Komponente, um *Pickup*-Events auszulösen, wenn der Spieler Objekte (Waffen, Schatz) berührt, und benachrichtigt per *Delegate*.

6. `ALabyrinthProjectile` (*Actor*): Simuliert das Kugel-Projectile mit Bounces über *UProjectileMovement*, wechselt bei Kollision in die Physik-Simulation und bleibt letztlich statisch liegen.

Den Überblick über die zentralen Klassen habe ich in einem UML-Klassendiagramm visualisiert, siehe den Anhang oder die Bilddateien anbei.

4. Laufzeitablauf

OnConstruction: Beim Platzieren oder Neuladen des *Actors* im Editor ruft die Engine `OnConstruction(`*Transform*`)` auf. `ALabyrinthGame` lädt die ASCII-Maze-Datei über `FFileHelper::LoadFileToStringArray()`, bricht im Fehlerfall ab (Log-Fehler) oder löscht sonst alle bisherigen Instanced-Mesh-Instanzen. Anschließend durchläuft es ein 32×32-Raster und fügt für jeden Zellwert „1“, „2“ oder „3“ je nach Material die Transform-Daten als neue Instanz in die entsprechenden *Instanced-Static-Mesh-Components* ein.

OnBeginPlay: Zum Spielstart führt die Engine `BeginPlay()` aus. Zuerst werden alte Kisten und Waffen-Pickups zerstört (Schleifen über *SpawnedCrates*/*SpawnedRifles*, *SpawnedStatue*). Dann spawnt `ALabyrinthGame` in einem ebenfalls 2-stufigen Ablauf Kisten (mit zufälliger Position und Rotation), Waffefn-*Pickups* (begrenzte Anzahl, verteilter Grid-Index) und schließlich als Schatz eine einzelne Statue an einem freien Rasterzentrum. Alle Spawns erfolgen per `UWorld::SpawnActor()`, Rückgaben der neu erzeugten *Actors* werden gespeichert.

Den Ablauf von *OnConstruction* und *OnBeginPlay* habe ich als Sequenzdiagramm visualisiert, siehe den Anhang oder die Bilddateien anbei.

5. Technische Highlights

- **Physik-Integration**: *Movable-Meshes* mit `SetSimulatePhysics(`*true*`)` ermöglichen interaktive Kisten und Projektile, während statische Objekte (Wände, Boden) von der Physik ausgenommen bleiben. Die Engine-eigene Kollisions- und Physiksimulation sorgt hier für realistisches Verhalten ohne zusätzlichen Code-Aufwand.
- **Minimalistisches ASCII-Format**: Die Wahl einer reinen Textdatei mit den Zeichen „0“, „1“, „2“ und „3“ für das Maze-Layout ist bewusst einfach gehalten. Der Parser in C++ liest zeilenweise 32 Zeichen, ordnet 1–3 den drei Materialtypen zu und überspringt feste 2×2-Ecken. Dies minimiert externe Abhängigkeiten und bleibt leicht erweiterbar.
- **Live-Coding und IDE-Workflow**: Mit JetBrains Rider und dem *UnrealLink*-Plugin konnten Änderungen in C++ ohne Vollkompilierung sofort im laufenden Editor getestet werden. Dieser schnelle Feedback-Loop beschleunigte die Entwicklung und erlaubte rasches Feintuning von Parametern (*CellSize*, Spawn-Wahrscheinlichkeiten etc.).

6. Fazit

Der Einstieg in Unreal Engine 5 über das First-Person-Template war ideal, um schnell eine funktionsfähige Spielinstanz zu haben und mich unmittelbar auf die Kernaufgabe zu konzentrieren, statt Character-Movement oder Input-Mapping von Grund auf zu implementieren. Insbesondere das Live-Coding aus Rider heraus beschleunigte jede Änderung an `OnConstruction()` und den Spawn-Methoden, sodass die Wandplatzierung und zusätzliche Features wie Kisten, Waffen und die Gold-Statue einfach getestet und feinjustiert werden konnten. Mein erster Lauf durch das Labyrinth hat Spaß gemacht und gezeigt, wie schnell der Weg zu einer grundlegenden Implementierung beschritten werden kann.

Gleichzeitig habe ich gemerkt, dass die Engine mit ihren zahllosen Untermenüs, verschiedenen Konfigurationspfaden und der 3D-Transformation eine steile Lernkurve hat. Kleine Anpassungen – etwa das Neupositionieren eines *Actors* oder das Umstellen eines Materials bzw. dessen Textur und deren Skalierung – führten mitunter zu langwierigen Klick-Marathons, und gelegentliche Editor-Crashes (vor allem nach *STRG+Z*) verlangten Gelassenheit. Insgesamt habe ich aber eine Menge gelernt: den Umgang mit Riders Live-Coding-Workflow, die Struktur von Actors und Components sowie die Asset-Pipelines im Editor inklusive Material- und Mesh-Graphen.

Hätte es zeitlich Sinn ergeben oder würde ich nach der Abgabe zur Implementierung zurückkehren, ließe sich das Projekt problemlos ausbauen. Für die nächste Entwicklungsstufe stelle ich mir beispielsweise vor, nach jedem Neustart nicht nur Kisten und Waffen, sondern ein komplett neues Labyrinth per Zufallsgenerator zu erschaffen, an den Wänden prozedural platzierte Lichtquellen anzubringen oder über einen Schalter die Schwerkraft der Kisten ein- und ausschalten zu können. Langfristig wären verschiedene Arten von Objekten, Waffen und Gegenspieler denkbar.

Insgesamt haben mich die Einarbeitung in die Engine, die Implementierung der geforderten Aufgabenstellung, die zusätzlichen Features, die Diagramme und der Abschlussbericht rund 24 bis 28 Stunden beschäftigt.

Anhang: UML-Diagramme

Alle drei Diagramme liegen dieser Abgabe als PNG-Bilddatei mit höherer Auflösung bei.

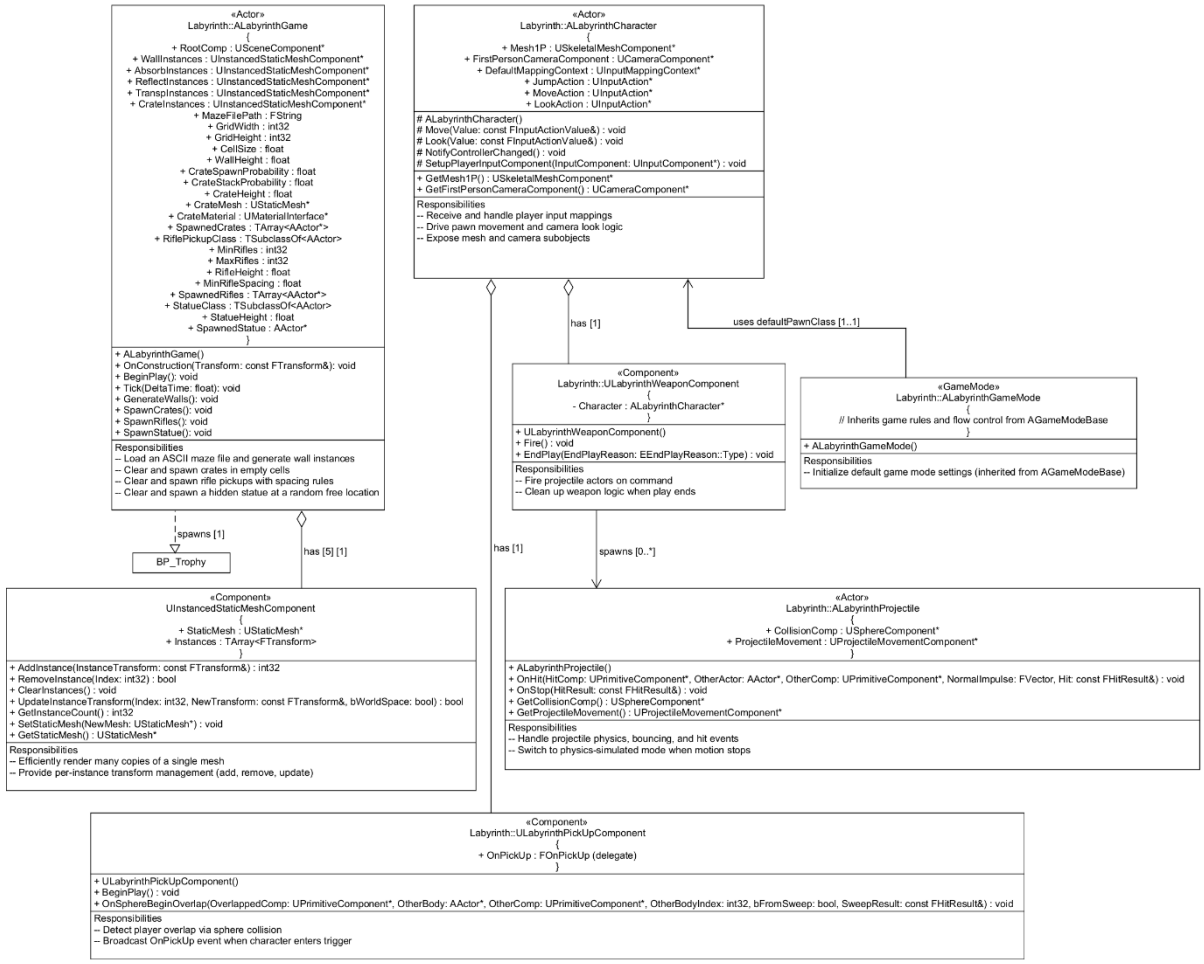

Abbildung 1: Klassendiagramm

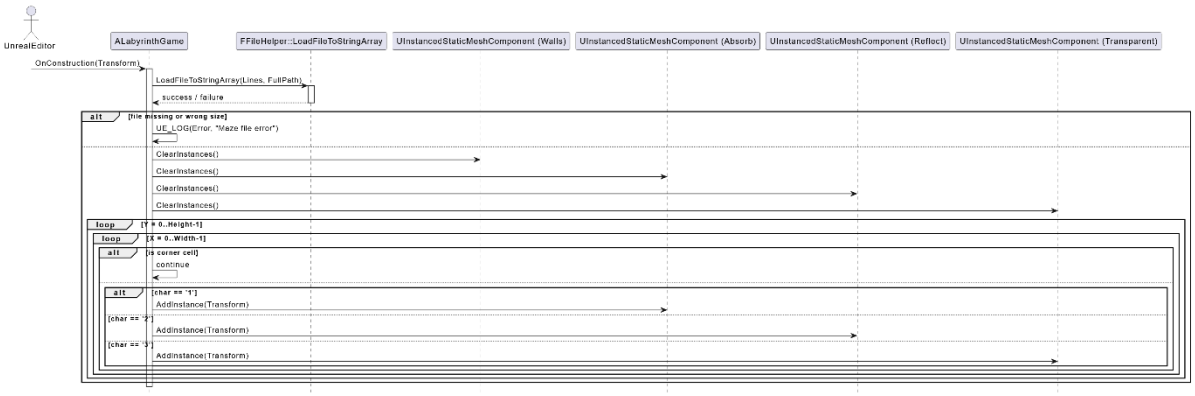

Abbildung 3: Sequenzdiagramm zum Level-Konstruktor

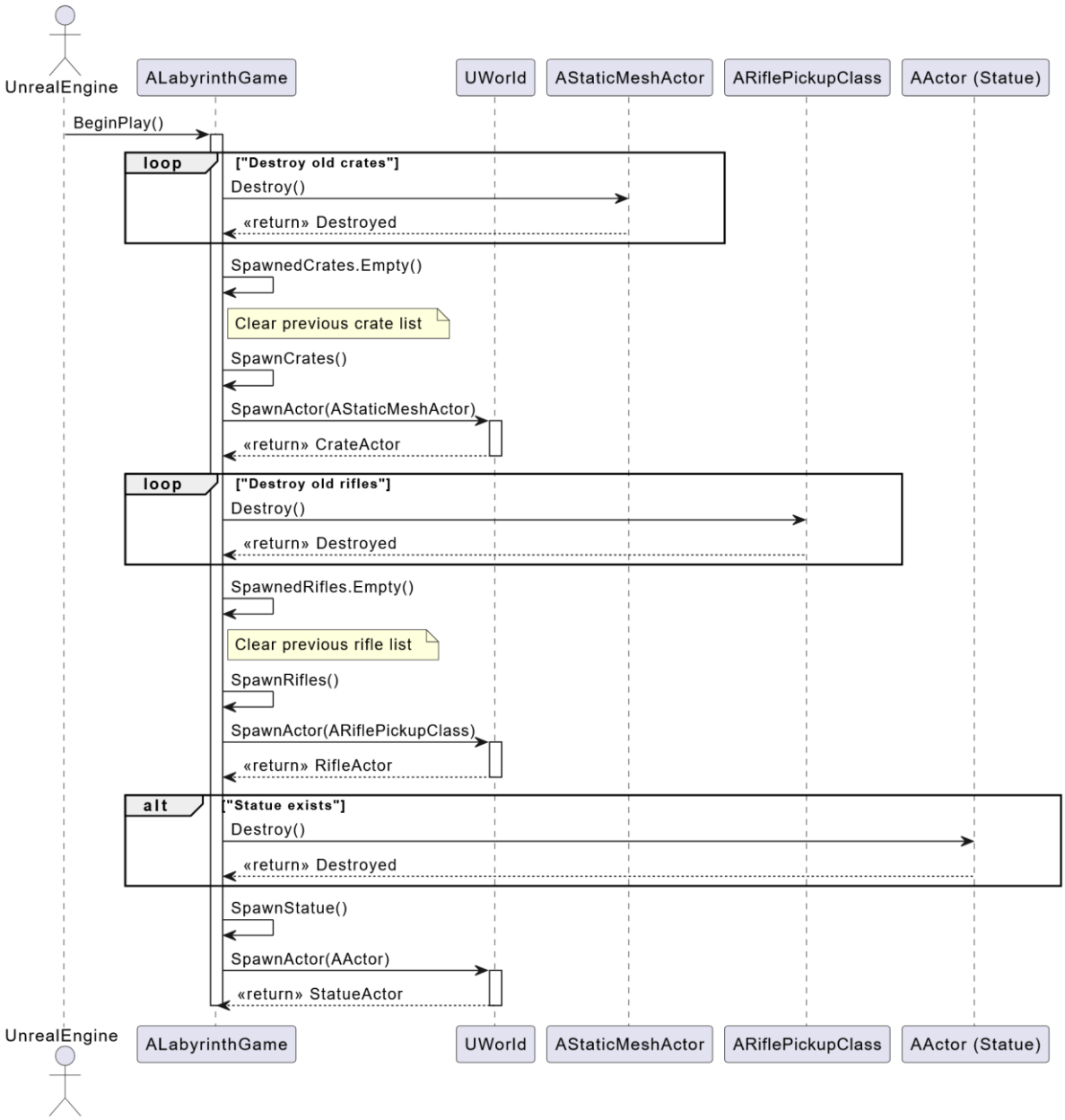

Abbildung 2: Sequenzdiagramm zum Spielstart