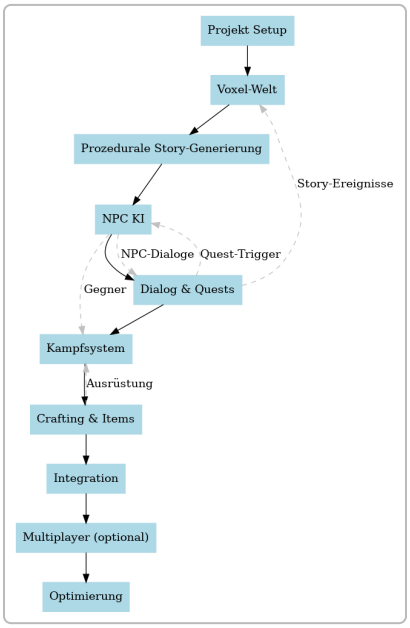
ChatGPT

Entwicklungsplan: Voxel-RPG im *Cube World*-Stil mit Godot 4.4

Überblick und Ziele

Ein voxelbasiertes 3D-Rollenspiel im Stil von *Cube World* soll in Godot Engine 4.4 mit GDScript entwickelt werden. Dieses Dokument beschreibt einen **detaillierten Entwicklungsplan** in Form einer Roadmap mit klar definierten Phasen. Wichtige Kernfeatures des Spiels sind: eine vollständig voxelbasierte Welt (mit kleineren Blöcken als Minecraft), KI-gestütztes NPC-Verhalten (Patrouillen, Reaktionen, Tagesabläufe), storybasierte Welt- und Dungeon-Generierung abhängig von Quests/Events, ein verzweigtes Dialogsystem, rundenbasiertes Kampfsystem, Crafting (Rezepte, Ressourcen, Items) und als optionales Ziel später Multiplayer (Koop) sowie evtl. ein Mobile-Port.

Die Umsetzung erfordert **technologische Grundlagen** (Godot-Engine-Features, Scripting und evtl. Addons) sowie eine sinnvolle **Modularisierung** des Projekts. Nachfolgend wird eine Roadmap vorgestellt, welche die Entwicklungsphasen, verwendete Technologien und Abhängigkeiten der Systeme beschreibt. Jede Phase enthält Hinweise zu Tools/Workflows, Integration der Systeme, Performance/Skalierbarkeit und ggf. Code-Beispiele oder Ressourcenverweise (mit Fokus auf GDScript und Godot 4.4).



Überblick über die geplanten Entwicklungsphasen und deren Zusammenhänge. Die Roadmap verläuft in Phasen von **Projektsetup** über Kernsysteme (Voxel-Welt, prozedurale Generation, NPC-KI, Dialoge/Quests, Kampf, Crafting) bis zu **Integration**, optionalem **Multiplayer** und abschließender **Optimierung**. Gestrichelte Pfeile markieren wichtige Wechselwirkungen zwischen den Modulen – z. B. beeinflussen *Story/Quests* die Weltgenerierung und das NPC-Verhalten (“Story-Ereignisse” und “Quest-Trigger” im Diagramm), und NPCs treten als Gegner im Kampfsystem auf. Diese Übersicht dient als Leitfaden für die detaillierte Planung jeder Phase.

Technologische Grundlagen und Tools

Bevor mit den Entwicklungsphasen begonnen wird, müssen die technologischen Weichen gestellt werden. Godot 4.4 bietet viele relevante Features: eine **leistungsfähige 3D-Engine** mit Vulkan-Renderer (Forward+), GDScript 2.0 (statisch typisierbar, besserer Editor-Unterstützung), eingebaute **Multithreading-Unterstützung** und GDExtension-Schnittstelle für native Module. Für unser Voxel-RPG werden insbesondere folgende Technologien und Addons eine Rolle spielen:

System/Feature	Godot 4.4 Technologien & Addons	Anmerkungen
Voxel-Welt	*Zylann's Voxel Module* (C++ via GDExtension) [1] *Voxel-Core* (GDScript-Addon) [2]	Vorgefertigte Voxel-Engine-Lösungen für Godot. Zylanns Plugin ist performant (C++), unterstützt große, infinite Terrain & LOD; Voxel-Core ist rein GDScript (plattformunabhängig, anpassbar) mit Features wie **Greedy Meshing** zur optimierten Mesh-Generierung [3] . Beide bieten Editor-Tools (Voxel-Modellierung, Import von MagicaVoxel *.vox* Dateien etc.) und vereinfachen die Voxel-Implementierung enorm. Eine eigene Voxel-Engine in GDScript zu schreiben wäre sehr aufwendig [1] .
Prozedurale Generierung	*FastNoiseLite*-Library (integriert in Godot) [4] *OpenSimplexNoise* (Godot-Klasse)	Godot liefert eingebaute Noise-Generatoren zur Erstellung von Höhenkarten, 3D-Rauschen etc. Mit FastNoiseLite können verschiedene Noise-Algorithmen (Perlin, OpenSimplex, Cellular etc.) genutzt werden, um Terrain, Biome und Dungeons prozedural zu erzeugen.
NPC-KI & Wegfindung	*NavigationServer3D*, *NavigationRegion* (eingeb. Navmesh-System) *AStar* Pfadsuche (Godot-Klasse)	Für komplexes Gelände kann ein Navigationsmesh generiert werden, auf dem NPCs mit Godots NavigationServer Pfade finden. Alternativ, besonders bei **Voxel-Gelände**, lässt sich A-*Pfadsuche auf einem Gitter nutzen (Knoten z.B. die Center von begehbaren Voxeln)* [5] *. NPC-Verhalten wird in GDScript umgesetzt (Zustandsautomaten, ggf. Godot-Signalmechanismen); für komplexe Verhaltensbäume gibt es Plugins wie* LimboAI* (C++ Behavior Trees) – optional einsetzbar, da auch mit einfachen FSMs viel erreicht werden kann.

System/Feature	Godot 4.4 Technologien & Addons	Anmerkungen
Dialog-System	*Dialogic 2* (Godot-Addon) *Dialogue Manager* (Addon für Godot 4.4) [6]	Für verzweigte Dialoge existieren fertige Lösungen: **Dialogic** bietet einen visuellen Editor zum Erstellen von Dialogabläufen (Charaktere, Porträts, Wahlmöglichkeiten). **Dialogue Manager** von Nathan Hoad (für Godot 4.4) ermöglicht das Schreiben von Dialogen in skriptähnlicher Syntax und deren einfache Integration [7]. Beide ersparen viel Entwicklungszeit bei der Implementierung eines Dialogsystems. Alternativ kann ein eigenes System auf JSON/Resources-Basis entwickelt werden, was aber mehr Aufwand bedeutet.
Kampfsystem (rundenbasiert)	Godot GDScript (Signale, Coroutinen, Timer) *OpenRPG* (Referenzprojekt, Godot 3.x)	Das Kampfsystem wird hauptsächlich in GDScript als eigenständiges Modul umgesetzt. Godots **Signals** und **Coroutines (await)** sind hilfreich, um Rundenabläufe zu koordinieren (z.B. auf Aktionen warten) [8]. Als Inspiration kann man sich Projekte wie *OpenRPG* [9] anschauen, wobei diese auf Godot 3 basieren.
Crafting & Inventar	Godot **Resource**-System *Inventory-Plugins* (optional)	Crafting erfordert eine solide Inventarverwaltung. Godots Resource-Dateien eignen sich, um **Item- und Rezeptdaten** zu speichern (z.B. als `.tres` mit export-Variablen für Name, Icon, Eigenschaften). Es gibt Community-Plugins für Inventare, aber ein eigenes Inventarsystem in GDScript (Grid-Slots, Stapelbarkeit etc.) ist gut machbar [10] und bietet volle Kontrolle über Crafting-Logik.
Multiplayer (Koop)	Godot **High-Level Multiplayer API** (Scene replication über RPCs) [11] *MultiplayerSpawner/Sync* (Godot 4)	Godot 4 vereinfacht Multiplayer durch eingebaute Replikation von Nodes. Über **Remote Procedure Calls (RPC)** können Aktionen und Zustände im Netzwerk synchronisiert werden. *MultiplayerSynchronizer*-Nodes erlauben automatische Syncs von Node-Properties zwischen Host und Clients. Für einen Koop-Modus kann ein Peer als Host/Server agieren, der z.B. Welt-Events und Queststatus autoritativ verwaltet. Wichtig ist, von Anfang an die Architektur so zu gestalten, dass Zustände synchronisierbar sind (z.B. deterministische Weltgenerierung mit gleichem Seed auf allen Clients, damit nur Änderungen übertragen werden müssen).

Zusammenfassend werden wir Godot 4.4 mit GDScript nutzen und gezielt auf die genannten Engine-Features und Plugins zurückgreifen, um Entwicklungszeit zu sparen und die geforderten Features umzusetzen. Insbesondere das **Voxel-Plugin** (Zylann oder Voxel-Core) ist ein zentrales Werkzeug, da es die Grundlage der Welt darstellt und performant tausende von kleinen Blöcken handhaben kann [1]. Ebenso werden wir vorhandene Systeme für Dialoge und Multiplayer nutzen, um uns auf die Spiellogik konzentrieren zu können.

Im nächsten Schritt folgt die in Phasen gegliederte Roadmap, welche die Umsetzung Schritt für Schritt beschreibt.

Phase 1: Projektsetup und Architekturplanung

Ziel dieser Phase: Einrichtung des Godot-Projekts, Planung der Projektstruktur in Module, Auswahl der notwendigen Addons, sowie Umsetzung erster Prototypen der Kerntechnologien (Voxel-Rendering, grundlegende Steuerung) als Machbarkeitsnachweis.

- **Projektinitialisierung:** Godot 4.4 Projekt anlegen, Versionsverwaltung (git) einrichten. Grundlegende Einstellungen vornehmen (Forward+ Rendering aktivieren für evtl. Compute Shader Nutzung, Physics 3D einschalten, etc.). Ggf. bereits die Zielplattform PC einstellen (wir entwickeln primär für Desktop).
- **Addons installieren:** Das **Voxel-Addon** einbinden (per AssetLib oder manuell). Für den Anfang empfiehlt sich Zylanns Godot-Voxel-Modul aufgrund der Effizienz und Terrain-Funktionen [1]. Ebenso das gewünschte **Dialog-Addon** (z.B. Dialogic oder Dialogue Manager) hinzufügen, damit spätere Dialoge leicht implementiert werden können. Weitere Plugins nach Bedarf (z.B. ein Inventory-Plugin zur Inspiration, Behavior-Tree-Plugin falls gewünscht, etc.).
- **Projektstruktur planen:** Ordner für Scripts, Szenen und Assets anlegen. Eine mögliche Aufteilung:
- `scenes/` – enthält Unterordner für *Welt/Terrain*, *Charaktere/NPCs*, *UI* (z.B. Dialogfenster, Inventarfenster), *Dungeons*, *Effekte* etc.
- `scripts/` – unterteilt nach Systemen: *WorldGeneration.gd*, *NPCBehavior.gd*, *QuestManager.gd*, *CombatSystem.gd*, *CraftingSystem.gd*, *DialogueManager.gd* (falls kein Plugin genutzt wird) usw.
- Autoload-Singletons erwägen für globale Manager: z.B. `TimeManager` (für Tageszeit), `QuestManager` (Quest-Logik), `NetworkManager` (für Multiplayer).
- **Coding Guidelines:** GDScript 2.0 mit statischen Typen nutzen, wo möglich, um Fehler früh zu erkennen. Signal-Verbindungen zentral dokumentieren. Insbesondere für Rundenkampf und Quest<->Welt-Interaktionen ist ein gut durchdachtes Signalsystem sinnvoll (z.B. Signal `quest_completed(quest_id)` im QuestManager, auf das NPCs oder Weltobjekte reagieren können).
- **Prototyping Kernsysteme:** Bereits in dieser Phase kleine Prototypen bauen:
- **Voxel-Welt Prototyp:** Eine Scene mit dem Voxel-Terrain-Node des Plugins erstellen, ein einfaches flaches Terrain generieren (z.B. 16×16×16 Blöcke) und im Viewport darstellen lassen. Dadurch verifizieren wir, dass das Plugin funktioniert und wir die API verstehen. Wenn ohne Plugin: als Machbarkeitsbeweis ein Skript schreiben, das einen Würfel aus vielen kleineren Cube-MeshInstances generiert, um direkt Performance-Limits zu sehen – anschließend diesen Ansatz aber durch Chunk-Meshing ersetzen.
- **Bewegungssteuerung:** Eine Spielersteuerung hinzufügen (z.B. ein KinematicBody/CharacterBody mit WASD-Steuerung und einer frei drehbaren Kamera), um schon jetzt durch die Welt fliegen/laufen zu können. So kann das Terrain live getestet werden.
- **Rendering-Test:** Beleuchtung einrichten (Sky, Sonne) und testen, wie die voxelbasierte Landschaft aussieht. Eventuell schon Godot 4.4 Global Illumination (GIProbe oder SDFGI)

ausprobieren, um eine schöne Voxelwelt-Beleuchtung zu erzielen. Diese visuellen Aspekte sind nicht kritisch für die Funktion, helfen aber bei Motivation und frühem Feedback.

Am Ende von Phase 1 steht ein lauffähiges Grundgerüst: das Projekt ist eingerichtet, die wichtigsten Plugins sind integriert, und ein erster kleiner Voxel-Level ist begehbar. Die Architektur der Code-Module ist geplant (wenn auch noch nicht implementiert) und wir haben Sicherheit, dass die gewählten technologischen Grundlagen funktionieren.

Phase 2: Voxel-Welt und Terrain-Engine

Ziel dieser Phase: Implementierung der **voxelbasierten Welt** – also die Terrain-Engine, die kleine Blöcke effizient verwaltet, anzeigt und prozedural generiert. Dies bildet das Fundament für alle weiteren Gameplay-Elemente.

Wichtig ist hier die **Wahl des Maßstabs**: "kleinere Blöcke als Minecraft" bedeutet z.B. halbe oder viertel Blockgröße. Wir könnten einen Block = 0,25 m definieren, was sehr feine Details erlaubt. Allerdings vervielfacht sich damit die Anzahl der Blöcke für eine gegebene Fläche – Optimierung ist also essenziell. Vorgehensweise:

- **Voxel-Engine Integration:** Das in Phase 1 gewählte Voxel-Plugin voll einsetzen. Bei Zylanns Modul: einen **VoxelTerrain**-Node hinzufügen. Bei Voxel-Core: einen **VoxelObject** nutzen. In beiden Fällen definieren wir ein **VoxelSet** bzw. Voxel-Typen (z.B. Gras, Erde, Stein, Wasser etc. mit entsprechenden Farben/Texturen). Das Plugin kümmert sich im Hintergrund darum, dass nicht jeder Würfel als einzelnes Mesh gezeichnet wird, sondern **optimierte Chunk-Meshes** erstellt werden – z.B. durch *Greedy Meshing*, bei dem zusammenhängende Flächen zu großen Polygonen verschmolzen werden [3] . Dadurch sinkt die Anzahl der Dreiecke und Draw-Calls drastisch, was für Performance nötig ist.
- **Chunking-System:** Die Welt wird in **Chunks** unterteilt (typisch 16^3 oder 32^3 Blöcke pro Chunk). Wir implementieren eine *Chunk-Manager* Klasse, die Nachbar-Chunks dynamisch lädt/generiert, wenn der Spieler sich bewegt (für potentielle große Welt oder unendliche Welt). Unbenötigte Chunks werden wieder entladen. Beide genannten Plugins bringen hier schon Funktionalität mit: Zylanns Modul unterstützt LOD und Streaming für unendliche Welten out-of-the-box. Alternativ kann man chunkweises Generieren via GDScript Threads durchführen: z.B. den *FastNoiseLite* auf einem Hintergrund-Thread nutzen, um Höhenkarten zu berechnen, dann im Hauptthread das Mesh aktualisieren. (Godot 4.4 erlaubt Threads und das Voxel-Modul nutzt diese auch für Generation, sowie Compute-Shaders zur Meshoptimierung, falls möglich [12] .)
- **Terrain-Generierung (Grundlagen):** Implementierung einer **Heightmap-Generierung** für Oberwelt-Terrain: z.B. Simplex Noise für hügelige Landschaften. Ein *WorldGenerator* Script kann pro Chunk Koordinaten einen Höhenwert liefern und den Voxel-Typ bestimmen (alles unterhalb des Höhenwertes = Gestein/Erde, oberste Schicht = Gras, darunter evtl. mehrere Schichten verschiedener Materialien). Bereits hier kann man Biome einführen (andere Noise-Frequenzen für verschiedene Regionen, oder zusätzlicher Temperatur/Feuchtigkeits-Noise zur Bestimmung von Wüste vs. Wald etc.). Die generierten Daten speisen wir ins Voxel-Plugin (bei Zylann z.B. via *VoxelStream* Interface oder direkte Setzen von Voxeldaten im Chunk).
- **Kleinere Blöcke umsetzen:** Da die Blocks kleiner als üblich sind, sollte die **Größenskalierung** konsistent sein. Wir passen ggf. Godots Projekt-Einheiten an (Standard 1 Einheit = 1 m). Wenn ein Block 0,25 m ist, dann bewegt sich der Spieler mit scheinbar 4x Geschwindigkeit relativ zu den Blöcken, wenn wir nichts ändern – hier evtl. anpassen (oder einfach akzeptieren, dass der Maßstab anders ist). Wichtig ist, dass die Kollisionskapsel des Spielers mehrere Blöcke hoch ist (z.B. 8 Blöcke hoch bei 0,25 m Blockgröße für ~2 m Charakter).

- **Physikalische Kollision:** Das Terrain soll begehbar sein. Das Voxel-Plugin kann automatisch **StaticBody-Kollisionen** für generierte Meshes erzeugen (Voxel-Core z.B. erwähnt "embed self-maintaining StaticBodies" für VoxelObjects [13] [14]). Wir stellen sicher, dass der Spieler nicht durch den Boden fällt und z.B. gegen Voxel-Wände laufen kann. Test: Kollision aktivieren und mit dem Charakter im generierten Terrain umherlaufen.
- **Grundlegende Interaktion:** In dieser Phase können wir bereits simple Interaktionen vorsehen, z.B. das Zerstören und Platzieren von Blöcken (wie in Minecraft). Das ist fürs RPG nicht zwingend Gameplay-relevant, aber gut zum Testen der Voxel-Engine (z.B. einen "Edit-Modus" per Tastendruck, in dem der von der Kamera anvisierte Block entfernt oder hinzugefügt wird). So sehen wir, ob das Update der Chunk-Meshes in Echtzeit performant ist.
- **Speicher und Sichtweite:** Da wir potenziell sehr viele kleine Blöcke haben, begrenzen wir die initiale **Sichtweite** (Renderdistanz). Z.B. 10×10 Chunks um den Spieler herum aktiv halten. Wir müssen ein gutes Gleichgewicht finden zwischen Sichtweite und Performance, insbesondere falls später Mobile berücksichtigt werden soll. Gegebenenfalls implementieren wir eine **LOD** (Level of Detail) – in größerer Entfernung werden z.B. kleinere Voxeldetails aus Performancegründen zusammengefasst. Zylanns Modul bietet Voxel-LOD (auf Basis einer Octree-Hierarchie), was wir konfigurieren können. Für den Start reicht aber oft, die Welt nicht unendlich groß zu machen, sondern ein begrenztes Gebiet zu definieren, um das Gameplay aufzubauen.

Nach Phase 2 haben wir eine funktionierende prozedurale Voxelwelt: Der Spieler kann eine Landschaft aus kleinen Blöcken erkunden, die Engine erzeugt/entlädt je nach Position Chunks, und die Performance ist durch Chunking/Greedy Meshing ausreichend. Diese Welt ist zunächst noch generisch (zufällig), aber in Phase 3 werden wir die Generierung mit Story-Elementen anreichern.

Phase 3: Prozedurale & Story-basierte Weltgenerierung

Ziel dieser Phase: Die zufällige Weltgenerierung wird mit **Story-Logik** verbunden, sodass Orte, NPCs und Dungeons in Einklang mit geplanten Quests oder Ereignissen entstehen. Hier verschmelzen **Procedural Content Generation (PCG)** mit **vordefinierten Storystrukturen** zu einer "lebendigen" Welt.

Vorgehensweise in zwei Schritten: Zunächst die generische prozedurale Generierung verfeinern (Biome, Strukturen), dann storybasierte Inhalte einweben.

- **Biom- und Struktur-Generierung:** Aufbauend auf der Heightmap aus Phase 2 fügen wir **Biomes** und **Static Strukturen** hinzu. Beispielsweise:
- **Biom-Verteilung:** Zusätzliche Noise-Maps bestimmen Vegetation und Klima. Bereiche mit hoher Feuchtigkeit -> Wälder (Bäume als vordefinierte Voxel-Strukturen pflanzen), trocken + warm -> Wüste (Sand statt Gras, Kakteen als Modelle).
- **Strukturen:** Kleine Siedlungen, Ruinen oder Türme können prozedural verteilt werden. Dies kann geschehen, indem an bestimmten Noise-Merkmalen (z.B. Flachland nah Wasser) ein vorgefertigtes Voxel-Struktur-Template instanziert wird. Hier zahlt sich die Möglichkeit aus, externe Voxelmodelle zu importieren (z.B. mit MagicaVoxel erstellte *.vox-Dateien von Häusern, die via Plugin als VoxelObject platziert werden [15]). Wir entwickeln eine Reihe solcher vorgefertigter Strukturen (Hütten, Bäume, Dungeon-Eingänge, etc.), um sie später durch die Story gezielt zu nutzen.
- **Dungeons**: Parallel zur Oberweltgenerierung überlegen wir eine Methode, Dungeons (Untergrund oder separate Instanzen) prozedural zu erzeugen. Evtl. ein spezieller Dungeon-Generator, der z.B. nach Roguelike-Art Räume und Korridore setzt. Diese Dungeons können

entweder direkt in der Voxelwelt unter der Erde angelegt werden (z.B. ein Höhlensystem), oder als separate Scenes, die beim Betreten geladen werden.

- **Story-Parameter definieren:** Nun der **storybasierte** Teil: Zunächst klären wir die Story/Quest-Struktur des Spiels im Groben. Beispielsweise gibt es einen Hauptquest-Strang mit bestimmten Schlüssellokationen (etwa ein Dorf, aus dem der Spieler stammt; einen bösen Kult in einem Tempel; einen Endgegner in einer Burg). Zusätzlich Nebenquests, die bestimmte NPCs und Orte betreffen. Diese Informationen formulieren wir als Parameter für die Weltgenerierung:
- *Story-Orte:* Liste von benötigten Orten: z.B. Heimatdorf (Startpunkt), Hauptstadt, Dungeon X für Quest Y, etc. Jedem Ort ordnen wir Eigenschaften zu (Position in der Welt, bevorzugtes Biom/Terrain, welche NPCs dort sein sollen, welches Quest damit verknüpft ist).
- *NPC-Platzierung:* Liste wichtiger NPCs mit Rollen (Questgeber, Begleiter, Händler, etc.), die an bestimmten Orten leben sollen.
- *Story-Events:* Bestimmte Ereignisse könnten Weltveränderungen auslösen (z.B. ein Vulkan bricht im Finale aus). Diese berücksichtigen wir, indem wir das Potential für solche Events in der Welt vorsehen (der Vulkan existiert in der generierten Welt, wird aber erst durch Event aktiv).
- **Gezielte Welt-Manipulation nach Story:** Nun passen wir den existierenden Generator an, damit er **Story-Vorgaben** berücksichtigt:
- Anstatt alle Strukturen rein zufällig zu setzen, erzwingen wir z.B., dass im Startgebiet definitiv ein Dorf generiert wird. Wir können hier mit *vorher festgelegten Seeds* arbeiten: z.B. nutzen wir einen konstanten Seed für Story-bezogene Objekte, damit sie reproduzierbar erscheinen, während der Rest der Welt einen anderen Seed hat. Oder wir "reservieren" bestimmte Weltkoordinaten für Story-Content.
- Beispiel: Quest „Finde das Amulett im Waldtempel". Dazu: Während der Generierung prüfen wir, ob Quest „Amulett" aktiv oder Teil der Story ist. Ist dies der Fall, erzwingt unser Generator, dass es in einem Wald-Biom einen **Tempel** gibt. Konkret: im Noise nach einem geeigneten Waldgebiet suchen, dort unsere *Tempel-Struktur* platzieren und einen speziellen NPC/Item darin spawnen. Die Quest-Daten halten Verweise: ("TempelKoordinate = (x,y)").
- Wir entwickeln dazu eventuell ein **Regelsystem** oder Skripte je Story-Ereignis: z.B. eine JSON/Config, die besagt *„Wenn Quest X in Welt, dann platziere Struktur Y in Biom Z und setze NPC A dorthin"*. Solche Regeln können wir hart codieren oder datengetrieben laden.
- In der Forschung zum prozeduralen Quest-Generieren wird ähnlich vorgegangen – das *CONAN*-System etwa baut auf einem vordefinierten Weltzustand (Orte, NPCs, Items) auf und generiert darauf Quests [16]. Wir gehen den umgekehrten Weg: definierte Quests beeinflussen den Weltzustand. Wichtig ist, Kohärenz zu erreichen, sodass Quests nicht wie Fremdkörper wirken, sondern in der Welt logisch eingebettet sind.
- **Verbindung von Welt und Quest-System:** Bereits jetzt sollten wir die Brücke schlagen zum Quest-Manager. Wir implementieren einen rudimentären **QuestManager** (oder nutzen ein JSON-Format für Quests), der Quests mit Orten/NPC verknüpft. Die Weltgenerierung registriert beim QuestManager die Positionen/Bedingungen, z.B. *QuestManager.register_location("Waldtempel", chunk_x, chunk_y, temple_id)*. Der QuestManager kann später genutzt werden, um z.B. Questmarker auf der Karte anzuzeigen oder die richtige Dialogzeile für NPCs zu liefern, die einen Quest vergeben.
- **Testlauf Story-Gen:** Ein Durchlauf der Weltgenerierung mit einem einfachen Story-Szenario durchführen. Beispielsweise eine Mini-Story: *Starte im Dorf – Dungeon generiert sich in der Nähe mit einem Boss.* Wir schauen, ob das Dorf korrekt generiert wurde (z.B. 5-6 Häuser, NPCs darin), ob der Dungeon an sinnvoller Stelle ist und erreichbar. Falls Dinge unpassend erscheinen (Dungeon unter Wasser generiert oder NPC im Boden), passen wir die Regeln an. Möglicherweise iterieren wir hier öfter, bis Welt und Story harmonieren.

Nach Phase 3 haben wir eine prozedural generierte Welt, die **nicht bei jedem Start völlig anders** ist, sondern durch Story-Vorgaben geprägt wird. Es existieren definierte Orte und NPCs, die für Quests relevant sind, eingebettet in ansonsten vielfältige prozedurale Landschaft. Dieser Ansatz kombiniert das

Beste aus beiden Welten: Variation und Entdeckungsreiz durch Procedural Generation, aber doch eine sinnvolle, von der Story geleitete Struktur.

(Diese Phase ist komplex und wird ggf. parallel mit späteren Phasen verfeinert werden, da Story und Gameplay-Mechaniken iterativ angepasst werden. Eine enge Verzahnung mit Phase 8 – Quest-System – ist hier gegeben, auch wenn wir erst später die Quests voll implementieren.)

Phase 4: NPCs und KI-Verhalten

Ziel dieser Phase: Entwicklung des **NPC-Systems** – inklusive NPC-Charaktere (Freundliche, neutrale, Feinde), deren KI-Verhalten im Alltag (Patrouillen, Tagesablauf) und in Reaktion auf den Spieler (Gespräche, Kampf). Die NPCs sollen die Welt beleben und als Questgeber, Händler oder Gegner fungieren.

Aufgaben in dieser Phase:

- **NPC-Grundgerüst:** Wir definieren eine **Base-NPC-Szene** (z.B. als CharacterBody3D oder als KinematicBody, je nach Bewegungsbedarf). Diese enthält grundlegende Komponenten: visuelle Darstellung (z.B. ein Placeholder-Mesh oder später ein Voxel-Modell eines Charakters), eine Kollisionsshape, und ein Skript `NPC.gd` mit den grundlegenden Eigenschaften (Name, aktuelle Aktion/State, Referenz auf evtl. Dialog oder Quest). Alle spezifischen NPCs (Dorfbewohner, Monster etc.) werden Instanzen oder abgeleitete Szenen dieser Basis sein.
- **Wegfindung einrichten:** Damit NPCs patrouillieren oder einem Tagesablauf folgen können, benötigen sie **Wegfindung**. In einer voxeligen Welt können wir zwei Ansätze kombinieren:
- Für *Städte/Dörfer* oder generell ebene Areale: pre-berechnete Pfade oder Wegpunkte (z.B. manuell definierte Routen, an denen Dorfbewohner entlanglaufen).
- Für *Wildnis*: Nutzung der Godot-**Navigation**. Wir können in begehbaren Bereichen (z.B. auf dem Terrain ohne zu steile Hänge) ein NavigationMesh generieren. Allerdings ist ein Navmesh auf einem voxel-dynamischen Terrain tricky (es müsste nach Änderungen neu generiert werden). Alternativ nutzen wir ein **A*_ Gitter:** z.B. legen wir ein 2D-Gitter über die begehbaren Oberflächen und markieren begehbare Knoten. Godots `AStar3D` oder `AStarGrid2D` kann Pfade darauf finden. Ein Nutzerbericht zeigt: Man kann Start- und Endpunkte vorgeben und dazwischen A *nutzen, was sehr skalierbar für viele NPCs ist* [5] *. Wir könnten z.B. jedem wichtigen NPC fixe Punkte zuweisen (Haus -> Markt -> Haus) und A* berechnet die Route dazwischen über das Voxelgelände.
- Zusätzlich: Für Gegner-AI im Kampf (falls auf offenem Gelände) oder verfolgende Monster ist Pfadsuche ebenfalls nötig (wenn auch in Phase 6 relevant).
- **KI-Zustandsmaschine:** Implementierung einer **State Machine** je NPC. Zustände könnten sein: `Idle` (steht herum), `Patrol` (läuft Wegpunkte ab), `Chase` (verfolgt Spieler, falls Alarm), `Sleeping` (nachts z.B.), `Dead` etc. In GDScript kann man das simpel via `match state: case STATE_IDLE: ...` machen. Oder man nutzt ein Node-basiertes Verhalten (jede NPC-Szene enthält Nodes für die einzelnen Aktionen und aktiviert diese entsprechend). Godot 4 unterstützt auch Coroutine und Timers – hilfreich, um z.B. eine Wartezeit in Idle zu realisieren (NPC steht 5 Sekunden, dann wechselt zu Patrouille).
- Wir definieren pro NPC-Typ das Verhalten: Dorfbewohner haben evtl. einen **Tagesablauf** (morgens Feldarbeit -> mittags Markt -> abends Taverne -> nachts Schlafen). Das können wir skripten, indem wir an `TimeManager.time_of_day` koppeln: im NPC Script etwa `if time_of_day == 8:00: goto_state(FARM)`.
- Gegner haben einfachere FSM: Wandern in Radius -> wenn Spieler in Sicht, Attacke (hier Übergang zum Kampfmodus Phase 6).

- Für komplexere KI kann man ein **Behavior Tree** Plugin wie *LimboAI* nutzen, das visuell bearbeitet werden kann. Aber oft reicht FSM + ein paar **zufällige Variationen** (z.B. 10% Chance, dass NPC einen alternativen Pfad nimmt, damit nicht alle wie Roboter identisch handeln).
- **Patrouillen & Routen:** Für Wachen oder Stadtbewohner definieren wir Patrouillenpunkte. Das kann in der Szene als Pfad eingebaut sein (Path3D + Curve) oder als Liste von Koordinaten. NPC folgt diesem Pfad in einer Schleife. Godots Navigation kann uns an die Zielpunkte führen, oder wir skripten einfache Geradeaus-Bewegungen zwischen Punkten.
- **Reaktionen auf Spieler:** Die NPCs sollen auf den Spieler reagieren:
- *Sichtfeld:* Einfacher Ansatz: ein Area3D (Kegel oder Kugel) vor dem NPC als Sichtfeld, das den Spieler detektiert. Wenn der Spieler in Reichweite kommt, löst das Area ein Signal aus (`body_entered` -> Spieler gesehen). Je nach NPC-Typ: Feind wechselt zu `Chase` oder `Attack` State; freundlicher NPC könnte den Kopf zum Spieler drehen oder einen Begrüßungssatz sagen (später Dialog öffnen).
- *Alarmketten:* Wenn ein NPC etwas bemerkt (z.B. ein Schrei bei Kampf), könnten nahe NPCs es hören. Hier ggf. ein einfaches System: NPC ruft `emit_signal("alarm", self.position)` und alle NPCs in Umkreis reagieren (durch globales Management oder Areas).
- **Tagesablauf & Scheduling:** Um *Tageszyklen* umzusetzen, nutzen wir einen globalen *TimeManager* (z.B. als Autoload). Dieser tickt jede Sekunde Spielzeit und skaliert auf einen Tag/Nacht-Zyklus (z.B. 1 Spielstunde = 1 Minute Echtzeit). NPCs registrieren sich ggf. beim TimeManager oder pollen `time_of_day` in `_process`. Zur Effizienz: Nicht jeder NPC sollte jede Frame die Zeit prüfen. Besser mit Timern oder signalisieren: TimeManager kann zu bestimmten Stunden ein Signal senden, z.B. `time_reached(hour)`. NPC hört darauf und entscheidet: *„Oh, 18:00 – jetzt nach Hause gehen“*. Spiele wie Stardew Valley nutzen oft vordefinierte Zeitpläne pro NPC. Wir können analog eine Liste von Aktionen mit Uhrzeiten pro NPC hinterlegen.
- **KI-gestützte Entscheider (optional):** “KI-gestützt” könnte auch bedeuten, fortgeschrittene Techniken einzusetzen, z.B. ein **Goal-Oriented Action Planning (GOAP)** oder Utility-System, wo NPCs basierend auf Bedürfnissen Aktionen wählen (Hunger -> isst, Langeweile -> geht spazieren). Das wäre sehr komplex; vermutlich reicht ein deterministischer Plan (Schedule) plus etwas Randomness. Falls Zeit ist, könnte man z.B. Dorfbewohner kleine Variationen geben: Wenn kein Spieler in Nähe ist, wählen sie aus ein paar zufälligen Idle-Aktivitäten (am Brunnen stehen, auf einer Bank sitzen etc.), um nicht starr zu wirken.
- **Implementierung & Tests:** Wir implementieren das NPC-Verhalten schrittweise. Starten mit einem NPC (z.B. einer Wache, die patrouilliert). Testen im Spiel: Wache läuft ihren Weg ab? Reagiert auf den Spieler? Dann ausbauen: mehrere NPCs in ein Dorf setzen, ihren Zeitplan differenzieren (eine läuft umher, eine steht am Marktstand, abends gehen beide ins Haus).
- Wichtig: Performance beobachten. Zu viele aktive NPCs können kostspielig sein. Ein Tipp aus GTA und anderen Open-Worlds: *Despawn* von NPCs außerhalb der Sichtweite. Wir können implementieren, dass nur im Umkreis von X Chunks um den Spieler NPCs wirklich existieren. Entfernte Dörfer “schlafen” oder spawnen NPCs erst, wenn man näher kommt [5] (ähnlich “GTA pedestrian streaming”). Der QuestManager kann dafür sorgen, dass wichtige NPCs persistieren, unwichtige aber nicht.
- Auch Pathfinding kann teuer sein mit vielen NPCs – daher Pfade möglichst vorcomputen oder selten updaten. Patrouillenrouten stehen fest, müssen nicht dauernd neu berechnet werden. A*-Routen zu dynamischen Zielen (z.B. Spieler) berechnen wir nur bei Ereignis (Spieler entdeckt -> berechne Pfad einmal).

Nach Phase 4 verfügen wir über eine Welt voller NPCs, die ihren Routinen nachgehen und auf den Spieler eingehen. Das Spiel wirkt dadurch belebt. Die NPCs bilden außerdem die Grundlage für Interaktionen in den nächsten Phasen: Sie können Dialoge initiieren (Phase 5) oder Kämpfe auslösen (Phase 6). Wir haben nun gewissermaßen das *“Alltagsleben”* im Spiel implementiert.

Phase 5: Dialogsystem mit Verzweigungen

Ziel dieser Phase: Implementierung eines **Dialogsystems**, das verzweigte Gespräche mit NPCs ermöglicht – inkl. Spieler-Auswahlmöglichkeiten und Konsequenzen (z.B. unterschiedliche Questannahmen). Dieses System ist entscheidend für Storytelling und Quests.

Es gibt zwei Ansätze: **Eigenentwicklung** oder **Nutzung eines Addons**. Angesichts der Komplexität verzweigter Dialoge ist ein Addon sinnvoll, dennoch beschreiben wir beides:

- **Dialog-Addon (Dialogic/Dialoguemanager):** Falls wir *Dialogic 2* nutzen, integrieren wir es jetzt vollständig. In Dialogic können wir mittels einer grafischen Oberfläche Dialoge schreiben: Sequenzen von Texten, Charakterportraits, Optionen für den Spieler mit Verzweigungen. Wir würden pro NPC (oder pro Quest) einen Dialog in Dialogic erstellen. Z.B. einen Dialog für den Bürgermeister im Startdorf, der den Spieler begrüßt und die Hauptquest gibt, mit Optionen "Fragen stellen" oder "Quest annehmen". Dialogic kümmert sich darum, die Auswahl zu verarbeiten, und wir können am Ende eines Dialogs ein Callback (Signal) bekommen, um z.B. eine Quest zu aktivieren.
- Sollte stattdessen *Dialogue Manager* (Nathan Hoad) verwendet werden: Dieser erlaubt das Schreiben von Dialogskripten in einer einfachen eigenen Sprache (ähnlich Ink oder Yarn). Wir könnten so in einem Textfile sagen:

```
NPC: "Willkommen, Reisender."
Player: ["Was ist dieses Dorf?" -> goto label info, "Auf Wiedersehen." -
> end]
```

  und das Addon parst dies und steuert den Dialogfluss. Vorteil: Versionierbarer Text, einfacher Übersetzbar. Beide Addons haben ihre Vorzüge. Wichtig ist: Integration testen – den Beispiel-Dialog ausführen, UI anpassen (Design des Dialogfensters passend zum Spiel).
- **Eigenes Dialogsystem (falls kein Addon):**
- Struktur: Wir erstellen eine Resource-Klasse `DialogNode` mit Feldern: Sprecher, Text, Optionen (Liste von Antworten, die jeweils zu einem anderen DialogNode führen oder ein Ergebnis triggern). Ein kompletter Dialog ist dann ein Netzwerk von DialogNodes. Wir könnten diese in einer JSON-Datei definieren oder als ScriptableObject (Resource) anlegen.
- Laufzeit: Wir bauen eine Szene `DialogBox.tscn` (Control-Nodes) mit Labels für Sprecher und Text, und Buttons für Auswahlmöglichkeiten. Ein Skript `DialogBox.gd` verwaltet den aktuellen Knoten und füllt die UI mit den Optionen. Wenn Spieler eine auswählt, lädt es den nächsten Knoten.
- Verzweigungen und Konsequenzen: Man kann jedem Knoten auch Aktionen mitgeben (z.B. `on_enter: QuestManager.start_quest("Tutorial")` wenn man einen bestimmten Zweig wählt). So lösen Dialogentscheidungen Spielereignisse aus.
- Dieses Rad neu zu erfinden ist aber zeitaufwändig. Daher wirklich nur, falls Addons nicht genügen oder spezielle Anforderungen (wie cinematics) dazukommen.
- **Dialog-Triggers im Spiel:** Wir entscheiden, wie Dialoge gestartet werden. Üblich: Wenn der Spieler nahe an einen NPC kommt und eine Interagieren-Taste drückt. Implementierung: Jeder NPC hat ein Area (Interact-Area). Bei `body_entered(Player)` zeigt ein UI-Hinweis "E drücken für Gespräch". Wenn E gedrückt, ruft Spieler-Script `npc.start_dialog()`. Das NPC-Script oder ein DialogManager ruft dann das Dialog-Addon auf oder zeigt das DialogBox-UI.
- Während des Dialogs sollte der Spieler sich nicht bewegen können und evtl. das Spiel pausieren (zumindest NPCs sollten nicht weglaufen). Wir können z.B. einen globalen `State` "InDialogue" setzen, den Bewegungsscript und KI-Scripts abfragen (und dann einfrieren).

- **Verzweigungen und Einfluss:** Schreiben der Dialoginhalte: Hier müssen wir sämtliche Quests und Story-Infos einfließen lassen. Die Dialoge verzweigen je nach Spielerwahl. Wir planen für wichtige NPCs die Dialogbäume. Tipp: Klein anfangen, z.B. nur 2-3 verzweigende Optionen, sonst wird es unübersichtlich. Später erweitern.
- Konsequenzen: Dialogwahl könnte unterschiedliche Quest-Pfade öffnen. Z.B. man kann einen NPC einschüchtern oder freundlich überzeugen – beide Wege führen zum Ziel aber anders. Solche Entscheidungen speichern wir im QuestManager (bool oder Wert, der den Questverlauf beeinflusst). Diese Werte können dann wiederum von späteren Dialogen abgefragt werden (Addon-Unterstützung vorausgesetzt, oder im eigenen System implementieren wir Variablen).
- **Dialog UI/UX Feinschliff:** Sicherstellen, dass das Dialogfenster optisch und bedienungstechnisch gut ist (Controller vs Maus-Steuerung, falls relevant, Schriftgröße, Porträt-Anzeige).
- Lokalisierung: Falls Mehrsprachigkeit ein Thema ist, jetzt berücksichtigen (Addons unterstützen oft Übersetzungen).
- Cinematic: Man könnte bei Dialogstart die Kamera auf die NPCs zoomen oder eine kleine Zwischensequenz ablaufen lassen. Das wäre Politur, kann aber auch in Phase 10 (Optimierung/ Polish) erfolgen.

Nach Phase 5 ist das Spiel um die Fähigkeit bereichert, mit NPCs zu sprechen. Questgeber können dem Spieler Aufträge erteilen, der Spieler kann Informationen einholen, und durch Antwortmöglichkeiten hat er Einfluss. Dieses Dialogsystem ist eng mit dem Questsystem verwoben: In Phase 8 (Quest & Story-Integration) werden wir sicherstellen, dass Dialoge und Quests synchron laufen (z.B. dass ein Quest erst als abgeschlossen gilt, wenn der entsprechende Dialog stattgefunden hat, und umgekehrt Dialogoptionen nur erscheinen, wenn man den Quest hat).

Phase 6: Rundenbasiertes Kampfsystem

Ziel dieser Phase: Entwicklung des **rundenbasierten Kampfsystems**, in dem der Spieler und Gegner abwechselnd oder nach einem Initiative-System Aktionen ausführen können. Dieses System soll nahtlos ins Spiel integriert werden – entweder als separater Kampfbildschirm oder direkt in der Welt, je nach Design.

Angesichts der voxeligen offenen Welt gibt es zwei Möglichkeiten, rundenbasierten Kampf zu realisieren: 1. **Taktisches Kampfsystem direkt in der Welt** – ähnlich *Fallout 1/2* oder taktischen RPGs: Wenn ein Kampf startet, wechselt das Spiel in einen rundenbasierten Modus, aber die Positionen/ Umgebung bleiben die gleichen. Der Spieler und NPCs stehen auf dem voxelbasierten Terrain und führen rundenweise Aktionen aus. 2. **Separater Kampfmodus** – ähnlich JRPGs: Bei Kontakt mit einem Gegner wechselt die Szene zu einem speziellen Kampfarena (vielleicht abgetrennt), wo rundenbasiert gekämpft wird, und nach dem Kampf kehrt man in die normale Welt zurück.

Hier nehmen wir Option 1 an, da es immersiver ist (und Cube World eher offene Kämpfe hatte, wenn auch in Echtzeit). Das bedeutet, unser Kampfsystem muss *in-place* funktionieren, eventuell mit einem **Rundenmanager**, der die Eingaben steuert.

Vorgehen:

- **Kampftrigger:** Wir legen fest, wann ein Kampf beginnt. Vermutlich, wenn der Spieler einen aggressiven NPC “aggro” triggert (z.B. der Spieler greift an oder wird von einem Monster entdeckt). In dem Moment schalten wir in den Rundenmodus:

- Der *CombatManager* (Singleton oder Scene Node) wird aktiviert. Er sammelt alle *Teilnehmer* im Kampf: Spielercharakter(e) und feindliche NPCs in der Nähe. Für Koop könnte es mehrere Spieler geben, die mitzählen.
- Die Welt (Bewegung, KI außerhalb des Kampfes) pausiert oder läuft stark verlangsamt außerhalb des Kampfbereichs, damit nicht woanders währenddessen Dinge passieren.
- Optional: Ein *Kampflogik-UI* erscheint (z.B. eine Initiative-Leiste oder Menüs für Aktionen).
- **Initiative und Rundenablauf:** Festlegen, wie die Reihenfolge bestimmt wird. Klassisch: Jede Einheit hat einen *Initiative*-Wert (oder Speed). Man könnte rundenweise abwechselnd machen (alle Spieler, dann alle Gegner), aber spannender ist **individuelle Initiative**: Wir berechnen zu Kampfbeginn eine Reihenfolge oder verwenden ein **initiative-turn system** (ähnlich D&D: Wurf + Bonus, oder in JRPGs: der mit höherer Geschwindigkeit kriegt evtl. extra Züge).
- Konkrete Implementierung: Wir halten eine Liste `turn_order` von Character-Instanzen. Diese wird z.B. nach einem Stat sortiert [17] oder einfach abwechselnd Spieler/Feind falls wir simpel halten.
- Der CombatManager hat eine Variable `current_turn_index` und zeigt an, wer dran ist.
- Start der Runde: `current_entity = turn_order[current_turn_index]`. Wenn das der Spieler ist, wird Spielersteuerung freigegeben (oder ein Menü "Wähle Aktion" angezeigt). Ist es ein NPC, führt dessen KI sofort einen Zug aus.
- Nach einer Aktion -> eventuell *Rundenphase* beenden -> nächster Index. Wenn am Listenende -> Rundenindex 0 (neue "Runde").
- Achte darauf, dass Aktionen Zeit dauern könnten (Animationen, Effekte) – wir nutzen Godot **Signals/await** um sequenziell vorzugehen. Z.B. Spieler klickt "Angriff" -> wir spielen Attack-Animation ab und warten auf Signal `animation_finished` bevor der nächste an der Reihe ist. GDScript `await` ist hierfür praktisch (man kann z.B. auf ein Signal der Waffe warten, die bei Treffer auslöst).
- **Aktionsmöglichkeiten definieren:** Typische Aktionen: *Angreifen*, *Fähigkeit einsetzen*, *Gegenstand benutzen*, *Verteidigen*, *Fliehen*. Wir erstellen ein Menü oder Tastenbelegung für den Spieler, um diese zu wählen.
- Für den Anfang: nur *Angriff* (zieht HP vom Gegner ab) und *Gegenstand* (z.B. Trank trinken) implementieren.
- Später können Skills/Magie hinzu (was Einbau eines Skillsystems braucht, eventuell Phase 10).
- Gegner-KI kann simpel sein: Standardangriff auf den Spieler, evtl. Heiltrank wenn low HP, etc. Das kann in einer Funktion `npc.choose_action()` codiert werden (z.B. *if hp<30% then heal else attack*).
- **Werte und Kampfsystem-Mechanik:** Definieren eines **Stats-Systems**: z.B. jedes Character hat HP, Angriffsstärke, Verteidigung, Geschwindigkeit. Diese beeinflussen Schaden und Reihenfolge. Wir legen Formeln fest: Schaden = Angriff - Verteidigung (einfach) oder etwas RPG-typisches. Wir könnten auch rundenbasiert auf *Gitter* kämpfen (dann hätten wir Bewegungsreichweite etc.), aber das verkompliziert es sehr; möglicherweise belassen wir es bei statischen Positionen oder minimaler Bewegung im Kampf.
- Allerdings wenn der Kampf in der Welt stattfindet, Position spielt eine Rolle. Evtl. erlauben wir pro Zug eine Bewegung (z.B. 5 Blöcke laufen) plus Aktion – das wird dann taktisch (ähnlich SRPGs). Ob wir das wollen, hängt vom gewünschten Gameplay ab. Falls ja, müssten wir ein **Grid** definieren (z.B. das voxel grid selbst als Kampfgrid benutzen!). Das wäre elegant: Jede Voxelposition könnte als Feld dienen; aber wegen der kleinen Blöcke wären das sehr viele Felder. Evtl. nutzt man ein gröberes Grid (z.B. 1 Feld = 4x4 Blöcke).
- Diese Entscheidungen beeinflussen das UI (müsste evtl. ein Gitter anzeigen) und AI (Gegner bewegt sich zum Spieler oder in Deckung).
- **Implementierung des Rundenmanagers:** Wir schreiben ein Script `CombatManager.gd`, das ggf. als Autoload existiert, aber wohl besser als Szene, die bei Kampf instanziert wird. Es enthält Logik:

```
var turn_order = []
var turn_index = 0
func start_combat(characters):
    turn_order = characters.sorted(by_speed)
    turn_index = 0
    begin_turn()
func begin_turn():
    var entity = turn_order[turn_index]
    if entity.is_player:
        show_player_options(entity)
    else:
        var action = entity.choose_action()
        perform_action(entity, action)
func end_turn():
    turn_index = (turn_index + 1) % turn_order.size()
    # Falls neue Runde beginnt, evtl. Status-Effekte ticken etc.
    begin_turn()
```

Das obige Pseudocode veranschaulicht den Ablauf in vereinfachter Form. In Wirklichkeit braucht es z.B. `await` oder Signals, um die Sequenz zu steuern (man kann z.B. dem Spieler eine Signalverbindung geben: "Aktion gewählt" -> CombatManager empfängt -> führt aus -> ruft end_turn).

- **Integration in Spielwelt:** Wenn der Kampf vorbei ist (alle Gegner HP <=0 oder Spieler tot), *auflösen*: Bei Sieg z.B. Loot verteilen (Kisten spawnen oder direkt ins Inventar), XP vergeben (falls Levelsystem). Dann CombatManager Scene entfernen, Spielzustand "Exploration" wieder aktivieren. NPCs, die gestorben sind, entweder entfernen (QueueFree) oder als "Leiche" belassen. Hier sollte auch der QuestManager informiert werden, falls der Tod eines NPC questrelevant ist ("Töte Bandit Anführer" Quest erfüllt -> emit quest_complete).
- Wichtig: Während Kampf vermeiden wir, dass andere NPCs plötzlich eingreifen oder die Tageszeit umspringt unpassend. Eine Möglichkeit: *Kampf-Instanzierung*. Wenn man in Kampf geht, könnte man einen **Kampfmodus-Overlay** machen – z.B. Umgebung etwas abdunkeln, andere NPCs in Nähe inaktiv setzen.
- Alternativ, in Open-World könnte es auch passieren, dass ein zweiter Gegner in den Kampf "reinrutscht" (wie bei JRPG, neue Gegner dazu -> turn_order erweitern). Das muss man designen – es kann auch spannend sein, spontan Hilfe zu bekommen oder weitere Monster tauchen auf.
- **UI für Kampf:** Anzeigen von Infos: Lebensbalken der Teilnehmer, wer ist am Zug (hervorheben), eventuell ein Turn-Order-Display (Reihenfolgeportraits). Auch die Kampfaktionen des Spielers werden über UI ausgewählt (Menü oder Shortcut-Tasten). Diese UI bauen wir mit Godot Control-Nodes. Z.B. ein CanvasLayer für KampfHUD.
- **Balancing & Testing:** Zahlreiche Testkämpfe durchführen, um Bugs zu finden (z.B. Initiative-Listensprünge, wie in dem Godot-Forum-Beispiel wo ein Charakter übersprungen wurde [18]). Kleine Debug-Hilfen einbauen, z.B. Taste um Kampf mit Dummy-Gegner zu forcieren.
- Tunen der Stats damit Kämpfe weder trivial noch unmöglich sind. Hier kann ein **Rundenlog** (Textausgabe wer was macht, Schaden) helfen, um Abläufe nachzuvollziehen.

Nach Phase 6 hat unser Spiel endlich **Kampf**. Dies ist ein großer Meilenstein, da nun Kernspielmechaniken – Erkundung (Phase 2/3), Interaktion (Phase 5) und Kampf – abgedeckt sind. Der rundenbasierte Kampf fügt dem voxeligen Explorationsspiel eine strategische Ebene hinzu. Wichtig ist als nächstes die Verknüpfung mit den übrigen Systemen: Gegner sollten Beute droppen, was ins

Inventar/Crafting (Phase 7) fließt, und Kämpfe sind oft Teil von **Quests** (Phase 8), z.B. “Besiege Monster X”.

Phase 7: Crafting-System und Items

Ziel dieser Phase: Implementierung des **Crafting-Systems**, inklusive Ressourcen sammeln, Rezepte definieren und Items herstellen, sowie ein Inventarsystem zur Verwaltung der Items. Crafting erweitert das Spiel um ein Survival-/Fortschritts-Element und greift eng mit Quests (z.B. “sammle 10 Holz und baue ein Boot”) und Kampf (Ausrüstung herstellen) ineinander.

Schritte zur Umsetzung:

- **Inventar-System Grundgerüst:** Zunächst brauchen wir eine Struktur, um Items aufzubewahren. Wir erstellen eine Scene `Inventory` (oder direkt in Player eingebettet) mit z.B. einem Array von Item-Slots. Jedes Slot kann eine bestimmte Anzahl eines Items halten (Stack).
- Datenhaltung: Wir definieren eine GDScript-Klasse oder Resource `Item` mit Eigenschaften: `id`, `name`, `icon`, `max_stack`, ggf. `type` (Rohstoff, Ausrüstung, Quest-Item) und spezifische Werte (für Ausrüstung z.B. Angriffsstärke). Diese Items definieren wir für alle benötigten Objekte: Holz, Stein, Erz, Trank, Schwert etc. Dies kann in einer zentralen Registry oder per Resources-Dateien geschehen (z.B. `items/wood.tres`, `items/iron_sword.tres`).
- Player hat dann ein Array `inventory = []` von Strukturen {Item, count}. Alternativ ein Dictionary mapping Item->count für ungeordnete Lagerung.
- UI: Erstellen eines Inventar-UI (Grid von Slot-Sprites, die mit Item-Icons befüllt werden). Es gibt viele Tutorials, wie man Drag&Drop von Item-Icons implementiert, etc., das können wir rudimentär umsetzen (z.B. Mausklick zum Benutzen, später Drag&Drop falls Zeit).
- **Sammeln von Ressourcen:** In der Welt (Phase 3) sollten nun **Ressourcen** platzierbar sein – z.B. Erzvorkommen, Bäume, Kräuter. Diese repräsentieren wir vielleicht als spezielle Voxel-Blöcke (z.B. Erz-Voxel) oder separate Scene (ein “Baum”-Objekt mit collisionshape). Wenn der Spieler mit entsprechender Aktion (Axt benutzen, Mine-Tool) interagiert, zerstören wir das Objekt und fügen dem Inventar ein Item hinzu (z.B. “Holz”).
- Implementierung: Evtl. vereinfachen – einen generischen “Abbau” per Waffenschlag oder Interaktion. Jedes abbaubare Objekt hat im Skript definiert, was es dropt (z.B. Tree -> 5× Holz). Nach Abbau spawnt Loot oder geht direkt ins Inventar.
- Gegner können auch Loot droppen (schon in Phase 6 berücksichtigen: nach Kampf Items ins Inventar).
- **Rezept-System:** Kern des Craftings sind **Rezepte**. Wir definieren eine Datenstruktur für Rezepte, z.B. ein Dictionary oder Resource `Recipe` mit Feldern: `inputs` (Liste oder Dict von {Item, Menge}) und `output` ({Item, Menge}). Beispiel: Rezept “Eisenschwert” = Eingaben: 2×Eisenbarren + 1×Holz -> Ausgabe: 1×Eisenschwert.
- Wir sammeln alle Rezepte in einer Liste oder einem Dictionary nach Rezeptname. Diese Daten könnten in einer JSON oder CSV extern liegen, oder hardcoded in GDScript. Z.B.:

```
var recipes = {
  "iron_sword": {
     "inputs": {"iron_ingot": 2, "wood": 1},
     "output": {"iron_sword": 1}
  },
  # ... weitere Rezepte
}
```

Eine elegantere Methode: Rezepte als Resources speichern, dann könnte man im Editor bequem neue Rezepte hinzufügen.

- **Crafting-UI und Logik:** Wir brauchen ein Interface, um zu craften. Möglichkeiten:
- *Crafting Table Konzept:* Vielleicht gibt es im Spiel bestimmte Werkbänke/Öfen. Der Spieler öffnet dort ein Crafting-UI.
- *Freies Crafting:* Spieler öffnet jederzeit ein Crafting-Menü (Minecraft-like).
- Zur Vereinfachung hier annehmen: Crafting nur an bestimmten Stationen (Schmiede, Alchemietisch etc.), da es storymäßig Sinn ergibt.
- UI: Eine Liste aller verfügbaren Rezepte (bzw. die, für die der Spieler die Station hat). Wenn der Spieler ein Rezept auswählt, prüfen wir im Inventar die benötigten Zutaten. Falls vorhanden, aktivieren "Herstellen"-Button.
- Bei Klick auf "Herstellen": Im Script iterieren über `inputs`, ziehen vom Inventar ab, fügen `output` Item hinzu. Vielleicht Soundeffekt oder kurze Animation (Schmiedehammer Klang).
- Rezeptverfügbarkeit: Man kann Rezepte von Anfang an alle zeigen, oder manche erst erlernen lassen (Quest-Belohnungen, Bücher). Das könnten wir umsetzen, indem wir z.B. im Player eine Liste `known_recipes` führen. Anfangs nur einfache Rezepte, durch Quests oder NPC-Dialoge werden neue hinzugefügt ("Du hast das Schmieden gelernt: neue Rezepte verfügbar!").
- **Ausrüstungssystem:** Da Crafting oft Ausrüstungsherstellung beinhaltet, überlegen wir ein **Equipment**-System. Z.B. Spieler hat Slots für Waffe, Rüstung, etc., die aus Inventar angelegt werden können. Herstellen eines Schwerts sollte dem Spieler ermöglichen, es auszurüsten für bessere Werte im Kampf.
- Wir können dies vereinfachen: Ausgerüstete Gegenstände befinden sich einfach im Inventar und wir checken beim Kampfscript, ob z.B. ein Schwert im Inventar ist, dann erhöht das den Angriffswert. Besser aber: Player hat Variablen `equipped_weapon`, `equipped_armor` die auf Item referenzieren. Wechsel per UI (drag Waffe auf Waffenslot).
- Crafting liefert dann Items, die in diesen Slots verwendet werden können.
- **Verknüpfung mit Quests:** Schon jetzt kann man Quests einbauen wie "Sammle X und craft Y". Der QuestManager kann das Inventar beobachten (z.B. per Signal `inventory.item_added(item)` oder Polling) um zu erkennen, wenn die benötigten Items da sind. Oder wir implementieren, dass beim Craften gezielt geprüft wird: *if recipe == quest.target_recipe: QuestManager.mark_completed(quest)*.
- **Speicherung:** Falls Save/Load geplant, spätestens jetzt mitdenken – Inventar und gekannte Rezepte/Questfortschritt müsste speicherbar sein (Godot hat dafür das Dictionary->JSON oder ConfigFile Mechanismen).

Nach Phase 7 hat der Spieler ein Inventar, kann Gegenstände aufsammeln und diese zu neuen Items craften. Dieses System fördert die **Erkundung** (Suche nach Ressourcen), gibt dem **Kampf** mehr Tiefe (Beute verwertbar, bessere Ausrüstung craftbar) und fließt in Quests ein. Das Spiel nähert sich damit einem vollwertigen RPG-Erlebnis.

Phase 8: Quest-System und Story-Integration

Ziel dieser Phase: Vollständige **Integration des Quest- und Storysystems** mit der Spielwelt. Nachdem Welt, NPCs, Dialoge, Kampf und Crafting vorhanden sind, muss nun sichergestellt werden, dass **Quests** das Zusammenspiel dieser Systeme steuern und reflektieren. In dieser Phase bauen wir ein Quest-Management-Modul und vernetzen alle relevanten Stellen (Weltgenerierung, NPC-Verhalten, Dialoge, Gegenstände, Kampf) damit, um ein stimmiges Erlebnis zu schaffen.

Aufgaben:

- **Quest-Datenstruktur:** Entwurf einer Struktur, um Quests zu repräsentieren. Eine Quest hat typischerweise:
- `id`, `titel`, `beschreibung` (für das Questlog UI),
- `state` (z.B. NotStarted, InProgress, Completed, Failed),
- eventuelle `requirements` (Vorbedingungen oder zu erfüllende Ziele, z.B. *collect X items*, *kill boss Y*, *go to location Z*),
- `rewards` (XP, Items, Zugang zu neuem Gebiet, etc.),
- `next_quest` oder Auswirkungen auf Folgestory. Wir können eine Quest als Resource oder Dictionary speichern. Z.B.:

```
quest = {
  "id": "main_quest_1",
  "title": "Das Geheimnis des Waltempels",
  "desc": "Untersuche den Tempel im Verfluchten Wald und finde das
Amulett.",
  "state": "NotStarted",
  "goal": {"type": "goto_location", "target": "Waldtempel"},
  "reward": {"xp": 100, "items": {"Amulett": 1}}
}
```

  Komplexere Quests mit Teilschritten könnten `subgoals: [..]` enthalten.
- **QuestManager-Implementierung:** Als Singleton (`QuestManager.gd`) verwaltet eine Liste aller Quests. Hauptfunktionen:
- `start_quest(id)`: Setzt `state = InProgress` und evtl. triggert sofortige Welt/NPC Änderungen.
- `complete_quest(id)`: Markiert als abgeschlossen, gibt Belohnungen, schaltet Folgequests frei.
- `fail_quest(id)`: (optional, falls Scheitern möglich) markiert als gescheitert.
- `is_quest_active(id)`: Helper für NPCs/Dialoge um zu prüfen, ob Quest läuft. Außerdem ein Signal `quest_updated(quest_id, new_state)` zur Benachrichtigung anderer Systeme.
- **Quest-Auslöser im Spiel:** Quests können gestartet werden durch:
- **Dialoge:** (häufigster Fall) – Wenn der Spieler im Dialog zustimmt, einen Quest anzunehmen. Dank Phase 5 können wir z.B. im Dialog-Addon bei einer bestimmten Option ein Callback aufrufen, der `QuestManager.start_quest("quest_id")` ausführt.
- **Bestimmte Orte betreten:** z.B. eine Quest könnte starten, wenn der Spieler zum ersten Mal einen bestimmten Bereich betritt (“Ein dunkler Schatten zieht auf…” -> Quest wird aktiv). Dafür können wir Trigger Areas in der Welt setzen, die beim Betreten `QuestManager.start_quest` aufrufen.
- **Item gefunden:** Auch möglich, dass der Fund eines Items einen Quest triggert (“du hast ein mysteriöses Artefakt gefunden -> neuer Quest es zu identifizieren”). Da das Inventarsystem Signale senden kann, könnte man dort anknüpfen.
- **Quest-Ziele verfolgen:** Hier vernetzen wir die Systeme:
- *Sammel-Quests:* Wenn Ziel ist, X von Item Y zu sammeln, prüfen wir das Inventar. Entweder ständig (jedes Mal, wenn Item hinzukommt, vergleicht QuestManager die Menge) oder explizit beim Versuch der Abgabe. Oft werden Sammelquests in Dialogen abgeschlossen (“Bring dem Schmied 10 Erze”). Dann würde der Dialogcode prüfen: *if player.inventory.has(Erz,10):* -> entsprechende Dialogoption “Hier sind sie” wird verfügbar, ansonsten sagt NPC “Du hast noch nicht alles.”

- *Besiege-Quests:* (Kampfziele) – Wir können Gegner mit einer Quest-ID markieren. Z.B. Boss-Gegner hat `npc.quest_tag = "kill_boss_abc"`. Wenn dieser im Kampf stirbt, ruft sein Script `QuestManager.mark_objective_done("kill_boss_abc")`. Der QuestManager weiß, welcher Quest damit verknüpft ist, und setzt es als erfüllt. Oder einfacher: nach jedem relevanten Kampf prüft CombatManager, ob im Turnier jemand mit Quest-Bedeutung starb, und meldet es weiter.
- *Ort-Quests:* (Erkunden) – entweder via Trigger Area wie oben oder per Distanzcheck. Z.B. Quest "Erreiche die Bergspitze": wir könnten auf dem höchsten Punkt einen invisible NPC/Area platzieren, der bei Kontakt Quest abschließt.
- *Dialog-Quests:* (Infos herausfinden) – kann man lösen, indem man entsprechende Dialogknoten ansteuert. Evtl. QuestManager setzt im Dialog gewisse Pfade frei oder Dialog sendet back an QuestManager.
- **World & NPC Reaktion auf Quests:** Nun das wichtige: *Quests beeinflussen Welt und NPCs*. Durch unseren storybasierten Ansatz aus Phase 3 ist einiges schon initial angelegt (z.B. der Waldtempel existiert nur, weil Quest aktiv). Aber auch dynamisch während des Spiels sollen Änderungen passieren:
- *NPC-Verhalten:* Ein Quest „Beschütze das Dorf" startet – nun werden z.B. alle Wachen-NPCs aggressiver oder ein NPC folgt dem Spieler als Begleiter. Wir können im QuestManager bei `start_quest` solche Änderungen auslösen, z.B. den KI-State einer Wache ändern (`wache.current_state = ALERT`). Oder einen neuen NPC spawnen (der Begleiter).
- *Weltveränderung:* Quests können Voxelwelt verändern. Z.B. *"Zerstöre den Verderbten Baum"* – nach Abschluss könnte der Baum im Wald (ein spezielles Voxel-Structure) verschwinden und durch einen normalen Baum ersetzt werden. Implementierung: Der entsprechende Baum-NPC/Objekt hört auf `quest_updated`, sieht dass sein Quest abgeschlossen, und führt `queue_free()` oder tauscht Mesh. Oder QuestManager manipuliert das Terrain (z.B. platziere einige "Verfallene" Blöcke nachdem Quest startet um Atmosphäre zu ändern).
- *Dungeons/Orte öffnen:* Vielleicht war ein Dungeon versperrt, bis ein Quest startet ("Finde den Schüssel, um die Ruine zu betreten"). Konkret: die Tür zur Ruine ist ein StaticBody, der nur passierbar wird (collision off), wenn QuestManager sagt "Quest X aktiv, Tür aufmachen".
- Durch solche Mechanismen stellen wir sicher, dass **Spieleraktionen spürbare Auswirkungen** haben – die Welt "reagiert" auf die Story. Dies steigert die Immersion erheblich.
- **Quest-UI (Log):** Um dem Spieler Überblick zu geben, bauen wir ein einfaches Questlog-UI. Z.B. ein Fenster mit einer Liste aktiver Quests, jeweils Titel und vlt. Kurzbeschreibung/Ziel. Der QuestManager liefert dafür Daten. Dieses UI lässt sich per Taste (z.B. "L") öffnen. Optionally zeigen wir auch Zielmarker in der Welt (z.B. ein Pfeil oder Markierung am Rand des Screens für die nächste Quest-Location).
- **Testing mit Beispiel-Questreihe:** Wir sollten nun einen kleinen **Story-Durchlauf** testen:
- Startquest vom Bürgermeister annehmen (Dialog -> Quest start, Welt hat das Dungeon schon gesetzt aus Phase 3).
- In den Waldtempel gehen (Trigger -> Kampf mit Boss, Boss stirbt -> Questziel erfüllt, droppt Amulett).
- Zurück zum Bürgermeister, Dialog erkennt Quest erledigt (QuestManager sagt Completed, Dialog hat entsprechende Zeile "Gut gemacht!").
- Belohnung erhalten (z.B. XP, Item).

Bei diesem Durchlauf beobachten: Hat die Queststeuerung funktioniert? Sind ggf. Questschritte in falscher Reihenfolge möglich (was wenn Spieler direkt in Tempel geht vor Annahme? – evtl. QuestManager muss verhindern, dass Boss spawnt bevor Quest aktiv; oder er spawnt trotzdem, dann kann man Quest even vor Annahme erfüllen -> auch handhabbar: QuestManager merkt, Boss tot ohne

Quest = lege Item hin, Quest kann sofort abgeschlossen werden beim Annehmen, etc. Diese Edge-Cases bedenken).

- **Polish und Verbinden aller Systeme:** In dieser Phase lösen wir auch letzte Verbindungsprobleme:
- Crafting-Quest Verbindung: Wenn Quest will, dass man einen Gegenstand craftet (z.B. "Braue einen Heiltrank"), stellen wir sicher QuestManager erkennt das (vielleicht einfacher: Quest wird abgeschlossen durch Dialog "Ich habe den Trank gebraut", aber das setzt voraus, dass Item im Inventar).
- Multiplayer (falls relevant für Quests): In Koop müssten Quests für alle zählen. Der Host könnte QuestManager hosten und Clients bekommen Updates. Das vertiefen wir in Phase 9.
- Fehlerfälle: Was, wenn Spieler einen wichtigen NPC tötet? (Vielleicht unmöglich machen – Friendly NPCs unverwundbar oder laufen weg statt sterben).
- **Speichern/Laden**: Spätestens jetzt das Speichersystem implementieren, falls geplant, weil alle wichtigen Daten (Welt-Seed, veränderte Blöcke, NPC Zustände, Quests, Inventar) nun existieren. Save in Godot z.B. via `File` API zu JSON.

Nach Phase 8 sollte das Spiel einen durchgängigen **Storymodus** erlauben: Der Spieler kann Quests erhalten und abschließen, die Welt erkunden, kämpfen, Items craften und die Story vorantreiben. Alle Systeme sind nun eng verzahnt: Quests beeinflussen die Welt und NPCs, Dialoge hängen von Queststatus ab, Kämpfe und Crafting liefern Quest-relevante Ergebnisse. Das Spiel ist inhaltlich komplett (wenn auch noch nicht optimiert).

Phase 9: Multiplayer-Grundlagen (Optional)

Ziel dieser Phase: Vorbereitung und ggf. Implementierung eines einfachen **Multiplayer-Koop-Modus**. Da Multiplayer nur optional angedacht ist, kann dieser Schritt parallelisiert oder nach dem Singleplayer erfolgen. Dennoch ist es sinnvoll, früh zumindest die Architektur multiplayer-fähig zu gestalten.

Schritte für Multiplayer:

- **Architektur-Entscheidung:** Wahrscheinlich **Peer-to-Peer mit Host**: Einer der Spieler eröffnet ein Spiel als Host (Server) und andere treten bei. Für Coop (z.B. 2-4 Spieler) ist das ausreichend. Eine Dedicated-Server-Struktur wäre komplexer und fürs erste nicht nötig.
- **Synchronisationsstrategie:**
- **Deterministische Welt:** Unsere prozedurale Weltgenerierung kann auf allen Clients mit gleichem Seed ablaufen, so dass die statische Welt nicht über Netzwerk übertragen werden muss. Das heißt: Wenn alle Mitspieler das Spiel starten, generieren sie lokal identische Welten (vorausgesetzt, Quests/Story-Seed ist auch gleich). Damit müssen nur **dynamische Änderungen** synchronisiert werden: Spielerbewegungen, Platzieren/Zerstören von Blöcken, NPC-Spawns/Zustände, Questfortschritt, Inventar.
- **State Replication:** Godot 4 bietet *MultiplayerSynchronizer* und *MultiplayerSpawner* Nodes, die man an Nodes hängen kann, um deren Property-Änderungen und Spawning automatisch über RPC zu synchronisieren. Wir könnten dem Spieler-Character-Node einen MultiplayerSynchronizer geben – dann werden Position, Animationen etc. an alle geschickt. Für Voxel-Terrain-Änderungen: eventuell schwierig große Datenmengen (viele Blöcke). Besser: Wichtige Änderungen, wie "Block X an Position Y entfernt" als RPC senden, und alle Clients führen es lokal aus (ihr VoxelPlugin ändert den Chunk). So bleibt es minimal.
- **Authority:** Im Coop kann der Host die authoritative Instanz für kritische Dinge sein (Kampfresultate, Questabschluss), um Konsistenz zu gewährleisten. Oder man synchronisiert

Zustände peer-to-peer. Godots high-level API erlaubt es, dass z.B. der QuestManager nur auf Host existiert und Clients remote aufrufen "Quest erledigt", Host setzt state und sync zu allen.

- **Netzwerk-Tech in Godot nutzen:**
- Ein simples Lobby-System: Host drückt "Spiel starten (Host)", Godot `create_server(port)`. Client drückt "Beitreten", `join_server(ip,port)`. Austausch ggf. über IP manuell oder Matchmaking außen vor (für lokal reicht direct IP).
- Synchronisation einbauen: Für die wichtigsten Nodes (Player, NPCs, vielleicht Monster) setzen wir `node.set_multiplayer_authority(...)` und nutzen `rpc` oder `rpc_id` Aufrufe in relevanten Funktionen. Beispiel: Die Player-Bewegung könnte 20x pro Sekunde die Position über RPC senden (oder besser, wir nutzen die eingebaute Tick-Synchronisation des MultiplayerSynchronizer).
- **Multiplayer Testing**: Godot erlaubt das Starten mehrerer Instanzen im Debug zum Testen. Wir sollten in einem kleinen Level den Sync testen: Zwei Spieler sehen sich, können herumlaufen und springen, ein NPC läuft rum, beide sehen ihn. Wenn einer einen Block abbaut, verschwindet er bei beiden. Das sind viele Ecken, aber man fängt klein an (erstmal nur Positionssync).
- **Koop-Gameplay-Aspekte:**
- *Quests:* Entscheiden, ob Quests pro Spieler oder gemeinsam sind. Coop legt nahe: Quests werden geteilt. Also wenn Spieler 1 Quest annimmt, bekommt Spieler 2 sie auch als aktiv. Der QuestManager (nur auf Host) verwaltet es und sendet Updates (kann z.B. per RPC(`all`, quest_data) die Logeinträge aktualisieren).
- *Kampf:* Rundenbasiert mit mehreren Spielern: Der CombatManager muss dann beide Spieler in die turn_order nehmen. Evtl. können sie nacheinander agieren. Implementierung mit Netzwerk: Input von beiden koordinieren. Hier könnte man rundenweise jeweils nur dem aktiven Spieler die Kontrolle geben. Auf Client-Seite kann man Logic trennen: Der Host rechnet die Kampflogik, Clients schicken nur "ich wähle Angriff". Dann Host entscheidet Treffer und verteilt Schaden (via RPCs). So bleiben alle synchron.
- *Friendly Fire?* Coop ist meist gemeinsam gegen KI, also vermutlich kein PvP – wir ignorieren also Spieler-gegen-Spieler Schaden.
- *Inventar tauschen:* Sollten Spieler Items handeln können? Könnte man erlauben: z.B. ein Handels-UI oder sie werfen Items auf den Boden (Dropping item -> spawnt Pickup Node -> sync to all).
- **Limitierung für Mobile:** Falls ein späterer Mobile-Port, bedenken wir, dass Mobile Multiplayer meist online mit Server oder Local WLAN wäre. Performance auf Mobile ist begrenzt, also vielleicht nur 2 Player auf Mobile, nicht 4.
- **Optionalität wahren:** Da Multiplayer optional ist, kann diese Phase auch nur teilweise umgesetzt werden. Wichtig ist aber: Das Spiel sollte auch im Singleplayer sauber laufen, wenn Multiplayer-Code vorhanden ist. Also überall, wo wir `rpc` nutzen, sicherstellen, dass es im Singleplayer (no network) keine Fehler gibt. Godots APIs sind meist robust (RPCs ohne Verbindung tun einfach nichts). Wir könnten via `Engine.is_network_server()` bestimmte Logik toggeln.
- **Testen & Debuggen:** Multiplayer ist fehleranfällig – viel Testen mit mehreren Instanzen: Synchronisieren sich Quests korrekt? Was passiert, wenn ein Spieler mitten im Kampf disconnected? (Vielleicht out of scope, aber zumindest Crash verhindern). Kollisionen der Spieler – Godot Kinematic Collisions sind nicht automatisch synchron (wenn zwei Spieler sich schubsen sollten, das ist nochmal komplex; evtl. ignorieren wir Kollision zwischen Spielern einfach).

Nach Phase 9 wäre unser Spiel im Idealfall für 2+ Spieler im Koop erlebbar. Das Hauptaugenmerk liegt aber auf Singleplayer – Multiplayer bietet einen zusätzlichen Wiederspielwert. Falls die Zeit knapp ist, kann man diesen Part auch rudimentär lassen (nur Bewegung/Position syncen) oder als "späteres Update" vorsehen.

Phase 10: Optimierung, Skalierbarkeit und Feinschliff

Ziel dieser Phase: Zum Abschluss werden Performance-Optimierungen durchgeführt, um Skalierbarkeit (gerade wegen der voxelbasierten Welt und vielen Systemen) sicherzustellen. Außerdem plattformspezifische Anpassungen (für möglichen Mobile-Port) und genereller Feinschliff (UX, Grafik, Sound) vorgenommen. Diese Phase ist oft iterativ und teils parallel zu vorherigen Phasen.

Wichtige Aspekte:

- **Voxel-Performance & Speicher:** Profiler-Analyse der Voxelwelt: Wie viele Draw Calls, Tris etc. haben wir bei maximaler Sichtweite? Können wir diese reduzieren?
- Möglichkeiten: **LOD aktivieren** (wenn nicht schon), um weit entfernte Landschaft als vereinfachtes Mesh zu haben.
- **Frustum Culling** nutzt Godot automatisch pro MeshInstance – unsere Chunks sind MeshInstances, also okay. Aber falls viele Chunks im View frustrum sind, hilft nur LOD oder begrenzte Sicht.
- **Occlusion Culling:** In dichten Wäldern oder Dungeons könnte man Occluder einbauen (Godot 4 hat Occlusion culling für MeshInstances). Voxel-Plugin generiert evtl. keine Occluder von sich aus, aber in einem Dungeon-Szenario könnten wir eigene Occluder planes setzen.
- **Meshing optimieren:** Wir überprüfen Chunk-Größe. Größere Chunks = weniger Nodes, aber jedes Chunk-Mesh-Update ist teurer. Kleinere Chunks = flexibleres Update, aber mehr Draw Calls. Evtl. Benchmark mit 16^3 vs 32^3 Chunks.
- **Multithreading**: Sicherstellen, dass Generierung in Threads läuft, damit Mainthread nicht stottert. Zylanns Plugin macht das, sonst müssten wir eigene Thread-Pools nutzen (Godot bietet `Thread` class; wir könnten für Weltgenerierung welche spinnen).
- **Memory:** Voxelwelten können viel RAM fressen. Prüfen, dass nicht zu viele Chunks im Speicher bleiben. Evtl. einen Cache-Limit implementieren: z.B. max 100 Chunks loaded; älteste entladen wenn drüber.
- **NPC/KI-Performance:** Wenn sehr viele NPCs vorhanden sind, kann das `_process` stark belasten. Daher:
- Weit entfernte NPCs deaktivieren (haben wir in Phase 4 bedacht).
- **Physics**: Viele NPCs als Physics Bodies können teuer sein. Nicht alle NPCs brauchen Physik – z.B. stationäre NPCs könnten simple Area-Detections statt Kinematic benutzen. Aber wir hielten NPCs als CharacterBodies, das ist okay solange <100 aktiv gleichzeitig.
- Pathfinding: Falls wir Navmesh nutzen, Updaten wir es nur, wenn Terrain sich ändert (bei voxel Zerstörung). Große dynamische Änderungen lieber selten. A* pathfinding auf Grid mit 1000+ nodes ist okay, aber 100k nodes weniger – also vllt Grid-Kacheln pro 1x1m statt pro 0.25m Block.
- **Rendering & Mobile:** Auf Mobile-Geräten müssen wir eventuell die Grafik runterschrauben:
- Godot 4.4 auf Mobile nutzt GLES3 Renderer fallback, viele Vulkan-Features (Compute Shaders, GI) gehen nicht. Wir implementieren eine **Grafik-Einstellungsoption**: z.B. *Low*, *Medium*, *High*. Low: reduzierte Sichtweite, kein GI (stattdessen baked lightmaps falls möglich), einfachere Shader (Voxelmaterial vllt unlit or vertex-lit). High: schöne Schatten, GI, Partikel.
- UI auf Mobile: Touchsteuerung einplanen (virtuelle Joysticks). Das kann in Phase 10 eingebaut werden. Evtl. separate InputMapping für Touch vs Maus/Tastatur.
- Performance auf mobilen SoCs: Voxel meshes könnten dort schwer sein. Evtl. muss Welt kleiner sein oder weniger detailliert (Blockgröße etwas größer für Mobile build, oder bestimmte heavy Features abschalten).
- **Memory & Leaks:** Durchlaufen des Spiels mit einem Profiler, auf der Suche nach Memory-Leaks (z.B. Node nicht freed?). Godot free't Nodes die out-of-tree sind nicht automatisch, daher achten, dass z.B. Partikeleffekte oder getötete NPCs irgendwann entfernt werden.
- **Polishing:**

- **Grafik verbessern:** Voxel-Optik aufpeppen – evtl. Screen-Space Ambient Occlusion (SSAO) um die Ecken der Blöcke dunkler zu machen, das gibt Tiefe. Post-Processing wie Bloom für magische Effekte.
- **Sound hinzufügen:** Schritte des Spielers, Kampfgeräusche, Musik in verschiedenen Gebieten, UI-Sounds. Sound beeinflusst zwar nicht direkt Features, aber gehört zum Feinschliff.
- **Feedback & UX:** Spielertests machen, schauen ob Quests klar verständlich sind, ob das Crafting-Menü intuitiv ist, usw. Kleine Verbesserungen: Markierungen auf der Minimap, Tutorial-Texte am Anfang, anpassbare Einstellungen (Lautstärke, etc.).

- **Bugfixing:** Alle während der Entwicklung gesammelten Bugs beheben. Insbesondere Multiplayer-Bugs, die tricky sein können (Desync etc.), ausmerzen.

- **Skalierbarkeit**: Wenn das Spiel expansionsfähig sein soll (neue Gebiete, mehr Spieler online, etc.), überlegen wir modulare Erweiterbarkeit. Z.B. durch Daten getriebene Ansätze (neue Rezepte einfach per JSON hinzufügen, neue Quests ohne Code). Das liegt jedoch mehr im Bereich zukünftiger Updates.

Nach Phase 10 ist das Projekt bereit für einen Release: Die Performance ist optimiert, sodass es auch auf durchschnittlicher Hardware flüssig läuft, ein erster Testlauf auf Mobile ist gemacht (ggf. Grafik runtergeschraubt aber lauffähig), und alle Systeme greifen stabil ineinander.

Damit haben wir den Weg von der Idee zu einem umfangreichen voxelbasierten RPG geebnet. Natürlich können einzelne Phasen nochmals iterativ durchlaufen werden (z.B. neue Quests hinzufügen = Phase 8 Inhaltserweiterung, neue Items = Phase 7 etc.), aber die Roadmap stellt sicher, dass die Kernsysteme in sinnvoller Reihenfolge entstehen und integriert werden.

Zum Abschluss noch einige **Empfehlungen**: während der Entwicklung immer wieder **kleine Testspiele** machen, um das Zusammenspiel zu erleben. Früh Feedback einholen (Spielerfahrung, Performance). Die modulare Struktur beibehalten – so kann man Teile austauschen (z.B. anderes Dialogsystem) ohne das ganze Spiel zu zerreißen. Und speziell bei einem Voxelspiel: auf die Performance achten und lieber früh optimieren, da eine voxelbasierte Welt schnell an Grenzen kommt, wie auch die Godot-Community betont ("tausende einzelne Würfel rendern" ist teuer – besser mit optimierten Plugins arbeiten [1]). Mit diesem Plan und Godot 4.4 als Basis steht der Entwicklung eines *Cube World*-ähnlichen RPGs nichts mehr im Wege. Viel Erfolg!

Quellen: Die Planung stützt sich auf Godot-Dokumentation, Forenbeiträge und Plugin-Readmes, u.a. zu Zylanns Voxel-Engine [1], zum GDScript-Voxel-Addon Voxel-Core [2], Hinweise zur NPC-Pathfinding mittels A* [5], Godot-Dialogsysteme (Dialogue Manager) [6] und Forschung zu prozeduraler Quest-Generierung [16]. Diese Ressourcen und die Godot-Community-Erfahrungen helfen, die genannten Techniken fundiert umzusetzen.

---

[1] Help me with the game and godot - Help - Godot Forum
https://forum.godotengine.org/t/help-me-with-the-game-and-godot/48859

[2] [3] [13] [14] [15] GitHub - ClarkThyLord/Voxel-Core: Voxel plugin for the Godot game engine!
https://github.com/ClarkThyLord/Voxel-Core

[4] FastNoiseLite — Godot Engine (stable) documentation in English
https://docs.godotengine.org/en/stable/classes/class_fastnoiselite.html

5 NPCs completing different actions based on a schedule : r/godot
https://www.reddit.com/r/godot/comments/15b7ee3/npcs_completing_different_actions_based_on_a/

6 7 GitHub - nathanhoad/godot_dialogue_manager: A powerful nonlinear dialogue system for Godot
https://github.com/nathanhoad/godot_dialogue_manager

8 How would you approach creating a turn based battle system : r/godot
https://www.reddit.com/r/godot/comments/qhkt1b/how_would_you_approach_creating_a_turn_based/

9 Godot OpenRPG Learn to create turn-based games - GitHub
https://github.com/DamnWidget/godot-turn-based-rpg

10 How to Make Inventory System in Godot [Tutorial]
https://forum.godotengine.org/t/how-to-make-inventory-system-in-godot-tutorial/86730

11 Godot Docs – 4.4 branch.html
file://file-37rnLDnC6EFc6e4FPHCQi9

12 Getting Voxel Tools - Voxel Tools documentation
https://voxel-tools.readthedocs.io/en/latest/getting_the_module/

16 Procedural Quest Generation Rooted in Variety & Believability
https://dl.acm.org/doi/fullHtml/10.1145/3582437.3587181

17 18 Party-Based Strategy RPG Combat Script (Godot-4, GDScript) - Programming - Godot Forum
https://forum.godotengine.org/t/party-based-strategy-rpg-combat-script-godot-4-gdscript/51353