HOCHSCHULE
DER MEDIEN

Projektdokumentation

Software Projekt
Nr. 4209

Gerda

Erstellt im SS 2023

von

Janno Jens
(jj047@hdm-stuttgart.de)

Lea Junker
(lj042@hdm-stuttgart.de)

Henry Ruß
(hr044@hdm-stuttgart.de)

Betreuer: Prof. Dr. Stefan Radicke

Wir bestätigen hiermit, dass wir die Ausarbeitung selbständig erarbeitet haben und detaillierte Kenntnis vom gesamten Inhalt besitzen.

Janno Jens *Lea Junker* *Henry Ruß*

Inhaltsverzeichnis

1 Übersicht

1.1 Links

- Github Repository für das Spiel: https://github.com/Three-s-A-Crowd-Games/Gerda
- Github Repository für die (modifizierte) Engine: https://github.com/Three-s-A-Crowd-Games/godot

1.2 Projektbeschreibung

Gerda ist ein Action-Roguelike top-down Shooter mit Mining-Twist. Man kämpft sich als Untergrundtier durch verschiedene Biome, upgradet seine Waffen und gräbt sich durch die Welt.

1.3 Engine

Gerda wurde in der Godot4.0-Engine umgesetzt.
Diese wurde wegen ihrer Portabilität, Kompatibilität, Einfachheit und Geschwindigkeit gewählt.
Weiterer Grund zur Wahl der Godot-Engine war, dass die Engine Open-Source ist und so Modifikationen erlaubt (siehe 4.3).

2 Visuals

Alle Grafiken wurden mit dem Pixel Art Tool Aseprite erstellt.

2.1 UI

Um das Programm so modular wie möglich zu gestalten, sind die einzelnen Komponenten wie Buttons oder Label in horizontalen bzw. vertikalen Boxen gelagert, welche an eine bestimmte Stelle des Bildschirms verankert sind (siehe Abbildung 1). Dadurch passt sich das Interface auf verschiedene Bildschirmgrößen an. Um die Anzeigen auf dem HUD mit den entsprechenden Funktionalitäten zu verbinden werden Signale verwendet, die bei entsprechenden Events mit den nötigen Informationen gesendet werden.

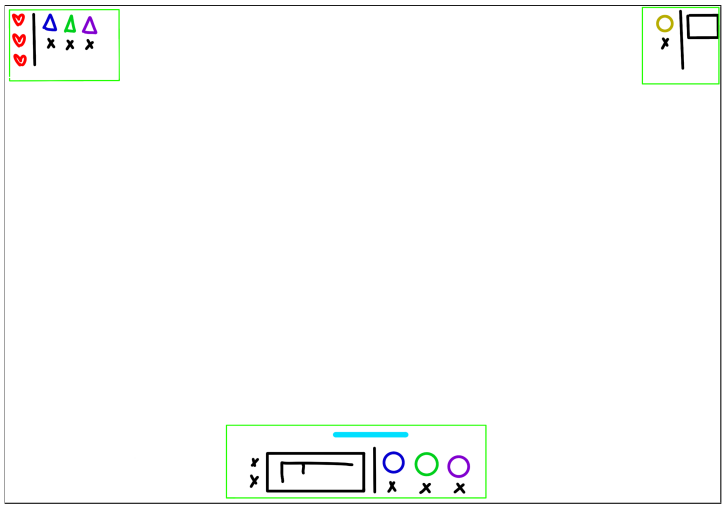

Abbildung 1: Beispiel für ein Interface. Die grünen Boxen sind jeweils am Bildschirmrand verankert, mit extra Rand.

3 Architektur

3.1 Komponenten-System

Ein grundlegendes Architektur-Muster von Gerda ist das Component-Pattern.
Funktionalitäten wie das Minen, Flashen oder Dashen sind (im Rahmen der Limitationen) als gekapselte Komponente implementiert, welche dann einem Akteur hinzugefügt werden können. In Verbindung mit Vererbung konnten wir so einen hohen Grad an Modularität erreichen.

3.2 Scene-Tree Aufbau

Beim Start des Spiels werden alle Singleton Nodes in die Scene-Tree root eingefügt und zusätzlich unsere Main Node, die später das gesamte Spiel enthält. Zu Beginn sind dort dann das Startmenü-Level und ein Canvaslayer mit dem Startmenü. Dieses Canvaslayer wird dann auch für den Menüverlauf verwendet um die anderen Menüs anzuzeigen. Sobald dann ein Charakter ausgewählt wurde, wird das Startmenü-Level durch die `GameWorld` ersetzt. In dieser wird dann der Spieler, das HUD und andere in-game Menüs, sowie das eigentliche Level geladen. Dieser Zustand wird dann auch beibehalten, bis der Bossraum erreicht wird. In dem Moment wird dann nur das Level, das das momentanene Biom enthält, mit dem Level für den Bossraum ausgetauscht. So kann einfach die Umgebung getauscht werden, ohne den Zustand des Spielers zu verlieren. Wenn dann das Spiel verlassen wird und man zurück zum Hauptmenü kommt wird die `Gameworld` wieder gelöscht, sodass man wieder im Anfangszustand ist.

3.3 Singletons

Singletons, bzw. Autoload-Nodes in Godot, ermöglichen eine szenenübergreifende Datenverwaltung und Spielsteuerung. Sie werden neben den normalen Scene-Tree geladen und sind von überall aus erreichbar, vergleichbar mit statischen Objekten.
Gerda setzt aktuell auf 3 Singletons, um die Datenverwaltung in Aufgabenbereiche zu unterteilen.

1. EnemyCreator
2. MutatorManager
3. StructureRegistry

Der EnemyCreator stellt die Enemy-Spawn-Mechanik zu Verfügung. Dies beinhaltet das Verwalten der Spawn-Chancen aber auch das Ermitteln von möglichen Spawn-Positionen. Für letzteres sind unkonstante Variablen nötig, welche in einem statischen Kontext in Godot4 nicht existieren.
Der MutatorManager verwaltet, welche Mutatoren gerade aktiv sind und stellt Informationen über diese zu Verfügung. Er dient als generelle Schnittstelle der Mutator.
In der StructureRegistry werden die verschiedenen Räume der Biome hinterlegt und vorbereitet. So können anderen Klassen auf diese dann zugreifen.

4 Game-Welt

4.1 TileMap-Setup

4.1.1 Generelle TileMap

Die Spielwelt in Gerda ist eine TileMap. Diese ermöglicht sehr einfach das Mining-System umzusetzen, aber hilft durch ihre Schichten auch bei der visuellen Anordnung der Welt. Gerda verwendet 3 Schichten:

1. Ground-Layer
2. Block-Layer (Y-Sort enabled)
3. Mining-Overlay-Layer (Y-Sort enabled)

Die TileMap wird mit Tiles aus biom-spezifischen TileSets gefüllt, deren TileAtlasse ebenfalls auf diese Schichten aufgeteilt sind. Es gibt zusätzlich ein NOPE-TileAtlas, welcher in 4.2 wichtig wird.

4.1.2 Tile-Daten

Jedem Tile aus einem Atlas können Daten zugewiesen werden, welche angeben, wie andere Akteure mit diesen Tiles interagieren. Diese werden in Gerda wie folgt genutzt:
Der **Block-Atlas** definiert für Wände und Oberseiten von Blöcken zwei Kollisionsschichten. Eine für Kollision mit Entities und eine für Line-of-Sight-Checks und das Mining. Es gibt außerdem für jede Oberseite ein alternatives Tile zur Verwendung als obere Wand, welches eine angepasste Kollisionsschicht hat, damit Entities etwas hinter der Oberseite verschwinden können. Zusätzlich wird Licht-Verdeckung definiert und die zwei Custom-Data-Attribute für das Mining-System definiert: die Härte des Steins und ob es ein Ore ist oder nicht.
Der **Ground-Atlas** setzt auf insgesamt drei Varianten pro Untergrund, welche sich pur in ihrem Navigations-Overlay unterscheiden. So gibt es ein voll navigierbares Tile für normalen Boden, ein Tile das nur im oberen Bereich navigierbar ist, für Boden der von oberen Wänden verdeckt wird und ein Tile ohne Navigation zur Verwendung unter normalen Wänden, da hier Boden nur visuell vorhanden sein soll.

4.2 Prozedurale Generation

Jede Welt in Gerda wird prozedural generiert, wobei ein Ablauf von verschiedenen Algorithmen verwendet wird.
Hierbei können Variablen in der Klasse `God` angepasst werden, um die Größe der Map, Menge an Dungeons etc. anzupassen.

4.2.1 Vorbereitung

Vor der eigentlichen Erstellung der Welt und deren Heightmaps werden zwei wichtige Arrays/Dictionaries erstellt:
Einerseits ein Array mit über Poisson-Disk-Sampling (PDS) generierten Punkten für die zufällig verteilten Räume. PDS sorgt hier für eine zufällige Verteilung, bei gleichzeitiger Beachtung von Mindestabständen zwischen Räumen. Aus diesem Array wird direkt ein Punkt für den Falltür-Raum (das Ziel für jedes Biom) ausgewählt.
Außerdem ein Dictionary mit Punkten welche von einem Algorithmus über inverse distance weighting generiert wurden. Diese Punkte werden in Richtung des Falltürraums immer dichter. Hier wird ein Dictionary verwendet, da dieses für seine Keys effektiv ein Set verwendet[1], wodurch Lookup-Funktionen wesentlich effizienter sind als in Arrays.

4.2.2 Generelle Welt

Als erstes wird eine Boundary erstellt, welche den definierten Bereich einschließt.
Um das Höhlensystem zu berechnen, wird länger-welliges PerlinNoise verwendet. Basierend

[1] Reddit Beitrag zu Sets in Godot

auf dieser Heightmap werden dann Ground und Block-Tiles platziert.
Beim setzen der Blöcke wird einerseits beachtet, ob der Block eine Wand (egal ob eine untere oder obere) oder sogar ein Erz sein muss. Um letzteres zu determinieren wird eine zweite PerlinNoise-Heightmap befragt, welche mit einer höheren Frequenz erstellt wurde.
Immer wenn Ground gesetzt werden muss, wird eine zentrale Funktion befragt, so auch z.B. beim Minen. Diese Funktion befragt das in 4.2.1 angesprochene Dictionary um spezielle Ground-Tiles an den Punkten zu platzieren.

4.2.3 Raum-Platzierung

Nach der Welt werden die verschiedenen Räume in der Welt platziert.
Generell werden für jedes Biom eigene Räume in eigenen Szenen aufgebaut, welche dann in der `StructureRegistry` registriert werden. So kann für jede gewollte Struktur ein TileMap-Pattern extrahiert werden und die in den Szenen platzierten Interactables, Dekorationen und Gegner in die eigentliche Spielwelt übertragen werden.
Um die TileMap-Patterns auch korrekt in die Spielwelt zu übertragen, werden Nope-Tiles verwendet, um zu markieren, welche Tiles aus der vorgenerierten Welt entfernt werden sollen. Ist das Pattern platziert, wird eine Cleanup-Funktion gestartet, welche die Nope-Tiles entfernt und das Pattern in die Welt angleicht, also wenn nötig die Wände des Patterns korrigiert oder die Wände der Spielwelt berichtigt, wenn nur ausgehöhlt wurde.
Die Reihenfolge der Platzierung:

1. Falltürraum (Position wurde bereits in 4.2.1 bestimmt)
2. Anvil-Räume (hiervon eine definierte Anzahl in einem Radius in den Ecken der Welt, eine andere definierte Anzahl an den generierten Random-Punkten in der Welt)
3. Dungeons (eine leicht Variable ($x \pm y$) Anzahl an den generierten Random-Punkten in der Welt)
4. Spawn-Raum (an dem letzten verbleibenden generierten Random-Punkt)

4.3 Engine-Optimierungen

Unser Einsatz von einer großen TileMap mit Navigation und YSort hat Godot4 an Performance-Grenzen gebracht.
Da wir nicht die einzigen mit diesen Problemen sind, konnten wir Issues und Fixes finden, die jedoch entweder noch gar nicht oder erst in Godot4.1 eingebaut wurden.

4.3.1 Y-Sort

Die Verwendung von Y-Sort auf einer großen TileMap hat wegen ineffizienten Berechnungen die FPS deutlich gesenkt, wie auch in diesem Issue beschrieben.
Der Fix aus diesem PR kann in manchen Projekten zwar Probleme bereiten, Gerda ist jedoch nicht betroffen.
Dieser Fix ist offiziell erst für Godot4.2 angesetzt, da er noch überarbeitet werden soll.

4.3.2 Navigation

In der Godot4-Engine wird standardmäßig Edge-Connection zum Verbinden von Navigations-Bereichen einzelner Tiles verwendet.
Da diese Edge-Connections vergleichsweise ineffizent berechnet werden, sorgte dies für extreme Lag-Spikes (300ms Berechnungen) beim Abbauen von Blöcken.
Seit Godot4.1 existiert dank diesem PR eine Option, diese Edge-Connections aus zu schalten, wodurch Connections nur noch durch `edgeKey` berechnet werden, was wesentlich schneller geht.

5 Gegner und deren Verhalten

Im momentanen Zustand gibt es 2 verschiedene Typen an Gegnern mit jeweils einem Vertreter. Zum einen ist das der Wurm als stationärer Gegner 5.1 und die Spinne als bewegender Gegner 5.2. Beide Typen sind jeweils als Basisklasse mit der entsprechend notwendigen Funktionalität implementiert und werden für die spezifischen Gegner dann erweitert.

5.1 Stationärer Gegner

Die Idee hinter einem Stationären Gegner ist offensichtlicher Weise zum einen, dass er sich nicht bewegt. Zum anderen ist es die Eigenschaft dass er einen bestimmten Aktivierungsbereich hat, der den Aktivitätsstatus des Gegners regelt. Wenn der Spieler diesen Bereich betritt wird der Gegner aktiviert und kann dann sein jeweiliges Verhalten ausführen. Dementsprechend wird er auch wieder deaktiviert wenn der Spieler den Bereich verlässt.

5.1.1 Wurm

Der Wurm erweitert das System des stationären Gegners um einen Timer, dessen Ende eine Attacke initiiert. Die Attacke des Wurms ist dann ein parabolisch fliegendes Giftprojektil, das er aus seinem Mund schießt. Außerdem wird seine Hurtbox deaktiviert sobald er sich selbst deaktiviert, was zur Folge hat, dass er in seinem idle-Status keinen Schaden nehmen kann.

5.2 Bewegender Gegner

Um bei den bewegenden Gegnern das genaue Verhalten möglichst anschaulich und modular implementieren zu können, haben wir uns dazu entschieden, einen **Behaviour-Tree** zu verwenden. Dafür benutzen wir Adrien Quillets Yet Another Behaviour Tree Plugin, das unter der Apache License in der Godot AssetLib veröffentlicht wurde. Dies ermöglicht es uns, einen Behaviour-Tree als Szene mit den entsprechenden Nodes zu bauen und einfach anzupassen. In jedem Physics-Process-Frame wird der Behaviour tree durchlaufen um die nächste Aktion des Gegners zu bestimmen.
Zum Pathfinding wird ein Navigationsmesh verwendet, das durch Navigationsregionen auf den Boden-Tiles zusammengebaut wird. Zur Berechnung der Pfade wird dann jeweils der in Godot eingebaute A*-Algorithmus verwendet. Dies hat zur Folge dass die Pfade immer optimal sind. Eine Konsequenz daraus ist, dass mehrere Gegner gleichzeitig auf dem gleichen Pfad laufen wollen und sich gegenseitig den Platz wegnehmen bzw. einfach den selben Platz einnehmen und sich „stacken“. Um das zu vermeiden, verwenden die Gegner die von Godot zur Verfügung gestellte RVO Object Avoidance. Dadurch haben die Gegner ein „Bewusstsein“ ihrer näheren Umgebung und können ihren Pfad anpassen, damit sie neben anderen laufen.

5.2.1 Spinne

Die Spinne, als bisher einziger Vertreter der bewegenden Gegner, hat ein relativ normales Laufverhalten. Dieses funktioniert wie folgt:
Generell gibt es zwei verschiedene Modi in denen eine Spinne sein kann. Einen **Verfolgungsmodus** und einen **Wandermodus**. Wenn der Spieler für die Spinne sichtbar ist, befindet sich die Spinne im Verfolgungsmodus und verfolgt diesen. Wenn der Sichtkontakt abbricht wird er für einen kurzen Zeitraum weiterverfolgt um eine etwas höhere „Intelligenz“ der Spinne zu simulieren, da sie sonst direkt in den Wandermodus übergehen würde. In diesem Modus sucht sich die Spinne eine beliebige Position in ihrem Umkreis und macht sich dann auf den Weg dort hin. Wenn die Spinne diese Position erreicht hat oder sie eine vorher definierte Maximaldistanz abgelaufen hat, bleibt sie für einen kurzen Moment stehen und sucht sich dann eine neue Position. In jedem Fall wird vom Wander- wieder in den Verfolgungsmodus gewechselt sobald die Spinne Sichtkontakt zum Spieler hat.

5.3 Boss Gegner

Am Ende jedes Bioms ist ein Bossgegner anzutreffen den man besiegen muss, um das nächste Biom zu betreten bzw. das Spiel zu beenden. Diese Bosse erhalten ein eigens für sie kreiertes Level, welches einen Raum enthält, in dem dann der Kampf stattfindet.

5.3.1 Spinnen-Boss

Am Ende des ersten Bioms kämpft man gegen eine große Spinne. Diese hat einen kleinen Raum, der dem Spieler relativ wenig Platz gibt und immer in der Nähe hält. Die Spinne hat zum Angreifen verschiedene Möglichkeiten. Da die Wahl des Angriffs von mehreren Faktoren, wie der Position des Spielers oder dem Stand verschiedener Timer abhängt, haben wir uns dazu entschieden das Verhalten mit einer **State-Machine** zu modellieren und umzusetzen. Die State-Machine als Teil der AnimationTree-Node in Godot bestimmt dabei für jeden State die mögliche Nachfolge-States und wie ein Übergang stattfindet. Die tatsächlichen Implementationen der States sind dann als Animationen in einer Animationplayer-Node implementiert, welche verschiedene Eigenschaften (Properties) der Spinne animieren bzw. verändern. Um einen Wechsel des aktuellen States zu initiieren wird dann im Code für die Spinne bei einem Event die entsprechende Übergangsbedingung in der State-Machine aktiviert. So wird zum Beispiel das Schwingen der Vorderbeine initiiert, wenn der Spieler den Bereich vor der Spinne betritt. Die State-Machine ermöglicht es auch, verschiedene Übergangsbedingungen zu priorisieren, um im Fall, dass mehrere zutreffen, eine Regel für die Auswahl befolgt werden kann.

6 Momentaner Zustand

6.1 Crashes

Wir sind uns Bewusst, dass im momentanen Zustand immer wieder Crashes vorkommen wenn man das Spiel mit der .exe Datei startet. Wir konnten die genauen Gründe bis jetzt leider nicht ausmachen, da keinerlei Fehlermeldungen geworfen werden. Wir haben bisher auch noch keinen Weg gefunden die Crashes konstant zu reproduzieren. In den meisten Fällen kommen die Abstürze, aber nur am Anfang vor und passieren dann nicht mehr.