ChatGPT

Roll-a-Ball Gesamtanalyse & Entwicklungsempfehlungen

Aktueller Stand der Spielstruktur und Levelprogression

Basisaufbau: Roll-a-Ball: Steampunk Collector umfasst derzeit **drei vorgefertigte Level (Level 1–3)**, die als Tutorial fungieren [1] [2]. Jeder Level bringt dem Spieler neue Mechaniken bei (z.B. Bewegung, Sammeln, Sprung) in ansteigendem Schwierigkeitsgrad. So besitzt **Level 1** (einfach) typischerweise eine kleine Arena mit **5 Collectibles** [3], **Level 2** (mittel) sollte anspruchsvoller sein (geplant waren ~8 Collectibles, engere Wege, erste Hindernisse) [3], und **Level 3** (schwer) bildet den Höhepunkt mit größter Karte, ~12 Collectibles und sämtlichen Steampunk-Elementen [4]. Nach Plan soll nach Abschluss von Level 3 nahtlos in **prozedural generierte Level** übergeleitet werden [1]. Dabei soll jedes weitere Level schwieriger werden, etwa gemäß der im Dokument definierten Schwierigkeitsstufen *Easy/Medium/Hard* (z.B. 5 → 8 → 12 Collectibles) [5].

Level-Aufbau und Übergänge: Aktuell sind die Tutorial-Level als separate Unity-Szenen (`Level1.unity`, `Level2.unity`, `Level3.unity`) umgesetzt. Ein zentraler `LevelManager` in jeder Szene verwaltet die zu sammelnden Objekte und den Szenenabschluss [6] [7]. In **Level 1** und **Level 2** scheint der Übergang ins nächste Level teilweise manuell konfiguriert zu sein (im Inspektor via `nextSceneName` oder im Skript) – so sollte z.B. Level 1 den nächsten Level auf „Level2" setzen [8]. Es wurde jedoch festgestellt, dass **in Level 2 die `LevelManager`-Konfiguration unvollständig war**: `totalCollectibles` war falsch, `nextSceneName` nicht auf „Level3" eingestellt und der `difficultyMultiplier` nicht erhöht [9]. Diese Inkonsistenzen könnten dazu führen, dass **Levelübergänge fehlschlagen** oder der Schwierigkeitsanstieg nicht wie vorgesehen erfolgt. Insbesondere **nach Level 3** fehlt aktuell eine definierte Fortsetzung – im Code wird für Level 3 kein nächster Level bestimmt [10]. Dies bedeutet, dass **die automatische Progression in prozedural generierte Level noch nicht implementiert** ist (außer man setzt den Wert manuell) [11] [12]. Die Erfolgskriterien nennen explizit, dass die Kette *Level 1 → 2 → 3 → Procedural* funktionieren muss [1] – hier besteht Nachholbedarf.

Schwierigkeitskurve: Die angestrebte **progressive Steigerung** der Schwierigkeit ist in den Spieldaten bereits angelegt, aber noch nicht konsequent im Leveldesign umgesetzt. Es existieren **Scriptable Objects** für Level-Profile (Easy/Mittel/Schwer) mit vordefinierten Parametern (Levelgröße, Collectible-Anzahl, Hindernisdichte etc.) [13] [14]. So sind z.B. für **Medium** 8 Collectibles und einige Hindernisse vorgesehen [15]. In der Praxis entspricht **Level 2** diesem Profil jedoch noch nicht voll: Der **Scene Report** identifiziert, dass Level 2 mit nur 5 Collectibles und kaum Hindernissen nicht dem erwarteten mittleren Schwierigkeitsgrad entspricht [3] [16]. Ähnliches gilt für **Level 3**, der zwar 12 Collectibles hat, aber noch **nicht alle vorgesehenen Mechaniken** nutzt (keine Zeitlimits, wenig kombinierte Hindernisse) und damit das Potential des schwersten Levels nicht ausschöpft [17] [18]. Insgesamt fehlen in Level 3 noch die „Master"-Features wie **komplexe Zahnräder, multi-stufige Plattformen, volle Steampunk-Atmosphäre und eine epische Finalsequenz**, die im Design vorgesehen sind [19] [20].

Technische Szenenstruktur: Die Unity-Szenen weisen teils Uneinheitlichkeiten auf, die durch automatisierte Analyse und Reparatur adressiert wurden. Ein speziell entwickelter *SceneNormalizerAgent* hat überprüft, ob zentrale Objekte überall vorhanden und korrekt instanziiert sind [21]. Dabei wurde u.a.

kontrolliert, ob **alle Level-Objekte als Prefab-Instanzen** vorliegen (statt Duplikate), ob **UI-Canvas + EventSystem** in jeder Szene vorhanden und richtig skaliert sind sowie ob wesentliche Tags/Layers gesetzt wurden [22] [23]. Die Befunde zeigten Inkonsistenzen: Beispielsweise waren in frühen Versionen einige **Collectibles nicht als Prefabs** angelegt oder es fehlten Referenzen auf Skripte/Tags – dies konnte zu Fehlfunktionen führen [24]. Auch UI-Probleme traten auf: Der Canvas war teils nicht mit einem `CanvasScaler` versehen oder UI-Elemente falsch verankert, was die Darstellung auf unterschiedlichen Auflösungen beeinträchtigt [21]. Dank der durchgeführten **Szenen-Reparaturen** sind diese Basisprobleme in Level 1 inzwischen behoben (Level 1 gilt als konsistent) [25], während **Level 2 und 3 noch teilweise Korrekturen benötigten** (z.B. Prefab-Nutzung vereinheitlichen, UI in Level 3 reparieren) [25] [26].

Zusammenfassung Ist-Zustand: Das Spielgerüst (GameManager, LevelManager, Spielersteuerung) funktioniert in den drei festen Leveln, jedoch **entsprechen Level 2 und 3 inhaltlich noch nicht vollständig ihren Designvorgaben** (fehlende Objekte, unausgewogene Schwierigkeit). Die **Levelübergänge** müssen überprüft werden, damit nach jedem Abschluss korrekt zum nächsten Level gewechselt wird. Insbesondere die **Automatisierung nach Level 3** – Übergang zu generierten Leveln – ist noch **nicht umgesetzt** und stellt eine der größten offenen Aufgaben dar. Positiv ist, dass die grundlegende Infrastruktur (Skripte, Profile, Prefabs) für prozedurische Level bereits vorhanden ist; allerdings bedarf es weiterer Integration und Feinschliff, um die geplante progressive Spielerfahrung zu realisieren.

Prozedurale Levelgenerierung nach Level 3

Nach den drei festen Leveln sollen **endlos weitere Level prozedural generiert** werden, um den Spielspaß aufrechtzuerhalten. Hierfür wurde ein System aus **Level-Profilen** und einem **Level-Generator** implementiert. Die **LevelProfile** (Easy/Mittel/Schwer) definieren Parameter wie Größe des Grids (z.B. 8×8, 12×12, 16×16 Felder), Anzahl Collectibles (5, 8, 12) sowie Hindernisdichte und besondere Features pro Schwierigkeitsgrad [27] [14]. Ein **LevelGenerator**-Skript nutzt diese Profile vermutlich, um zur Laufzeit eine zufällige Labyrinth-ähnliche Karte mit Bodenplatten, Wänden, Collectibles etc. zu erzeugen. In der Szene `GeneratedLevel.unity` wird dieses System getestet.

Aktueller Zustand: Die prozedurale Generierung ist in Ansätzen funktionsfähig, zeigte aber zunächst einige technische Probleme, die mittels automatisierter Fixes behoben wurden. So waren anfänglich generierte **Collectibles nicht aufsammelbar**, weil **Trigger und Skriptreferenzen fehlten** oder Tags falsch waren [24]. Ebenso gab es **Inkonsistenzen bei Materialien** auf dem Boden, da Zufallsmaterialien ohne zentrale Steuerung verwendet wurden [28]. Diese Probleme wurden mit neuen Hilfsskripten gelöst: Ein **GeneratedLevelFixer** korrigiert automatisch alle Collectibles (Collider auf Trigger, Tag = "Collectible", `CollectibleController` anhängen, etc.) [29]. Gleichzeitig führt ein **GroundMaterialController** ein deterministisches Materialsystem ein, das pro Zufallslevel konsistente Steampunk-Materialien auf Böden/Wänden verteilt [30] [31]. Die erfolgreiche Anwendung dieser Fixes wird in einem Report bestätigt („VOLLSTÄNDIG BEHOBEN“) [24]. Außerdem existiert ein **SceneValidator**-Tool, das generierte Level auf Vollständigkeit prüft (Player, UI, Audio, etc.) und Abweichungen meldet [32].

Trotz dieser Fortschritte ist die **Integration des Generators in den Spielablauf** noch unvollständig. Momentan muss der `GeneratedLevel` entweder manuell im Editor gestartet oder per Skript aufgerufen werden. Der **Übergang nach Level 3 zu** `GeneratedLevel` sollte automatisiert erfolgen (z.B. durch Setzen von `LevelManager.nextSceneName = "GeneratedLevel"` in Level 3) [12]. Im Reparatur-Plan wurde genau dies notiert – Level 3's LevelManager soll `nextScene = "GeneratedLevel"` erhalten [12] [33]. Weiterhin sollte der Generator wissen, **welches**

Schwierigkeitsprofil als nächstes zu verwenden ist. Laut To-Do-Liste wird in `GeneratedLevel.unity` initial das **Easy-Profil als activeProfile gesetzt** [34], was plausibel ist, da der erste prozedurale Level nach den Tutorials vermutlich wieder einfach starten soll. Eine **dynamische Steigerung** könnte dann so aussehen: Nach Abschluss eines generierten Easy-Levels wird das Profil auf Medium gewechselt und das Level neu erzeugt; nach Medium folgt Hard, usw. – bis eventuell eine Obergrenze erreicht ist oder immer schwierigere Variationen erstellt werden. Dieser Mechanismus muss noch programmiert oder konfiguriert werden, ist aber durch die vorhandenen Profile und das `LevelManager`-Konzept vorbereitbar.

Empfohlene Verbesserungen für prozedurale Levels:

- **Automatische Levelübergabe einrichten:** Implementieren Sie im `LevelManager` von **Level 3** den Übergang zur Szene *GeneratedLevel*. Dies kann durch Setzen von `nextSceneName = "GeneratedLevel"` im Inspector oder zur Laufzeit erfolgen [12]. So wird nach dem „Victory“ von Level 3 direkt die Generierung eines neuen Levels angestoßen [35] [11].

- **Schwierigkeitsauswahl/Progression:** Sorgen Sie dafür, dass das **passende LevelProfile** für jedes neue prozedurale Level gewählt wird. Initial sollte das **Easy-Profile** verwendet werden [36]. Anschließend kann man z.B. den `difficultyLevel` hochzählen und das nächste Level mit dem nächsthöheren Profil generieren (Medium, dann Hard). Auf diese Weise steigt der Anspruch tatsächlich **progressiv mit jedem generierten Level**, wie vorgesehen. Alternativ kann auch nach einigen Levels auf Hard weiterhin die Parameter erhöhen (z.B. noch mehr Collectibles oder Zeitlimit reduzieren), um endlos zu skalieren. Wichtig ist, die im Profil definierten Werte (Größe, Hindernisdichte etc.) auch im Generator zu berücksichtigen – prüfen Sie, ob der Generator diese Parameter bereits aus einem aktiven Profil liest, oder ob dies noch implementiert werden muss.

- **Qualität der Generierung verbessern:** Obwohl das System grundsätzlich Level erstellt, sollte der **Generator** direkt möglichst valide Level bauen, um nachträgliche Korrekturen zu minimieren. Derzeit behebt `GeneratedLevelFixer` zwar die größten Mängel (Collectibles, Materialien) automatisch [37], dennoch wäre es besser, wenn *bereits beim Erstellen* alle GameObjects korrekt instanziiert würden. Zum Beispiel könnte der Generator gleich Prefab-Instanzen mit den richtigen Komponenten erzeugen (Player, GoalZone, Collectibles etc.) statt Platzhalter. Dies reduziert die Abhängigkeit von Reparaturskripten während der Laufzeit. Die vorhandenen Editor-Tools (Menü **"Roll-a-Ball > Fix Generated Level"** [38]) sind hilfreich für Entwicklung und Test, aber im fertigen Spiel sollte der Ablauf ohne manuelle Eingriffe stabil laufen.

- **Inhaltliche Varianz:** Nutzen Sie die erweiterten Möglichkeiten des Steampunk-Themas auch in generierten Levels. Aktuell fokussiert das Profil-System vor allem auf Zahlenwerte (Größe, Anzahl, Dichte). Es wäre denkbar, generierte Levels auch mit **zufälligen Steampunk-Objekten** auszustatten – z.B. ein paar **MovingPlatforms oder RotatingObstacles** selbst in prozeduralen Levels zu integrieren, sobald der Schwierigkeitsgrad steigt. Diese könnten an zufälligen Stellen platziert oder anhand der `obstacleDensity` im Profil bemessen werden. Das würde die generierten Levels abwechslungsreicher machen. Entsprechende Komponenten sind vorhanden (`MovingPlatform.cs`, `RotatingObstacle.cs` etc. [39]), wurden aber bislang vor allem für manuell designte Level vorgesehen. Eine algorithmische Platzierung dieser Elemente ist anspruchsvoll, könnte aber schrittweise angegangen werden (z.B. Zufallspunkte auf dem Pfad als Plattform markieren).

- **Testing & Balancing:** Prozedural generierte Levels erfordern gründliches Testen, da unvorhergesehene Kombinationen auftreten können. Es sollte überprüft werden, ob **alle Collectibles erreichbar** sind, der **Player nicht außerhalb der Map spawnt** usw. (Probleme, die weiter unten beim OSM-Modus auftreten, können auch hier relevant sein). Möglicherweise ist es sinnvoll, einen **Mechanismus zum Neugenerieren** einzubauen (was durch Drücken von `R` bereits im Debug vorgesehen ist [40]). Spieler könnten so ein unlösbares Level neu würfeln. Wichtig ist, dass das Spiel erkennt, wann ein Level unlösbar ist – z.B. wenn Collectibles übrig bleiben, die nicht eingesammelt werden können – und dann automatisch einen Neustart anbietet. Hier kann die Statistik aus `LevelManager` (wenn `collectiblesRemaining` nach längerer Zeit >0 bleibt) als Indikator dienen.

Zusammenfassend ist das Fundament für endlose Level gelegt, doch es bedarf einiger **automatischer Kopplungen und Feinschliffe**, damit der Übergang nahtlos wird. Mit korrekter Weiterschaltung, Schwierigkeitsprogression und Optimierung des Generators (weniger Nachkorrekturen, mehr Varianz) wird der Wiederspielwert deutlich erhöht und die im Konzept vorgesehene **endlose Herausforderung** Realität.

OpenStreetMap-Modus (OSM-Level) – Analyse und Verbesserungen

Der **OSM-Modus** stellt ein besonderes Feature dar: aus realen Geodaten sollen dynamisch Level generiert werden, basierend auf einer vom Spieler eingegebenen Adresse oder seinem aktuellen Standort. Dieses System ist **teilweise implementiert**, aber noch nicht vollständig funktionsfähig und integriert. Die entsprechende Szene `Level_OSM.unity` existiert und enthält Kernkomponenten wie `MapStartupController`, `AddressResolver`, `MapGenerator` usw. [41]. Der Ansatz ist ambitioniert, bringt jedoch diverse Herausforderungen mit sich, die im aktuellen Projektstatus deutlich werden.

Identifizierte Schwachstellen der OSM-Integration:

- **Unvollständige Initialisierung:** Beim Laden der OSM-Szene ist zwar ein `MapStartupController` vorgesehen, jedoch **haperte es an der richtigen Konfiguration** [42]. Laut Fehlerbericht könnte der Controller nicht alle Referenzen gesetzt haben (GameManager, UIController) oder bestimmte Flags nicht aktiviert sein. Dies führt dazu, dass der Ablauf des Kartenladens nicht sauber startet.

- **Defekte UI für Adresseingabe:** Das UI-Panel zur Eingabe der Adresse und zum Starten der Kartengenerierung funktioniert derzeit **nicht**. Konkret wurden folgende Probleme festgestellt [43]:

- Das Textfeld (TMP_InputField) zur Adresseingabe ist nicht mit dem Script verbunden (Eingaben triggern keine Aktion).
- Der "Karte Laden"-Button hat **keinen Event-Handler** zugewiesen – ein Klick bewirkt nichts [44].
- Der "aktueller Standort"-Button (`UseCurrentLocation`) führt zu keinem definierten Verhalten bzw. löst Fehler aus.
- Es gibt kein visuelles Feedback während eines Ladevorgangs (kein Ladebalken, keine Meldung).

- Fehlerfälle (z.B. ungültige Adresse, keine Internetverbindung) werden dem Spieler nicht mitgeteilt (fehlendes Error-Message-Display).

- **Fehlerhafte Kartenkonvertierung:** Selbst wenn OSM-Daten geladen werden, ist die **Umwandlung in ein spielbares Level derzeit mangelhaft**. Der `MapGenerator` erzeugt zwar Straßennetze und Gebäude, aber in einer Weise, die dem Gameplay im Weg steht [45]:

- **Straßen** werden als physische Objekte generiert, die offenbar unbegehbar sind oder nicht als begehbarer Boden dienen (möglicherweise als Wände/Kollisionsobjekte).
- **Gebäude** erscheinen als große Blöcke an ihrer realen Position und **blockieren Wege** komplett [46] – der Spieler hätte kein Durchkommen, da ein Stadtgebiet meist dicht bebaut ist.
- **Collectibles** werden zufällig verteilt und landen dabei teils an **unerreichbaren Stellen** – z.B. innerhalb von Gebäuden oder außerhalb der begehbaren Fläche [47].
- Der **Player-Startpunkt** wird offenbar ungeschickt gewählt (etwa am Kartenmittelpunkt oder erstbester Straßenknoten), was dazu führen kann, dass der Spieler **außerhalb des spielbaren Bereichs** startet [48] (z.B. in der Luft über einem Gebäude oder mitten in einer Straßenkreuzung ohne Rampe).
- Die **GoalZone** (Zielbereich) wird ebenso unzugänglich platziert – etwa auf einem Hausdach oder an einer Stelle, die vom Spieler nicht erreicht werden kann [49].

- Der **Maßstab** der generierten Welt könnte unpassend sein: Es besteht die Gefahr, dass die Stadt entweder viel zu groß (Ausmaße, die der Ball nie abrollen kann) oder zu klein (alles gequetscht) generiert wird [50]. Eine falsche Skalierung würde Navigation und Spielgefühl stark beeinträchtigen.

- **Performance-Probleme:** Die naive Übernahme von echten Geodaten führt zu **Leistungsengpässen**, vor allem bei größeren oder detailreichen Orten [51]:

- Städte enthalten sehr viele Objekte (Gebäude, Straßensegmente, Details). Aktuell werden *alle* aus den OSM-Daten verfügbaren Elemente generiert, was schnell die Polygonzahl und Draw Calls in die Höhe treibt [52]. Ohne Begrenzung können selbst in einem 500 m Radius Hunderte Gebäude und Straßen entstehen.
- Es gibt **keine Vereinfachung** oder Filterung der Daten für Gameplay-Zwecke – kleine Gebäude oder unwichtige Objekte werden ebenso erstellt wie große, anstatt z.B. nur ein spielerisch relevantes Gerüst zu erzeugen [52]. Das bedeutet unnötige Last.
- **LOD (Level of Detail) oder Occlusion Culling** wird nicht erwähnt – alle Gebäude sind wohl in voller Detailstufe da. Auch eine Aufteilung in Sektoren (um entferntes Auszublenden) fehlt, was die Renderlast erhöht [51].
- Die **Physik** könnte strapaziert werden: Viele Kolliders (für jedes Gebäude/Wand) und rigidbodies (falls bewegliche Teile geplant) belasten CPU und Speicher.

- **Speicherverbrauch:** Große städtische Gebiete können sehr viele Datenpunkte haben. Wenn diese alle in Meshes umgesetzt werden, droht ein Memory-Overflow oder zumindest hoher RAM-Verbrauch [53]. Tatsächlich warnt auch die Overpass-API-Dokumentation, dass Abfragen großer Ausschnitte sehr lange dauern und viel Daten zurückliefern [54]. Dies kann zu spürbaren **Ladezeiten** oder sogar Timeout-Fehlern führen, wenn keine Begrenzung implementiert ist.

- **Fehlende Gameplay-Integration:** Der momentan generierte Stadtkartenausschnitt ist **rein statisch** und berücksichtigt keine Spielmechaniken [55]. Es fehlen:

- **Gezielte Collectible-Platzierungen:** Statt Collectibles blind um Gebäude zu streuen, sollten sie an interessanten Orten sein. Etwa könnten OSM-**Points of Interest (POIs)** genutzt werden – z.B. Wahrzeichen, Parks oder besondere Gebäude, um dort Sammelobjekte zu positionieren [55]. So hätte der Spieler klare Ziele in der Stadt.

- **Steampunk-Adaptation:** Die reale Karte wird nicht ins Spiel-Theme übersetzt. Es gibt z.B. **keine Deko-Elemente** wie Zahnräder oder Steam-Emitter an den Gebäuden, die aber das Steampunk-Flair ausmachen könnten. Auch **Ambient-Sounds** (Dampfgeräusche in der Stadt) fehlen bisher, obwohl das Erlebnis damit viel stimmungsvoller würde [56].

- **Leveldesign-Aspekte:** In einem guten Level gibt es üblicherweise geplante Routen, Abkürzungen oder Rätsel. Im OSM-Level fehlen solche Konzepte (verständlicherweise, da die Stadt roh ist). **Keine alternativen Routen** oder Abzweigungen wurden hervorgehoben – das Straßennetz ist zwar vorhanden, aber ob es spielerisch genutzt wird, ist fraglich [57]. Auch gibt es kein **Scoring** oder besondere Aufgaben, die an die reale Umgebung geknüpft sind (z.B. „Finde das Rathaus und sammle dort X ein" o.ä.).

- **Fehlerbehandlung und Fallbacks:** Wenn z.B. die Adresse nicht gefunden wird oder die API nicht erreichbar ist, versucht das System auf einen Fallback zurückzugreifen. In den Einstellungen ist ein **Fallback-Ort (Leipzig)** mit Koordinaten hinterlegt [58]. Der Code von `AddressResolver` zeigt ebenfalls, dass bei Fehlern auf einen Default (Berlin) umgeschaltet wird [59]. Allerdings muss geprüft werden, **wie robust diese Fallback-Mechanismen greifen**. Momentan werden Fehler nur teilweise abgefangen (z.B. Abfragefehler lösen einen Wechsel auf die Fallback-Koordinate aus) [59]. Ein umfassendes Fehlermanagement (mit Benutzerinfo) fehlt noch: Der Spieler sollte z.B. eine Meldung erhalten „Adresse nicht gefunden, generiere stattdessen Leipzig" – so etwas ist bislang nicht umgesetzt (OnError-Event wird geworfen, aber nicht sichtbar gemacht) [60] [61].

Empfehlungen zur Weiterentwicklung der OSM-Integration:

1. **UI und User Experience fixen:** Zunächst muss die **Benutzeroberfläche des OSM-Modus vollständig repariert** werden, damit der Spieler Adressen eingeben und den Ladevorgang starten kann [43]. Konkret:
2. **Event-Verknüpfung:** Verdrahten Sie das TMP_InputField und die Buttons mit den dafür vorgesehenen Methoden. Der `MapStartupController` bzw. `OSMUIConnector` sollte z.B. `LoadMapButton.onClick` mit der Funktion verknüpfen, die `AddressResolver.ResolveAddressAndLoadMap(inputText)` aufruft [62] [63]. Ähnlich muss der „Standort"-Button die GPS-Funktion triggern (falls implementiert) oder zumindest ein vordefiniertes Beispiel laden.
3. **Visuelles Feedback:** Implementieren Sie ein einfaches **Lade-Feedback**. Im UI-Layout sind Elemente wie *LoadingPanel*, *LoadingText* und *ProgressBar* vorgesehen [64] [65] – nutzen Sie diese. Beispielsweise kann `AddressResolver` während `LoadMapDataCoroutine` den Fortschritt (z.B. „Lade Kartendaten…") anzeigen und die ProgressBar inkrementell füllen. Bei Fehlern (OnError/Event) sollte eine deutliche Meldung wie „Adresse nicht gefunden" oder „Netzwerkfehler – zeige Standardkarte" im UI erscheinen [66]. Solches Feedback ist wichtig, damit der Spieler weiß, was passiert, statt ein eingefrorenes Spiel anzunehmen.

4. **Responsive Design:** Stellen Sie sicher, dass der Canvas mit **CanvasScaler (Scale With Screen Size)** eingerichtet ist und alle UI-Elemente korrekt verankert sind [21]. Gerade im OSM-Panel mit mehreren Buttons sollte in verschiedenen Auflösungen nichts abgeschnitten sein. Dies wurde als Problem genannt und kann leicht durch Anchor-Presets behoben werden.

5. **Stabilität & Fehlerbehandlung:** Die Kommunikation mit den externen OSM-APIs muss robust gestaltet werden:

6. **Timeouts und Retry:** Integrieren Sie sinnvolle **Timeouts** und ggf. Wiederholungslogik für Web Requests. Der `AddressResolver` nutzt Nominatim und Overpass mit je ~10 Sek. Timeout [67] [68]. Falls häufig Timeout auftritt, könnte man *einen Retry mit kleinerem Radius* versuchen oder auf einen anderen Overpass-Server ausweichen. Wichtig: Falls alle Stricke reißen, den Spieler nicht hängen lassen, sondern per Fehlermeldung informieren und auf Offline-Mode umschalten.
7. **Fallback-Karte laden:** Die Idee, bei Fehlern eine Default-Location zu laden, ist sinnvoll. Diese sollte voll funktional sein (z.B. Leipzig Innenstadt, wie in den Einstellungen gesetzt [58]). Testen Sie diesen Flow: Geben Sie eine absichtliche Quatsch-Adresse ein und prüfen, ob danach die Leipzig-Karte generiert wird und spielbar ist. Dokumentieren Sie diesen Mechanismus auch für den Nutzer („Bei unbekannten Adressen wird eine Beispielkarte geladen.“).

8. **Keine Abstürze:** Stellen Sie sicher, dass Fehler im Parsing der OSM-Daten das Spiel nicht komplett stoppen. Im Report wird erwähnt, dass `OSMMapData.cs` Parsing-Errors bei komplexen Daten auslösen könnte [69]. Solche Exceptions sollten abgefangen (try-catch) und z.B. als OnError-Event ausgegeben werden, anstatt ein Freeze oder Crash zu verursachen.

9. **Map-Generierung für Gameplay optimieren:** Damit die generierte Stadt **spielbar** wird, muss der `MapGenerator` intelligenter filtern und vereinfachen [70]:

10. **Straßen begehbar machen:** Anstatt jedes Straßenobjekt mit Collider zu erzeugen (was den Ball stoppen würde), könnten Straßen nur als **Textur auf dem Boden** oder als flaches Mesh ohne Collider generiert werden. So dienen sie als visuelle Orientierung, behindern aber nicht. Alternativ könnten Straßen als **begehbare Korridore** interpretiert werden: d.h. alles außer Straßen ist Wand/Obstacle. Diese Design-Entscheidung muss getroffen werden. Eine Idee: **Generiere ein begehbares Labyrinth aus dem Straßennetz**, indem Straßen zu Gängen werden und Gebäude zu Mauern. Dazu müsste man die Geometrie umkehren (Straßenflächen als begehbar markieren, restliche Fläche als Hindernis). Eine einfachere Variante: Platziere den Ball auf Straßenebene und lösche alle Gebäude außer ggf. dem äußersten Rand, sodass ein „Labyrinth“ aus Stadtblöcken entsteht.
11. **Gebäude vereinfachen:** Anstelle komplexer extrudierter Gebäude (mit zig Polygonen) sollten **vereinfachte Kollisionskörper** genutzt werden. Für Gameplay reicht es oft, Gebäude als einfache Quader (Bounding Box) hinzustellen oder sogar nur als **umrissartige Wände** entlang der Straßen. Dies würde mehr Freifläche fürs Rollen schaffen. Der Generator könnte z.B. nur Gebäude über einer gewissen Größe tatsächlich zeichnen und kleinere ignorieren, um das Spielfeld offener zu gestalten [52]. Außerdem könnte man **Durchgänge** einbauen: eventuell gezielt Lücken in Häuserblocks lassen, damit alternative Routen entstehen (z.B. Parks oder Plätze in OSM erkennen und diese als Freifläche belassen).
12. **Maßstab anpassen:** Experimentieren Sie mit dem **Maßstab** der Welt. Möglicherweise muss der Maßstab verkleinert werden (z.B. 1 Unity-Einheit = 2 m statt 1 m), damit eine Innenstadt in eine spielbare Fläche passt und der Ball in angemessener Zeit von A nach B rollen kann. Der `MapGenerator` rechnet Koordinaten aktuell 1:1 in Unity-Einheiten um [71] [72], was reale Distanzen bedeutet. Gegebenenfalls sollte man den Radius der geladenen Karte verkleinern (z.B. 300 m statt 500 m), um einen dichteren, aber kompakteren Level zu erhalten.
13. **Collectible- und Ziel-Platzierung:** Entwickeln Sie eine **intelligentere Platzierungslogik** für Gameplay-Objekte:
    - Lassen Sie **Collectibles an POIs oder markanten Orten** erscheinen [55]. Der `AddressResolver` liefert möglicherweise interessante Punkte (z.B. Overpass-Query nach `amenity=...`). Falls nicht, könnten zumindest große Gebäude oder Plätze als Orientierung dienen (z.B. **Collectible auf dem Marktplatz, in einem Park, auf einer Brücke** etc.). Auch die in OSM hinterlegten Tags (wie `tourism=attraction`) könnten genutzt werden, um besondere Punkte fürs Spiel zu definieren.

- Stellen Sie sicher, dass **Collectibles nicht in Wänden** schweben. Evtl. muss man nach der ersten Platzierung eine Validierung vornehmen: Liegt ein Collectible-Koordinatenpunkt innerhalb eines Gebäudepolygons? Dann versetze ihn etwas nach außen. Derzeit werden Collectibles einfach random um Gebäude gesprüht [73] [74], was oft unglücklich ist. Eine Verbesserung wäre, **Collectibles entlang der begehbaren Bereiche** (Straßen) oder auf freien Plätzen zu verteilen.
- **Start- und Zielposition**: Überarbeiten Sie die Logik zur Spieler-Spawn-Position und zum Ziel. Im Code wird der Player entweder ins Zentrum oder an den ersten Straßenknoten gesetzt [75], was suboptimal ist. Besser wäre, einen **freien Bereich** zu suchen – etwa einen Ort am Kartenrand oder einen Platz – wo der Ball sicher spawnen kann, ohne sofort mit einem Gebäude zu kollidieren. Ebenso sollte die **GoalZone** an einem erreichbaren Ort liegen, idealerweise weit genug vom Start, um eine Herausforderung darzustellen. Derzeit wird sie am größten Gebäude ausgerichtet [76] [77], was oft schwer zugänglich ist. Überlegen Sie stattdessen, das Ziel z.B. **am Rand der Karte** zu platzieren (so wird der Spieler gezwungen, die Stadt zu durchqueren), oder an einem **Point of Interest** (z.B. bei einer bekannten Sehenswürdigkeit der Stadt).

14. **Steampunk-Elemente einfügen:** Um den OSM-Level in das Spielthema einzubetten, sollten **Steampunk-Assets** eingebracht werden. Beispielsweise könnten in der generierten Stadt vereinzelt **bewegliche Zahnräder** oder **Dampfmaschinen** platziert werden – etwa an Stellen, wo in der Realität Industriebauten sind, oder zufällig an Kreuzungen als Hindernis. Der `MapGenerator` hat bereits Prefab-Felder für `gearPrefab`, `steamPipePrefab` etc. vorgesehen [78], was andeutet, dass solche Dekoration geplant ist. Nutzen Sie diese: z.B. könnte jede X-te Straße ein rotierendes Zahnrad aufweisen oder einige Hausdächer bekommen Rauchfang-Effekte. Auch **Lichtquellen** (Gaslampen) könnten an Straßenlaternen oder Gebäudeecken erscheinen, um den Level atmosphärisch auszuleuchten. Diese Elemente steigern die Immersion und machen klar, dass man sich trotz realer Karte in einem Steampunk-Spiel befindet.

15. **Performance-Optimierung:** Um den OSM-Level flüssig spielbar zu machen, sind Performance-Techniken erforderlich:

16. **Datenmenge begrenzen:** Begrenzen Sie die Größe des geladenen Kartenausschnitts je nach Gerät. 500 m Radius in einer Großstadt sind sehr viel – eventuell kann der Radius dynamisch verkleinert werden, falls die abgerufene Datenmenge eine gewisse Grenze überschreitet (der `AddressResolver` könnte die Anzahl gefundener Objekte prüfen und bei >X Objekten z.B. weniger laden). Overpass bietet Filtermöglichkeiten (z.B. nur Hauptstraßen laden). Nutzen Sie so etwas, um **nur relevante Map-Daten** zu holen (Hauptstraßen, größere Parks/Gebäude) statt jedes Wohnhaus [52].
17. **Mesh-Batching:** Der `MapGenerator` sammelt schon pro Typ CombineInstances, um Meshes zusammenzufassen [79] [80]. Stellen Sie sicher, dass nach Generierung diese Combined Meshes auch angewendet werden (Aufruf von `ApplyMeshBatching()` etc. am Ende) und die Einzelobjekte ggf. entfernt werden. Static Batching in Unity könnte hier helfen, jedoch sind viele Objekte vermutlich dynamisch generiert und nicht als static markiert. Daher ist das manuelle Kombinieren im Generator sinnvoll – z.B. alle Straßen eines Typs zu einem Mesh, alle Gebäude eines Typs zu einem Mesh bündeln. Das reduziert Draw Calls erheblich.
18. **LOD und Culling:** Fügen Sie für Gebäude vereinfachte LOD-Stufen hinzu, zumindest zwei: In der Ferne nur simple Boxen oder ausgeblendet, nahe in voller Pracht. Unity's LOD Group kann programmatisch hinzugefügt werden. Ebenso sollte **Occlusion Culling** genutzt werden, da in einer Stadt viele Gebäude die Sicht versperren – Berechnen Sie Occlusion Daten für die

generierten Objekte oder nutzen Sie Sektorisierung (Quad-Tree), um entfernte Stadtteile nicht immer zu rendern [81] [82].

19. **Physics & Objekt-Pooling:** Überlegen Sie, ob wirklich *jedes* Gebäude einen Collider braucht. Eventuell reichen Collider nur an den Umrissen der Straßen (City block Colliders) statt pro Gebäude. Reduzieren Sie die Anzahl aktiver physikalischer Objekte. Für Partikel und Effekte wie Dampf nutzen Sie **Object Pooling**, damit nicht zu viele GameObjects erzeugt und zerstört werden, was die Garbage Collection belasten würde [83].

20. **Asynchrone Generierung:** Der MapGenerator arbeitet bereits Coroutine-basiert, was gut ist [84] [85]. Achten Sie darauf, dass während der Generierung die Framerate stabil bleibt – ggf. die Arbeit noch feiner auf Frames verteilen (Parameter `maxBuildingsPerFrame`, `maxRoadSegmentsPerFrame` gibt es bereits [86]). So friert das Spiel weniger ein. Für sehr große Städte könnte man *Streaming* in Betracht ziehen: Also immer nur einen Stadt-Sektor generieren, sobald der Spieler sich nähert (dies ist jedoch sehr komplex und evtl. über das Ziel hinaus).

21. **Gameplay-Integration & Testing:** Abschließend muss der OSM-Level ins Gesamtspiel eingebunden und getestet werden:

22. **LevelManager-Unterstützung:** Stellen Sie sicher, dass `GameManager` und `LevelManager` auch im OSM-Level vorhanden sind und ihre Aufgaben erfüllen (Collectibles zählen, Levelende erkennen). Aus dem Debug-Report geht hervor, dass GameManager, Canvas etc. in Level_OSM *vorhanden* sind [87]. Testen Sie, ob beim Einsammeln aller Collectibles `LevelManager` das Level als abgeschlossen meldet und theoretisch eine nächste Szene laden würde (hier wäre vielleicht Rückkehr ins Menü oder Neuladen der OSM-Karte angebracht). Falls nicht, muss der LevelManager ggf. angepasst werden, um mit dynamischer Collectible-Anzahl klarzukommen.
23. **Regenerations-Feature:** Da reale Karten manchmal ungünstig ausfallen, sollte der Spieler die Möglichkeit haben, die Karte neu zu generieren. Im UI-Layout ist ein "RegenerateMapButton" vorgesehen [88] [89]. Implementieren Sie dessen Funktion (z.B. ruft er `MapGenerator.RegenerateMap(currentMapData)` auf, oder lädt dieselbe Szene neu). So kann man bei Bedarf einen neuen Seed oder einen veränderten Radius ausprobieren, ohne das gesamte Spiel zu verlassen.
24. **Menü-Integration:** Binden Sie den OSM-Modus ins Hauptmenü ein (so vorhanden). Der Spieler sollte vor Spielstart wählen können, ob er die **Kampagne (Level 1–3 + Endless)** spielen will oder den **OSM-Freeplay-Modus**. Alternativ könnte der OSM-Level erst freigeschaltet werden, nachdem Level 3 geschafft wurde, um es als Bonus-Modus zu präsentieren.
25. **Umfangreiche Tests:** Testen Sie den OSM-Modus mit verschiedenen Orten (Großstadt vs. Kleinstadt vs. ländlich). Prüfen Sie, ob das Spiel in allen Fällen spielbar bleibt. Achten Sie auf Spezialfälle – z.B. Orte ohne Straßen (dann keine Wege?), Orte mit viel Wasser (soll der Ball schwimmen können? Evtl. Wasser als Trigger implementieren, der den Spieler zurücksetzt [90]). Die Erfahrungswerte aus diesen Tests sollten dann ins Design einfließen (z.B. bestimmte Geländetypen ausschließen oder spezielle Logik dafür vorsehen).

Zusammengefasst ist der OSM-Modus ein innovatives Feature, das jedoch derzeit **noch erhebliche technische und designbedingte Probleme** hat. Priorität hat, die **Grundfunktionalität (Adresseingabe → spielbares Level)** zuverlässig zum Laufen zu bringen. Danach kann man iterativ die **Spielqualität** der generierten Level verbessern (vereinfachtes Leveldesign, Steampunk-Atmosphäre, Performance-Tuning). Es ist ratsam, sich ggf. an etablierten Lösungen zu orientieren – Tools wie CityGen3D zeigen, dass Städte in Unity generiert werden können, aber immer mit starker Vereinfachung und Optimierung. Letztlich soll der OSM-Modus Spaß machen und dem Spieler ein einzigartiges Erlebnis bieten, anstatt eine 1:1-Stadtsimulation zu sein. Dies erfordert kreatives

Nacharbeiten der rohen Kartendaten in Richtung „Spielplatz“, was mit den oben genannten Maßnahmen erreicht werden kann.

Dokumentation und Konsistenzüberprüfung

Bei der Analyse fiel auf, dass die **Projekt-Dokumentation (automatisch generierte `.md`-Dateien)** zwar umfangreich ist, aber inhaltlich nicht immer den tatsächlich implementierten Stand widerspiegelt. Es existieren detaillierte Berichte und Anleitungen (z.B. README, SceneReports, Phasen-Abschlussberichte), die teils **Widersprüche, veraltete Informationen oder KI-bedingte Ungenauigkeiten** enthalten. Hier einige Auffälligkeiten und Empfehlungen:

- **Überschätzung des Fertigstellungsgrads:** Mehrfach wird in der Doku ein Status *„complete“* oder *„vollständig implementiert“* angegeben, der durch die Analyse der Szenen relativiert werden muss. Zum Beispiel behauptet der *Phase 4 OSM Setup Guide*, das OSM-System sei **„vollständig implementiert und bereit zur Nutzung“** [91], was offenkundig zu optimistisch ist, da ja diverse OSM-Probleme (UI defekt, Gameplay unspielbar) existieren [42] [45]. Ähnlich im README: *„Phase 3 – Steampunk-Erweiterungen KOMPLETT“* [92] legt nahe, dass alle Steampunk-Features fertig im Spiel sind. Zwar sind die Scripts dafür vorhanden [39], doch wurden sie in Level 2/3 gar nicht oder nur teilweise eingesetzt (MovingPlatform, RotatingObstacle etc. fehlen noch in den Level-Layouts) [16] [18]. Diese Diskrepanz zwischen Dokumentation und Realität sollte bereinigt werden, damit das Team und Außenstehende klar erkennen, was tatsächlich schon funktioniert und was noch *Work-in-Progress* ist.

- **Inhaltliche Fehler durch KI-Generierung:** Einige `.md`-Dateien scheinen Halluzinationen oder falsche Annahmen der KI zu enthalten. Ein Beispiel ist die Angabe der **Unity-Version „6.1“** im README [93] – Unity verwendet derzeit ein Jahr-basiertes Versionsschema (z.B. 2021.3), eine Version 6.1 existiert so nicht. Vermutlich sollte das bedeuten „Unity 2021.3 oder höher“. Solche fehlerhaften Details könnten Leser verwirren und sollten korrigiert werden. Ebenso sind manche Klassennamen erwähnt, die zwar im Projekt existieren, aber deren Rolle überzeichnet wird. Z.B. *EGPUPerformanceOptimizer.cs* wird in einem SceneReport als ungenutztes Feature aufgeführt [94] – dieses Skript existiert tatsächlich [95] und optimiert Grafikeinstellungen, ist aber für das Gameplay nebensächlich. Die Dokumentation vermittelt hier den Eindruck, als wäre dies ein großes fehlendes Gameplay-Element, was nicht der Fall ist. Generell sollte man die automatisch erzeugten Texte kritisch prüfen: Enthalten sie Behauptungen, die der Code nicht stützt? Sind alle empfohlenen Schritte realistisch?

- **Redundanz und Versionsstände:** Es liegen manche Informationen redundant in verschiedenen Dateien vor. Beispielsweise finden sich Levelanforderungen sowohl im README (Abschnitt **Level-Schwierigkeiten**) [96] als auch in den SceneReports [97]. Dabei weichen die Formulierungen leicht voneinander ab. Solche Doppelungen können inkonsistent werden. Es ist ratsam, die Dokumentation zu konsolidieren – z.B. eine zentrale Stelle für Spielmechanik-Design (README) und separate Stellen für den tatsächlichen Reparaturfortschritt (SceneReports). Zudem gibt es Anzeichen, dass einige MD-Dateien veraltet sind (Backup-Versionen im Repo zeigen alte Stände). Man sollte aufräumen und nur die aktuellen, gültigen Dokumente behalten, um Verwirrung zu vermeiden.

- **Dokumentation der Bugs und Fixes:** Positiv zu vermerken ist, dass die SceneReports und Repair-Checklisten sehr detailliert die gefundenen Probleme und vorgesehenen Lösungen beschreiben (in deutscher Sprache, was für das Team gut verständlich ist). Diese dienen intern als hervorragende **To-Do-Liste** – z.B. die **Reparatur-Checkliste** [98] [99] listet konkret auf, was in

jeder Szene zu tun ist. Diese Auflistungen stimmen auch mit den oben analysierten Befunden überein (etwa „Level2: Collectibles von 5 auf 8 erhöhen, Hindernisse hinzufügen, Steampunk-Materialien anwenden“ steht in der Checkliste [26] und deckt sich mit unseren Empfehlungen). Es ist wichtig, diese Punkte tatsächlich abzuarbeiten und anschließend die Dokumentation zu aktualisieren (abhaken), um den Fortschritt festzuhalten. Der *CURRENT_REPAIR_STATUS.md* erfüllt das bereits ansatzweise mit Prozentbalken für Fortschritt [100]. Hier sollte kontinuierlich gepflegt werden, welche Schritte abgeschlossen sind, um stets ein **aktuelles Lagebild** zu haben.

- **Kommunikation für weitere Entwicklung:** Für anstehende Entwicklungsphasen sollte klar definiert sein, welche Features als nächstes angegangen werden. Die Doku liefert Ansätze (Prioritätenlisten, z.B. „Heute: Steampunk-Transformation für Level2+3, OSM-Integration stabilisieren“ [101]). Diese Prioritäten sind sinnvoll und sollten in der Teamkommunikation hervorgehoben werden. Das vorliegende Dokument (Gesamtanalyse) kann dazu dienen, die wichtigsten Baustellen zusammenzufassen und somit die **Entscheidungsgrundlage für die nächsten Schritte** zu bilden.

- **Abschließende Dokumentation:** Sobald die Entwicklungsarbeiten fortgeschritten sind, sollte die Dokumentation bereinigt und für eine Veröffentlichung vorbereitet werden. Dazu gehört:

- Entfernen von rein internen Berichten oder diese ins Entwickler-Wiki auszulagern (z.B. die sehr technischen SceneReports müssen nicht alle in der veröffentlichten README auftauchen).
- Erstellen einer **Benutzer-Dokumentation** bzw. Anleitung: Wie spielt man das Spiel, welche Modi gibt es, bekannte Einschränkungen. Teile des README erfüllen das bereits (Steuerung, Installation) [102] [103], andere Teile sind eher Entwickler-relevant.
- Korrektur offensichtlicher Fehler (wie das Unity-Versionslabel, falsche Fertig-Meldungen) und Übersetzung ins Englische, falls ein internationales Publikum angesprochen werden soll (aktuell ist vieles auf Deutsch, was für deutsche Entwickler okay ist, für internationales GitHub-Publikum aber ggf. übersetzt werden sollte).

Zusammenfassend ist die Dokumentation als **lebendes Werkzeug** zu verstehen, das den Entwicklungsprozess begleitet. Sie hat geholfen, strukturiert Fehler zu finden und Lösungswege vorzuschlagen. Nun muss sie Schritt halten mit den tatsächlichen Codeänderungen. Durch eine sorgfältige Überarbeitung stellt man sicher, dass **kein Wissensdefizit zwischen „Soll“ und „Ist“** bleibt: jeder im Team (und externe Contributor) weiß genau, was der aktuelle Stand ist und was als Nächstes ansteht. Dies erhöht die Effizienz und verhindert Missverständnisse während der weiteren Entwicklung.

Fazit

Die Gesamtanalyse zeigt, dass Roll-a-Ball: Steampunk Collector ein **solides Fundament** besitzt – die Kernmechaniken funktionieren, und viele fortgeschrittene Features sind bereits angelegt. Allerdings gibt es in mehreren Bereichen **Ungleichgewichte zwischen Planung und Umsetzung**. Die Tutorial-Level erfüllen noch nicht komplett die an sie gestellten Anforderungen in Sachen Schwierigkeit und Steampunk-Feeling, die **Levelprogression bricht nach Level 3** mangels automatischer Weiterleitung, und der ambitionierte **OSM-Modus** benötigt erhebliche Nacharbeit, um vom Tech-Demo-Stadium zur spielbaren Erfahrung zu gelangen.

Durch die oben vorgeschlagenen **strukturellen, funktionalen und architektonischen Verbesserungen** – insbesondere: Vereinheitlichung der Level-Struktur (Prefab-Nutzung, vollständige Manager-/UI-Präsenz), Ausschöpfung der Steampunk-Elemente in allen Leveln, verlässliche Übergänge und Schwierigkeitssteuerung, sowie intelligente Aufbereitung der OpenStreetMap-Daten für Gameplay – kann das Projekt auf den richtigen Kurs gebracht werden. Wichtig ist, diese Änderungen **schrittweise**

und getestet vorzunehmen, wobei die umfangreiche Dokumentation als Leitfaden dient. Schließlich soll eine Version erreicht werden, die **stabil läuft (keine Errors, flüssige Performance)** [104], inhaltlich überzeugt (anspruchsvolle Level, interessantes Endlosspiel) und das Steampunk-Stadt-Thema sowohl in den handgebauten als auch den generierten Leveln konsistent rüberbringt. Mit fokussierter Weiterarbeit entlang der identifizierten Baustellen steht einer erfolgreichen nächsten Phase nichts im Wege – die Grundlage ist gelegt, nun gilt es, daraus ein rundes Spielerlebnis zu formen. [105] [23]

---

[1] [8] [23] [26] [33] [34] [98] [99] [100] [101] [104] [105] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/wiki/docs/development/CURRENT_REPAIR_STATUS.md

[2] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/ProgressionManager.cs

[3] [9] [16] [97] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/wiki/docs/development/SceneReport_Level2.md

[4] [12] [17] [18] [19] [20] [81] [82] [94] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/wiki/docs/development/SceneReport_Level3.md

[5] [39] [40] [92] [93] [96] [102] [103] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/README.md

[6] [7] [10] [11] [35] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/LevelManager.cs

[13] [14] [15] [27] [36] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/Generators/LevelProfileCreator.cs

[21] [22] [25] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/AGENTS.md

[24] [28] [29] [30] [31] [32] [37] [38] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Reports/SceneReport_GeneratedLevel.md

[41] [87] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/OSM_Debug_Report.txt

[42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [55] [56] [57] [66] [69] [70] [83] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/wiki/docs/development/SceneReport_Level_OSM.md

[54] Overpass API - OpenStreetMap Wiki
https://wiki.openstreetmap.org/wiki/Overpass_API

[58] [64] [65] [88] [89] [91] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/Map/PHASE_4_OSM_SETUP_GUIDE.md

[59] [60] [61] [67] [68] GitHub
https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/Map/AddressResolver.cs

62 63 GitHub

https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/Map/OSMUIConnector.cs

71 72 73 74 75 76 77 78 79 80 84 85 86 90 GitHub

https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/Map/MapGenerator.cs

95 GitHub

https://github.com/161sam/Roll-a-Ball/blob/fb501f6fb2e2d537efd359399267df9381db3e8c/Assets/Scripts/EGPUPerformanceOptimizer.cs