

Studienprojekt
**Entwicklung eines Farming Simulators auf
einer zeitgemäßen Game Engine**

im Studiengang Softwaretechnik und Medieninformatik
der Fakultät Informationstechnik
Wintersemester 2023/24

Pavithra Sureshkumar

Zeitraum: 14.11.2023 - 15.02.2024
Prüfer: Prof. Dr.-Ing. Harald Melcher
Zweitprüfer: Prof. Dr.-Ing. Andreas Rößler

Inhaltsverzeichnis

1	Einleitung	1
1.1	Kurze-Zusammenfassung	1
1.2	Motivation/Einblick	1
2	Grundlagen der Godot Game Engine	2
2.1	Einführung in Godot	2
2.1.1	Vorteile und Besonderheiten	2
2.1.2	Erste Schritte	3
2.2	Die Oberfläche von Godot	5
2.2.1	Der Projekt Manager	5
2.2.2	Die Panelübersicht	6
2.3	Szenen und Nodes	8
2.3.1	Szenen	8
2.3.2	Nodes	8
2.3.3	Hierarchie von Nodes	9
2.4	GDScript	11
2.4.1	Was ist GDScript?	11
3	Realisierung des Farming Simulators	13
3.1	Konzeptualisierung des Farming Simulators	13
3.1.1	Spielidee und Motivation	14
3.1.2	Gestaltung des Hauptcharakters	16
3.1.3	Gestaltung des Inventarsystems	18
3.1.4	Aufbau der Datenstruktur des Inventarsystems	19
3.2	Praktische Umsetzung des Farming Simulators	21
3.2.1	Nomad Sculpt	21
3.2.2	Realisierung der Modelle mit Nomad Sculpt	23
3.2.3	Mixamo von Adobe und Blender	31
3.2.4	Blender und Godot	35
3.2.5	Umsetzung der Charakteranimationen: "Idle" und "Walk" in Godot	36
3.2.6	Umsetzung des Inventarsystems: Das erste Prototyp	43
4	Ergebnis und Ausblick	74
	Literatur	76

Abbildungsverzeichnis

2.1	Godot Logo	2
2.2	Godotengine macOS download Seite	3
2.3	Godot Projekt Manager	5
2.4	Godot Panelübersicht	6
2.5	Godot Scene	8
2.6	Godot Nodes Hierarchie	9
2.7	GDScript Code Beispiel Sprite2D	11
3.1	Mindmap: Gedanken zum Spiel	14
3.2	Animal Crossing Body	16
3.3	Charakter Skizze	17
3.4	Minecraft Inventarsystem	18
3.5	Einfaches Inventory System	18
3.6	Skizze von Datenstruktur des Inventarsystems	19
3.7	Nomad Logo	21
3.8	Interface Nomdad Sculpt	21
3.9	Erster Prototyp Charakter mit Nomad Sculpt	23
3.10	Zweiter Prototyp Charakter mit Nomad Sculpt	24
3.11	Charakter ohne richtigen Torso	25
3.12	Charakter mit richtigen Torso	26
3.13	Charakter ohne Paint	27
3.14	Charakter mit Paint	28
3.15	Charakter UV-Unwrap UV-Atlas	29
3.16	Haus Design	30
3.17	Mixamo Startpage	31
3.18	Hochladen des 3D Charakters	31
3.19	Animation Library von Mixamo	32
3.20	Mixamo exportieren der Animation	33
3.21	Mixamo und Blender Actions	34
3.22	Importieren des 3D-Modells und Animationsplayer in Godot	35
3.23	Godot Asset Library: Basic 3D Starter	36
3.24	Godot Project Settings	37
3.25	Godot Script	38
3.26	Godot AnimationTree	41
3.27	Godot AnimationTree Editor	41
3.28	Godot Slot UI	43

3.29 Godot Populate Slot Data	45
3.30 Charakter Inventarsystem: Inventar Daten, Slot Daten	49
3.31 Hotbar Inventar	56
3.32 Slot Data Beispiel: Quantität	61
3.33 PickUp Items	63
3.34 Item Data Atlas Texture	65
3.35 Item Data: Yellow Book	66
3.36 Charakter mit RayCast3D	71
4.1 Farming Simulator	74

Listings

3.1	Character movement	39
3.2	inventory.gd	44
3.3	inventory_data.gd	46
3.4	inventory_interface.gd	50
3.5	hotbar_inventory.gd	54
3.6	slot.gd	57
3.7	slot_data.gd	59
3.8	pickup_item.gd	62
3.9	item_data.gd	64
3.10	item_data_consumable.gd	67
3.11	character.gd	68
3.12	character_manager.gd	72
3.13	cute_mini_figure.gd	73

Kapitel 1

Einleitung

Die vorliegende Arbeit befasst sich mit der Entwicklung eines 3D Farming Simulators. Das Spiel wird unter Verwendung einer zeitgemäßen Game Engine (Godot) erstellt und die 3D-Assets mit NomadSculpt, Blender und Mixamo erstellt.

1.1 Kurze-Zusammenfassung

Das Ziel dieses Projektes ist die Entwicklung eines Farming Games unter Verwendung der Game Engine Godot v4.2.1.stable.official.

Die vorliegende Arbeit dokumentiert den gesamten Prozess, unter anderem der Planung, Erstellung und Bewältigung von Herausforderungen im Zusammenhang mit dem Projekt. Dabei sind die wichtigen Aspekte im Fokus: die Planung, die Umsetzung mit der Game Engine Godot und 3D Blender sowie die Erstellung des Spiels.

1.2 Motivation/Einblick

Das Projekt wird durchgeführt, um die heute zeitgemäße Game Engine Godot kennenzulernen. Das Farming Game soll zum Entspannen dienen und einem das Gefühl geben, in einer anderen Welt einzutauchen.

Kapitel 2

Grundlagen der Godot Game Engine

Um ein besseres Verständnis über der Godot Game Engine zu erlangen, muss man ein paar Grundlagen wissen. Dieses Kapitel gibt eine kleine Einführung zu Godot und erklärt grob wie die Game Engine zu bedienen ist und was man alles mit ihr erreichen kann.

2.1 Einführung in Godot

Godot ist eine freie open-source Game Engine und wurde von Juan Lienetsky und Ariel Manzur in 2007 entwickelt. Im Jahr 2014 wurde die Game Engine unter der MIT-Lizenz veröffentlicht und kann von der öffentlichen Seite von Godot heruntergeladen werden ([1](#), S.1).



Abb. 2.1: Godot Logo ([2](#))

2.1.1 Vorteile und Besonderheiten

Die Godot Game Engine ist eine Open-Source-Plattform für die Entwicklung von Computerspielen. Sie bietet viele Vorteile, wie die Erstellung von 2D-Plattformspielen und 3D-Plattformspielen. Außerdem ist die Benutzeroberfläche der Game-Engine sehr benutzerfreundlich gestaltet.

Alles, was mit Godot erschaffen wird, gehört zu 100 Prozent den Entwicklern, im Gegensatz zu anderen kommerziellen Game-Engines. Bei anderen Game Engines ist in

der Regel eine vertragliche Zusammenarbeit erforderlich. Der Entwickler kann selbst entscheiden, wie und wo das Spiel vertrieben wird. Viele kommerzielle Game Engines haben strikte Voraussetzungen für das Veröffentlichen von aufwendigen Spielen und erfordern den Kauf einer Lizenz (3, S.4).

Godot ist auch sehr transparent, denn die Entwickler können selbst entscheiden, ob sie eine bestimmte Eigenschaft der Engine modifizieren oder neue Eigenschaften implementieren möchten, ohne eine spezielle Genehmigung einholen zu müssen. Diese Eigenschaft ist besonders hilfreich bei der Entwicklung größerer Projekte, da man vollen Zugriff auf die internen Funktionen der Game Engine hat (3, S.5).

Für viele Entwickler und Anfänger ist Godot daher die bessere Lösung für die Spieleentwicklung. Godot wird nicht von einem Unternehmen entwickelt, sondern ist den Entwicklern gewidmet, die ihre Zeit und Erfahrung investieren, um die Engine zu erstellen, zu testen und Fehler zu beheben. Außerdem führen sie ausführliche Dokumentationen über die Game Engine. Viele Entwickler erstellen auch einige Prototypen und Templates von Starter-Game-Szenen, um mit Godot den ersten Schritt zu machen (3, S.5).

2.1.2 Erste Schritte

Um mit Godot durchzustarten, muss man die neuste Version von der Godot Seite herunterladen <https://godotengine.org/download/macOS/>. Die jetzige Version sollte im Bereich von Godot 4.x.x sein. Auf der Seite wird auch .NET angeboten, diese wird aber nur gebraucht, wenn man mit C# arbeiten möchte. Alternativ kann man Godot auch per Steam, itch.io oder mit dem Paketmanager (Homebrew, Scoop, Snap) der jeweiligen OS installieren (3, S.6).



Abb. 2.2: Godotengine macOS download Seite (2)

Nach dem klassischem download wird dann ein zip Ordner bereitgestellt, diese wird dann entpackt. Alternativ kann man diese auch zu dem Programm Ordner oder dem Applikationen Ordner rüberziehen. Nach dem öffnen der Applikation sieht man den Godot Projekt Manager Fenster (3, S.6).

Die folgenden Kapitel handeln von der Godot Engine, wie sie zu bedienen ist, die einzelnen Kernmerkmale der Godot Engine und die verschiedenen Elemente, um einen groben Überblick zu geben. was die Game Engine zu bieten hat. Die Kapiteln dienen als Basis um mit dem Projekt Farming Simulator durchzustarten.

2.2 Die Oberfläche von Godot

Dieses Kapitel beschreibt den Aufbau der Oberfläche von Godot.

2.2.1 Der Projekt Manager

Nach dem Öffnen von Godot erscheint zunächst der Projekt Manager. Hier kann man entscheiden, ob man ein neues Projekt erstellen möchte oder in der öffentlichen Asset-Bibliothek nach Projekten suchen möchte (3, S.7-8).

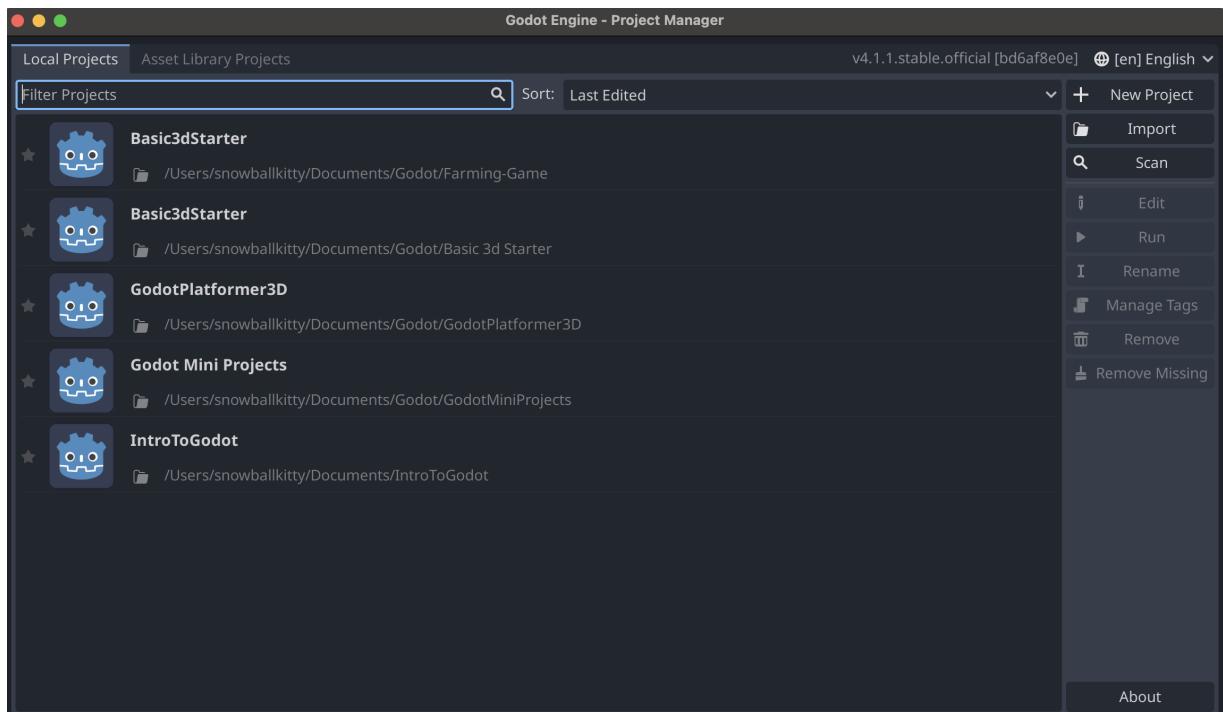


Abb. 2.3: Godot Projekt Manager

2.2.2 Die Panelübersicht

Die Godot-Oberfläche besteht aus mehreren Panels. Die einzelnen Abschnitte wie das Szenenbaum, der FileSystem-Abschnitt, die Arbeitsfläche und der Inspector haben ihre eigene Funktion (3, S.9-11).

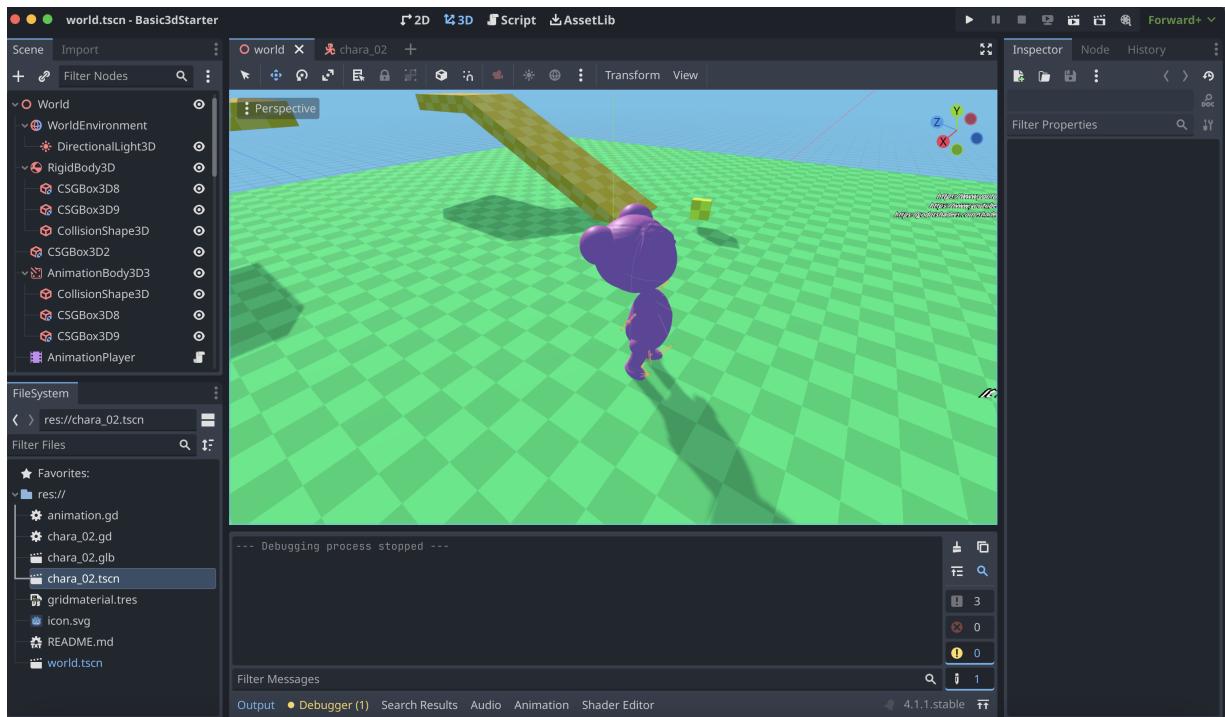


Abb. 2.4: Godot Panelübersicht

In Abbildung 2.4 ist zu erkennen, dass der größte Teil des Editor-Panels das Viewport ist, welches sich in der Mitte befindet. Der Viewport dient der Übersicht über die einzelnen Elemente, die ein Entwickler bearbeitet (3, S.9).

Im oberen Abschnitt der Oberfläche befindet sich die Liste der Arbeitsplätze. Hier kann zwischen 2D-, 3D- und Script-Modus gewechselt werden. Im Script-Modus wird der Spielcode bearbeitet. Der Tab AssetLib ist für das Herunterladen von Add-ons und Beispielprojekten zuständig, die von der Godot-Community bereitgestellt werden (3, S.10).

Über dem Viewport ist die Toolbar sichtbar. Hier können verschiedene Tools verwendet werden, um mit dem Viewport und den einzelnen Elementen zu interagieren (3, S.10).

In der oberen rechten Ecke befindet sich der Player, um das Spiel zu testen. Beim Testen des Spiels erscheint ein neues Fenster, das das Spiel debuggt (3, S.10).

In der linken unteren Ecke des Panels findet man das Dateisystem. Hier werden alle Dateien des Projektordners angezeigt. Alle Ressourcen werden im relativen Pfad res:// lokalisiert, was dem Stammverzeichnis des Projekts entspricht (3, S.10).

In der oberen linken Hälfte ist der Szenenbaum zu sehen, in Abbildung 2.4 auch als Scene bezeichnet. Dieser zeigt die aktuelle Szene an, die im Viewport bearbeitet wird (3, S.11).

Auf der rechten Seite befindet sich der Inspector, in dem die Eigenschaften der Spielobjekte angepasst werden können (3, S.11).

Mit diesen Informationen ist man nun bereit, mit der Godot Engine zu arbeiten. Die Funktionsweise der einzelnen Elemente in Godot lässt sich jedoch nur durch das Bearbeiten von Projekten erfahren. Die folgenden Kapitel geben einen noch tieferen Einblick in die Elemente von Godot, um das Projekt "Farming Simulator" zu starten.

2.3 Szenen und Nodes

Im Kapitel 2.2.2 wurden die einzelnen Elemente der Godot-Oberfläche grob aufgezeigt. Eines der wichtigen Elemente ist der Szenenbaum, welches in diesem Kapitel näher behandelt wird.

2.3.1 Szenen

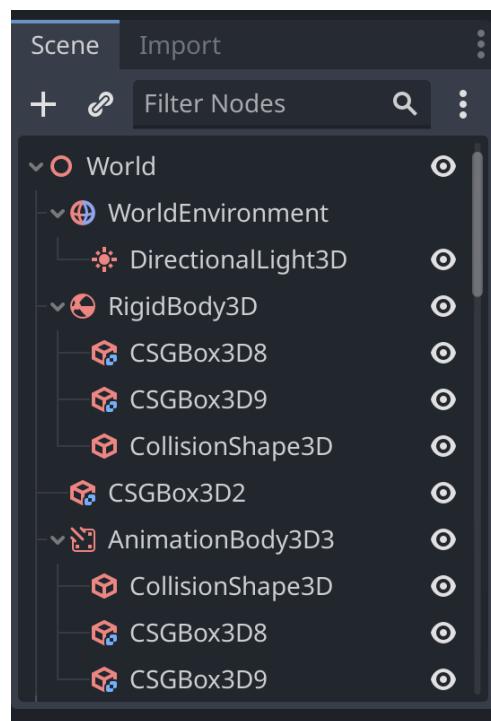


Abb. 2.5: Godot Scene

Eine Szene 2.5 wird in Godot verwendet, um die verschiedenen Spielobjekte im Projekt zu erstellen und zu ordnen. Zum Beispiel kann eine Szene nur für den Spieler erstellt werden, mit den dazugehörigen Nodes und Skripten, um den Spieler zu steuern. Eine andere Szene könnte für die Spielwelt erstellt werden, in der der Charakter mit den erstellten Objekten und Hindernissen in der Welt interagiert. Am Ende werden die Szenen kombiniert, um ein finales Spiel zu erstellen (3, S.12).

2.3.2 Nodes

Nodes sind die Grundbausteine von Godot, um Spiele zu erstellen. Eine Node ist ein Objekt, das verschiedene spezialisierte Spielfunktionen haben kann. Zum Beispiel kann eine Node ein Typ von einer Bildanzeige sein, um eine Animation abzuspielen, oder

ein 3D-Modell darstellen. Eine Node kann eine Sammlung von Eigenschaften besitzen, die dann verwendet werden können, um das Verhalten der Node anzupassen. Welche Nodes ein Entwickler verwendet, ist ihm überlassen, je nach der gewünschten Funktionalität. Es handelt sich um ein modulares System, das dem Entwickler genügend Freiraum und Flexibilität gibt, um die Spielobjekte zu erstellen (3, S.11).

Nodes bringen hauptsächlich ihre eigenen Eigenschaften und Funktionen mit. In Godot kann der Entwickler jedoch das Verhalten oder die Funktionalität der Nodes erweitern, indem er Skripte zu den jeweiligen Nodes hinzufügt. Diese Möglichkeit gestattet es dem Entwickler, mehr aus dem Standard-Node herauszuholen. Zum Beispiel kann man zum Sprite2D-Node eine Eigenschaft hinzufügen, die ein Bild anzeigen soll. Möchte man jedoch, dass das Bild beim Klicken verschwindet und beim erneuten Klicken wieder erscheint, muss man ein Skript einfügen, um dieses Verhalten zu erzeugen (3, S.12).

2.3.3 Hierarchie von Nodes

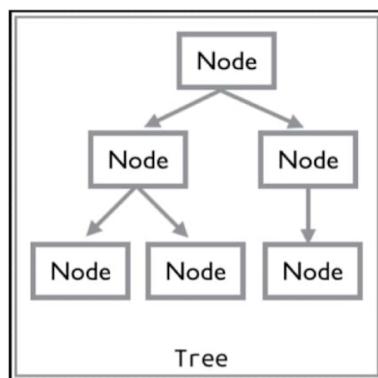


Figure 1.10: Nodes arranged in a tree

Abb. 2.6: Godot Nodes Hierarchie
(3, S.11)

In Abbildung 2.6 werden die Nodes als eine Baumstruktur dargestellt. In einem Baum werden die Nodes als Kinder von anderen Nodes hinzugefügt. Eine bestimmte Node kann viele Kinder haben, aber nur ein Eltern-Node. Wenn eine Gruppe von Nodes zu einem Baum zusammengefügt wird, nennt man dies eine Szene (3, S.11). Wird eine Szene abgespeichert, kann diese als eine neue Node zu einem anderen Node als Kind hinzugefügt werden (4).

Nodes sind ein mächtiges Werkzeug. Das Verständnis dieses Tools ist eine Voraussetzung, um die Objekte in Godot zu begreifen. Doch ohne eine richtige Spiellogik sind Nodes nicht viel wert. Eine Spiellogik legt Regeln fest, die die Objekte befolgen sollen

(3, S.12). Im nächsten Kapitel wird die Skriptsprache GDScript behandelt, die eine der Sprachen ist, die man bei Godot verwenden kann, um eine Spiellogik aufzubauen.

2.4 GDScript

Dieses Kapitel behandelt die verfügbaren Programmiersprachen in Godot, wobei hauptsächlich auf die Skriptsprache GDScript eingegangen wird.

In Godot kann man zwischen zwei Programmiersprachen wählen, um die Nodes zu scripten: GDScript und C#. GDScript ist die dedizierte, integrierte Sprache mit der engsten Integration zur Engine und ist die unkomplizierte Sprache, um Godot zu nutzen. Alternativ kann man auch mit C# programmieren, indem man die entsprechende Version von Godot herunterlädt, die diese Sprache unterstützt. Godot selbst ist in C++ geschrieben, und man kann durch die Verwendung von C++ mehr Leistung und Kontrolle über die Funktionen der Engine direkt erhalten (3, S.12).

Die beste Wahl zum Scripten mit Godot ist GDScript, da es eng mit der Godot - Programmierschnittstelle integriert ist und speziell für schnelle Entwicklung entwickelt wurde (3, S.5).

2.4.1 Was ist GDScript?

Die Syntax der Skriptsprache GDScript ist stark an die Python-Sprache angelehnt. Personen, die bereits Erfahrung mit anderen dynamischen Programmiersprachen wie JavaScript haben, sollten es relativ einfach finden, GDScript zu erlernen. Ähnlich wie Python ist GDScript eine benutzerfreundliche Programmiersprache, besonders geeignet für Anfänger.

```
extends Sprite2D
var speed = 200

func _ready():
    position = Vector2(100, 100)

func _process(delta):
    position.x += speed * delta
```

Abb. 2.7: GDScript Code Beispiel Sprite2D (3, S.13)

GDScript ist eine dynamisch typisierte Skriptsprache, das bedeutet, dass man keine Variablenarten deklarieren muss, wenn man eine Variable erstellt. Stattdessen verwendet GDScript Leerzeichen (Einrückungen), um Codeblöcke zu kennzeichnen. Der größte Vorteil von GDScript liegt in der engen Integration mit der Game Engine, was zu einer schnellen Entwicklung führt und somit wenig Code erforderlich macht.

Die Abbildung 2.7 zeigt grob, wie GDScript aussieht. Der Code repräsentiert ein Sprite2D, das von links nach rechts über den Bildschirm bewegt wird.

Falls weitere Fragen zur GDScript-Sprache auftreten, sollten entsprechende Informationen in der Godot - Dokumentation zu finden sein: <https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/index.html>.

Im weiteren Verlauf des Projekts werden weitere Code-Snippets in GDScript eingeführt.

Kapitel 3

Realisierung des Farming Simulators

Dieses Kapitel befasst sich mit der Umsetzung des Farming Simulators mithilfe der Godot Engine. Dabei werden nicht nur die technischen Aspekte erörtert, sondern auch die Gestaltung des Spiels. Es werden Fragen behandelt, wie beispielsweise: "Warum sollte überhaupt ein Farming Simulator-Spiel entwickelt werden?"

3.1 Konzeptualisierung des Farming Simulators

In diesem Unterkapitel werden hauptsächlich die Spielkonzepte diskutiert. Die technischen Aspekte und die praktische Umsetzung werden im nächsten Kapitel behandelt.

3.1.1 Spielidee und Motivation

Zunächst werden die Gedanken zum Spiel sortiert, dies wurde mithilfe einer Mind Map realisiert 3.1.

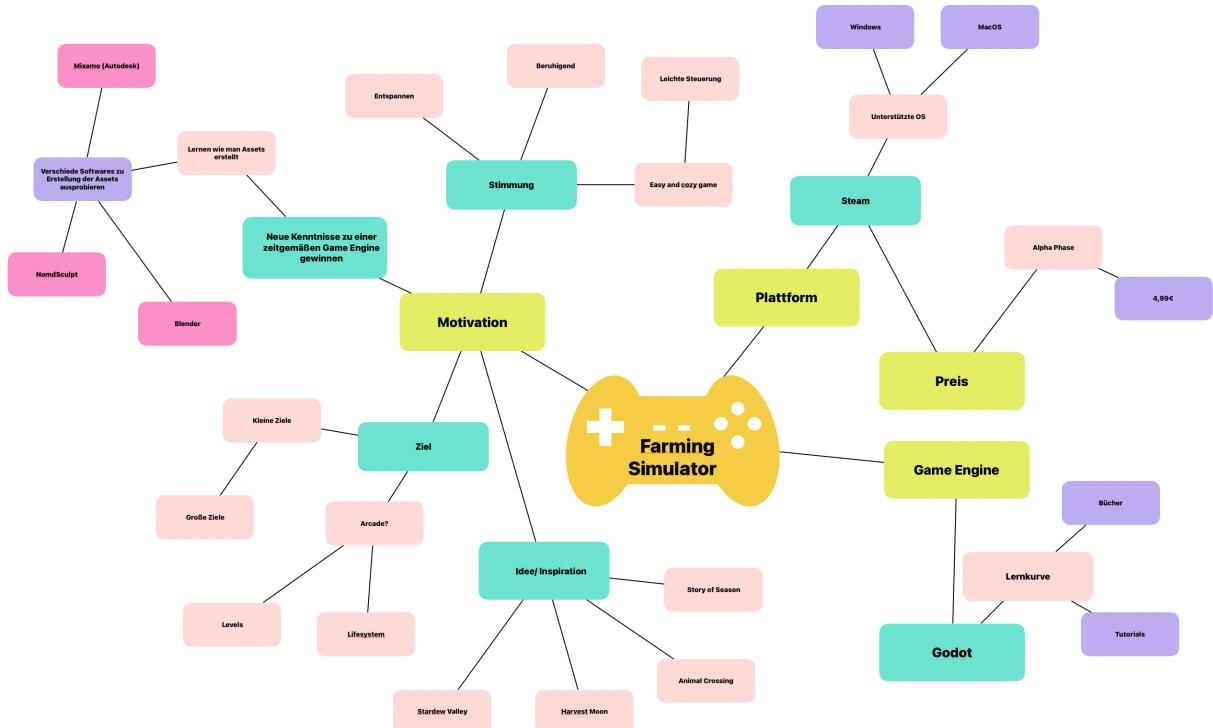


Abb. 3.1: Mindmap: Gedanken zum Spiel

Als erstes stellt sich die Frage: Warum überhaupt ein Farming Simulator Spiel?

Die Motivation hinter diesem Spielkonzept besteht darin, eine bestimmte Stimmung zu erzeugen. Die Idee wurde von bekannten Farming-Simulatoren wie Story of Seasons, Animal Crossing, Stardew Valley und Harvest Moon inspiriert. Bei all diesen Spielen liegt der Fokus auf "Easy Gaming" und "Cozy Gaming". Das Ziel ist es, dem Spieler eine beruhigende Atmosphäre zu vermitteln und ihn auf eine besondere Weise zu entspannen.

Natürlich gibt es auch Entwicklungsziele, die als Motivation dienen. Dazu gehört das Erlernen zeitgemäßer Game Engines sowie das eigenständige Erstellen von Assets unter Verwendung neuer Softwaretools wie Mixamo, NomadSculpt oder Blender. Das übergeordnete Ziel bei der Spielentwicklung ist es, kleine Schritte zu unternehmen und diese schließlich zu großen Fortschritten zu führen.

Im Verlauf des Entwicklungsprozesses stellte sich auch die spannende Frage: "Sollte man ein Arcade-Element integrieren?" Bei diesem Überlegungsschritt wurden Gedanken darüber angestellt, welches Ziel der Spieler im Spiel verfolgt. Ein Spiel besteht oft

aus kleinen Zielen, die der Spieler erreichen möchte. Dazu gehören nicht nur Level oder ein Lebenssystem, sondern auch der Zyklus bestimmter Aufgaben. Im Farming Simulator wird dies beispielsweise durch das Einpflanzen von Samen, das Ernten der Pflanzen und das erneute Erlangen von Samen dargestellt.

Die Auswahl der richtigen Game Engine ist ein entscheidender Aspekt für Entwickler, insbesondere für diejenigen, die wenig Erfahrung haben. Godot wird als ausgezeichnete Wahl für Anfänger empfohlen, insbesondere für die Entwicklung kleiner Indie-Spiele. Dennoch erfordert das Erlernen der verschiedenen Elemente der Game Engine erheblichen Aufwand, und die Lernkurve ist eher schleifend und nicht stetig steigend. Es ist ratsam, die Godot-Dokumentation zu studieren, Tutorials durchzugehen und auch Bücher zu lesen, um sich mit der Plattform vertraut zu machen.

Wenn es darum geht, das Spiel auf einer Plattform zu veröffentlichen, stellt sich die Frage: "Wo soll es hochgeladen werden?" Eine Möglichkeit besteht darin, es auf Plattformen wie Steam zu veröffentlichen. Auch die Auswahl des Betriebssystems, auf dem das Spiel laufen soll, ist entscheidend. Die Frage nach dem Verkaufspreis ist ebenfalls wichtig. In der aktuellen Alpha-Phase könnte eine angemessene Preisgestaltung bei etwa 4,99 Euro liegen.

All diese Überlegungen dienen dazu, einen groben Überblick über das Spiel zu erhalten und das angestrebte Ziel zu erreichen.

3.1.2 Gestaltung des Hauptcharakters

In diesem Kapitel wird die Gestaltung des Charakters behandelt.

Wie bereits im vorherigen Kapitel 3.1 besprochen, wurde das Farmingspiel von Animal Crossing inspiriert.



Abb. 3.2: Animal Crossing Body (5)

Dabei wurde die Figur 3.2 von Animal Crossing genauer betrachtet. Auf den ersten Blick fällt auf, dass die Figur sehr einfach und dennoch einheitlich gestaltet ist. Der Kopf ist eine Kugel, der Körper ein Zylinder, der oben leicht eingezogen ist, und die Hände sind einteilig, das heißt, es gibt keine Finger, sondern nur eine Kugel als Hand.

Nach der Analyse wurde eine grundlegende Skizze des Körpers mit Procreate angefertigt.

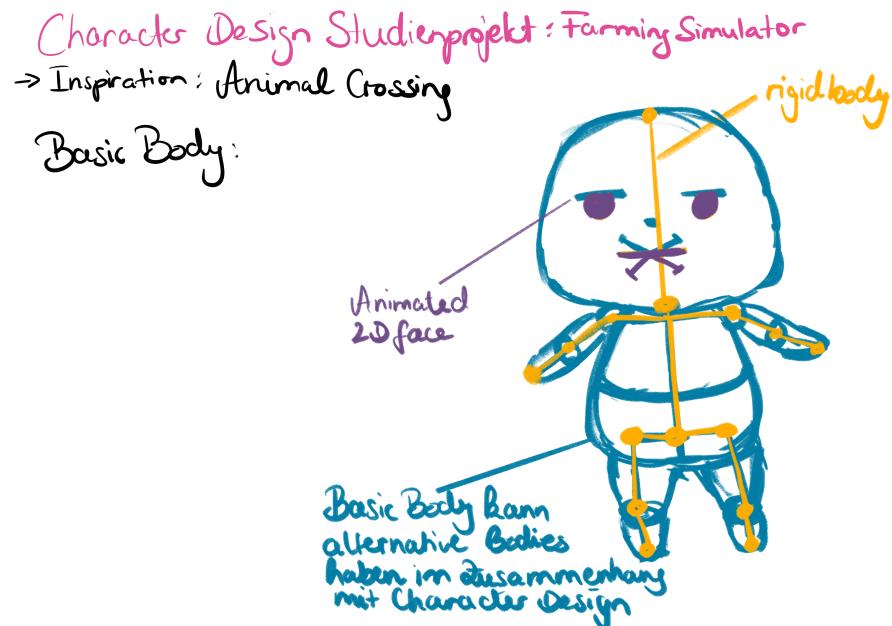


Abb. 3.3: Charakter Skizze

In Abbildung 3.3 wird der Charakter grob skizziert. Das grundlegende Körpermodell wird je nach Bedarf angepasst. Der Rigid Body wird später für Animationen verwendet. Das Gesicht kann entweder statisch oder in 2D animiert sein, abhängig von den Anforderungen.

3.1.3 Gestaltung des Inventarsystems

In diesem Kapitel wird das Konzept des Inventarsystems behandelt.



Abb. 3.4: Minecraft Inventarsystem (6)

In Abbildung 3.4 sieht man die Inspiration für das Inventarsystem, das im Farming Game verwendet wird. Es weist ein einfaches Design mit Slots und einem Ausrüstungsslot auf.

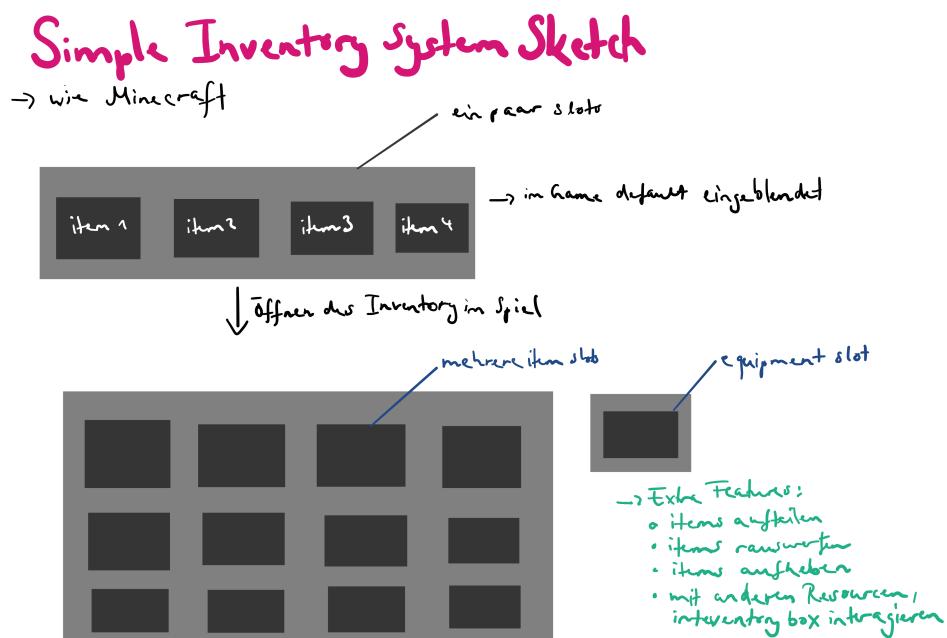


Abb. 3.5: Einfaches Inventory System

Wie in Abbildung 3.4 zu sehen ist, soll ein Inventarsystem geschaffen werden, das dem Spiel Minecraft ähnelt. Dazu wurde eine Skizze erstellt (3.5). Die Skizze zeigt, dass im Standardmodus das Inventar eingeklappt ist und nur die Gegenstände der ersten Reihe angezeigt werden. Beim Aufklappen des Inventars werden mehrere Slots sichtbar.

Die Anzahl der Slots pro Reihe kann nach den Wünschen des Entwicklers festgelegt werden. Ebenso ist ein Ausrüstungsslot vorhanden, der nur Werkzeuge des Charakters aufnimmt. Das bedeutet beispielsweise, dass Früchte, Gemüse oder Samen nicht in den Ausrüstungsslot gelegt werden können.

Die Funktionen des Inventarsystems umfassen:

- Aufteilen von Gegenständen durch Klicken
- Auswerfen von Gegenständen
- Einsammeln von Gegenständen
- Interaktion mit anderen Ressourcen wie Inventarboxen

3.1.4 Aufbau der Datenstruktur des Inventarsystems

Bevor man sich auf die Gestaltung der Szenen und die Erstellung von Skripten konzentriert, ist es wichtig, eine allgemeine Datenstruktur für das Konzept zu definieren. Wie sind die einzelnen Elemente zu realisieren?

Sketch Konzept Inventorysystem Datenstruktur



Abb. 3.6: Skizze von Datenstruktur des Inventarsystems

In der Abbildung 3.6 wird eine einfache Darstellung der Umsetzungsmöglichkeit der Datenstruktur eines Inventarsystem gezeigt. Ein Slot speichert eine Referenz von den Item-Daten sowie die Menge des jeweiligen Items. Die Items werden im Slot-Daten gespeichert. Das Inventar ist letztendlich eine Speicherung von Slot-Daten als Array.

Jede Szene die erstellt wurde, ist dann mit den entsprechenden Daten verbunden. Die Szene kommuniziert mit den jeweiligen Daten über Signale.

3.2 Praktische Umsetzung des Farming Simulators

In diesem Unterkapitel wird die praktische Umsetzung des Farming Simulators behandelt. Dabei werden die verwendeten Technologien näher betrachtet und erläutert.

3.2.1 Nomad Sculpt

Der Charakter wird in Nomad erstellt, einer 3D-Sculpting-App, die auf vielen Geräten läuft. Allerdings ist sie besonders für Tablets mit drucksensitivem Stift optimiert, wie beispielsweise den Apple Pencil oder den Samsung Galaxy Tablet-Stift (7).



Abb. 3.7: Nomad Logo (7)

Inspiriert wurde die App von anderen Desktop-Anwendungen wie ZBrush und Blender. Der Fokus von Nomad liegt auf der einfachen Bedienung der Benutzeroberfläche, ohne dabei wichtige Funktionen zu vernachlässigen (7).

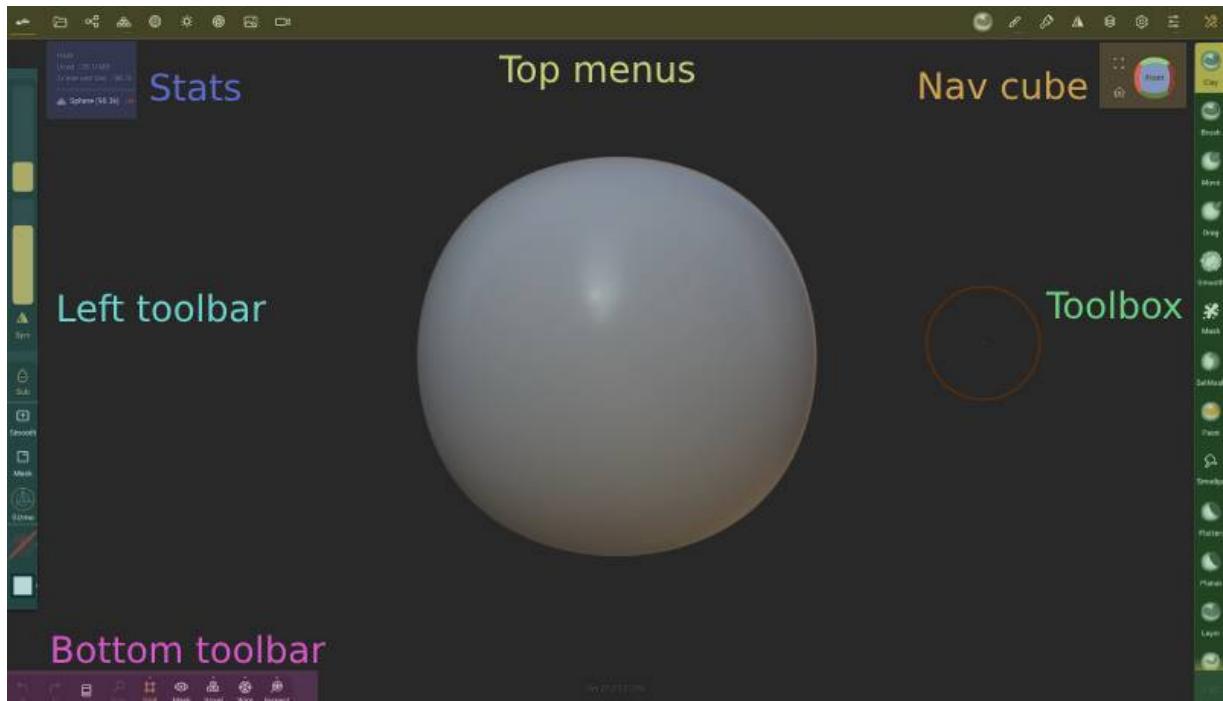


Abb. 3.8: Interface Nomad Sculpt (7)

In der Abbildung 3.8 ist das Interface von Nomad zu sehen, einschließlich der einzelnen Elemente zum Navigieren der verschiedenen Tools.

Im Top-Menü sind die meisten Nomad-Features zu finden. Auf dem oberen linken Menü befinden sich die Szene- und Objekt-Features, während auf der rechten Seite die Menüs für die Tools angeordnet sind. Auf kleineren Bildschirmen klappen diese Menüs zusammen, um Platz zu sparen. Das Stats-Fenster liefert Informationen zur Szene, dem ausgewählten Objekt, dem Maskenstatus und dem Speicherverbrauch. Der Nav Cube unterstützt bei der Orientierung und zeigt an, auf welcher Seite der Skulptur man sich befindet. Diesen kann man auch als Shortcut verwenden, um zu einer bestimmten Ansicht zu springen. Durch Ziehen des Cubes dreht sich die Ansicht entsprechend. Ein Klick auf das Rahmensymbol ermöglicht es, auf das ausgewählte Objekt zu fokussieren oder zur Standard-Heimansicht zurückzukehren. Die Toolbox ist scrollbar und gewährt Zugriff auf die einzelnen Tools. In der linken Toolbar findet man Slider für den Radius oder die Intensität der meisten Tools, kontextbasierte Knöpfe für andere Tools, Shortcuts für Symmetrie, den Alt/Sub-Modus, Maskierung, Glättung, das Gizmo und die Maloption (7).

Das Bottom-Toolbar besteht aus einem klassischen Undo-Button, der die vorgenommenen Operationen rückgängig macht, und einem Redo-Button, der die Operationen wiederherstellt. Der History-Button ermöglicht das Anzeigen der Historie. Der Solo Toggle-Button zeigt entweder das aktuelle Objekt oder alle Objekte an. Das Grid-Button zeigt die Gitter an, und durch langes Halten erscheinen mehrere Optionen für das Grid. Der Mask-Button verbirgt die maskierte Region des ausgewählten Objekts, was nützlich ist, um sich auf einen bestimmten Bereich zu konzentrieren. Der Voxel-Button dient als Shortcut für den Voxel Remesher. Durch langes Halten auf den Voxel-Button können Optionen für die Voxel Remesh-Qualität eingestellt werden. Der Wire-Button schaltet das Drahtgitter ein und aus. Das Inspect-Button zeigt zusätzliche Daten zum aktuellen Mesh an. Die Standardanzeige zeigt die UVs an, aber durch langes Drücken oder Wischen nach oben können die Eigenschaften inspiziert werden, sofern vorhanden, und ob diese im Hintergrund oder auf dem Mesh angezeigt werden (7).

Um mehr über Nomad zu erfahren, empfiehlt es sich, die offizielle Seite von Nomad zu besuchen: <https://nomadsculpt.com/manual/>.

3.2.2 Realisierung der Modelle mit Nomad Sculpt

Da es einige Vorarbeit erfordert, die Anwendung Nomad Sculpt zu verstehen, wurden verschiedene Prototypen erstellt.

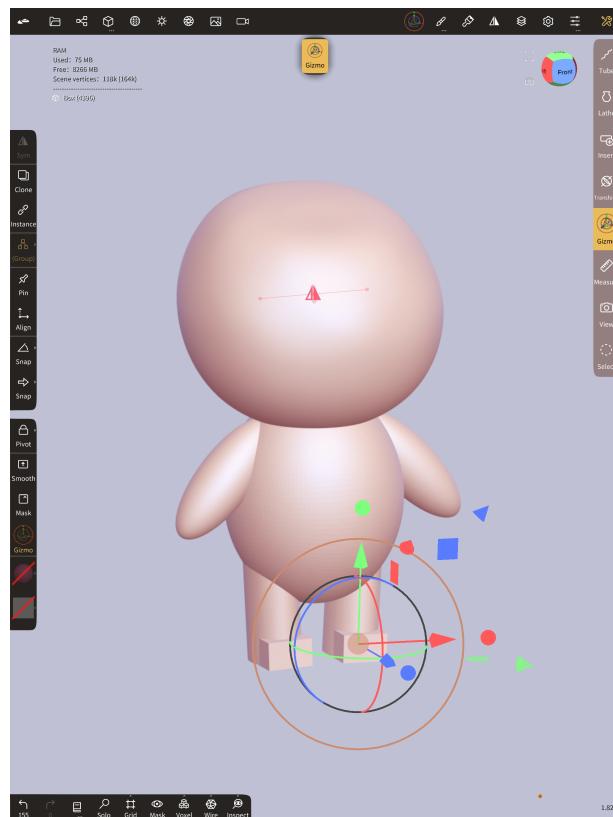


Abb. 3.9: Erster Prototyp Charakter mit Nomad Sculpt

Beim Prototypen in Abbildung 3.9 ist zu erkennen, dass der Charakter sehr einfach gehalten wurde. Obwohl hier mit Nomad Sculpt experimentiert wurde, entstand anschließend ein neuer Charakter. Da der Prototyp nicht den Anforderungen entsprach, die der Charakter erfüllen sollte.



Abb. 3.10: Zweiter Prototyp Charakter mit Nomad Sculpt

In Abbildung 3.10 ist zu sehen, dass der zweite Prototyp des Charakters einen verbesserten Aufbau im Vergleich zum ersten Prototypen in Abbildung 3.9 aufweist. Als Referenz dient die Abbildung von Animal Crossing 3.2 aus dem Konzeptualisierungskapitel. Das Modell hat nun zwei Arme mit kugelförmigen Händen, Ohren und einen Hals.



Abb. 3.11: Charakter ohne richtigen Torso

In Abbildung 3.11 ist der Charakter in einem etwas fortgeschrittenen Zustand zu sehen. Der Charakter verfügt nun über Beine und Füße. Ein Problem, das sich bei diesem Charakter jedoch herausstellte, war der Torso. Dieser sollte eine Einkerbung haben, um den Ober- und Unterkörper zu definieren. Ohne einen gut strukturierten Torso traten Schwierigkeiten beim Einsatz der Mixamo-Software von Adobe auf.



Abb. 3.12: Charakter mit richtigen Torso

In der oberen Abbildung 3.12 ist der Charakter mit einem gut strukturierten Torso zu sehen. Beim Animieren des Charakters oder beim Einsatz des Auto-Rigging mit Mixamo treten nun weniger Probleme auf. Diese Erkenntnis war jedoch vorhersehbar, da die meisten Modelle in Mixamo einen menschlichen Torso mit einer klaren Abtrennung zwischen Oberkörper und Hüftbereich aufweisen.



Abb. 3.13: Charakter ohne Paint

In der Abbildung 3.13 ist der fast fertige Charakter mit einem Kimono zu sehen. Auf den Kopf wurden zwei Accessoires hinzugefügt, einmal eine Schleife und einmal eine Katzenmaske als Haarreif, sowie Sandalen. Hier ist auch zu erkennen, dass der Charakter noch nicht bemalt wurde.



Abb. 3.14: Charakter mit Paint

In Nomad ist es möglich, Objekte mit dem Paint Tool zu bemalen. In der Abbildung 3.14 sind nun Farben zu erkennen. Das Design wurde hier bewusst einfach gehalten, und das Gesicht wird ein statisches Lächeln darstellen. Dabei wurde darauf geachtet, dass der Charakter, ähnlich wie in Animal Crossing 3.2, verniedlicht wird.



Abb. 3.15: Charakter UV-Unwrap UV-Atlas

Um das Material zusammen mit dem 3D-Modell zu exportieren, muss das Modell mit dem UV-Atlas von Nomad entfaltet werden. In Abbildung 3.15 ist der UV-Unwrap Prozess hinter dem 3D-Charakter zu sehen. Nach diesem Vorgang ist es nun möglich, das Modell mit dem Material und den Texturen als glTF (Graphics Library Transmissions Format) zu exportieren. Das glTF-Format ermöglicht die Verwendung des Modells im Spiel und kann leicht in andere Spiele oder Anwendungen exportiert werden.

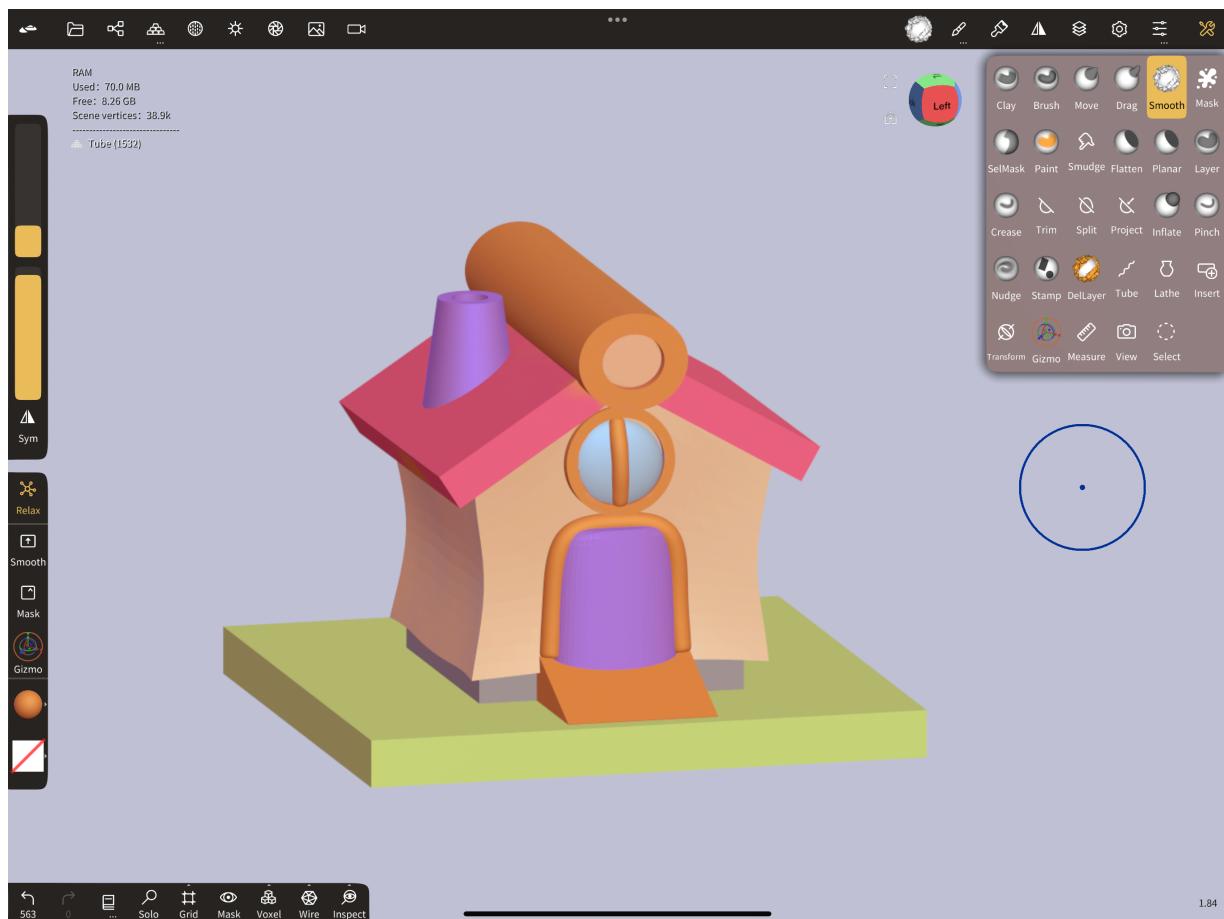


Abb. 3.16: Haus Design

In der Abbildung 3.16 ist das Haus zu sehen, welches später in die Spielszene implementiert wird.

3.2.3 Mixamo von Adobe und Blender

Dieses Unterkapitel behandelt die Umsetzung der Animation des Charakters mithilfe von Mixamo und Blender.

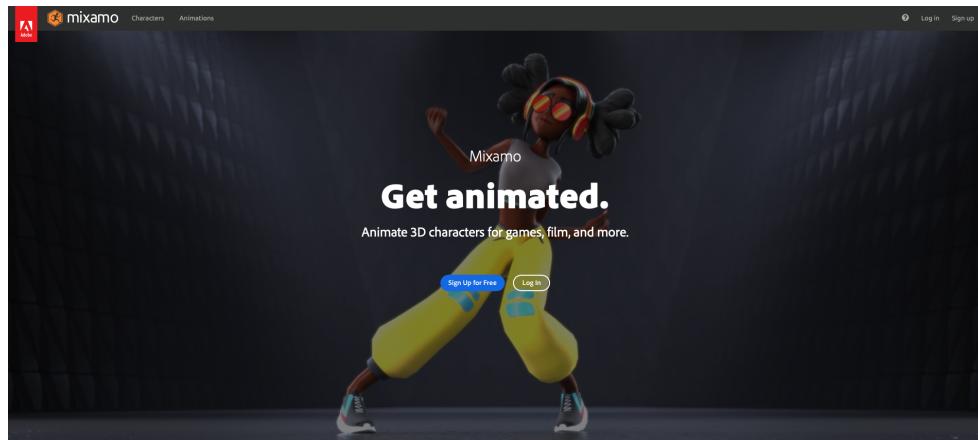


Abb. 3.17: Mixamo Startpage (8)

Mixamo ist ein Online-Service von Adobe, es ermöglicht, 3D-Charaktere hochzuladen und zu animieren. Zudem bietet Mixamo auch eigene 3D-Charaktere an, die ebenfalls genutzt werden können.

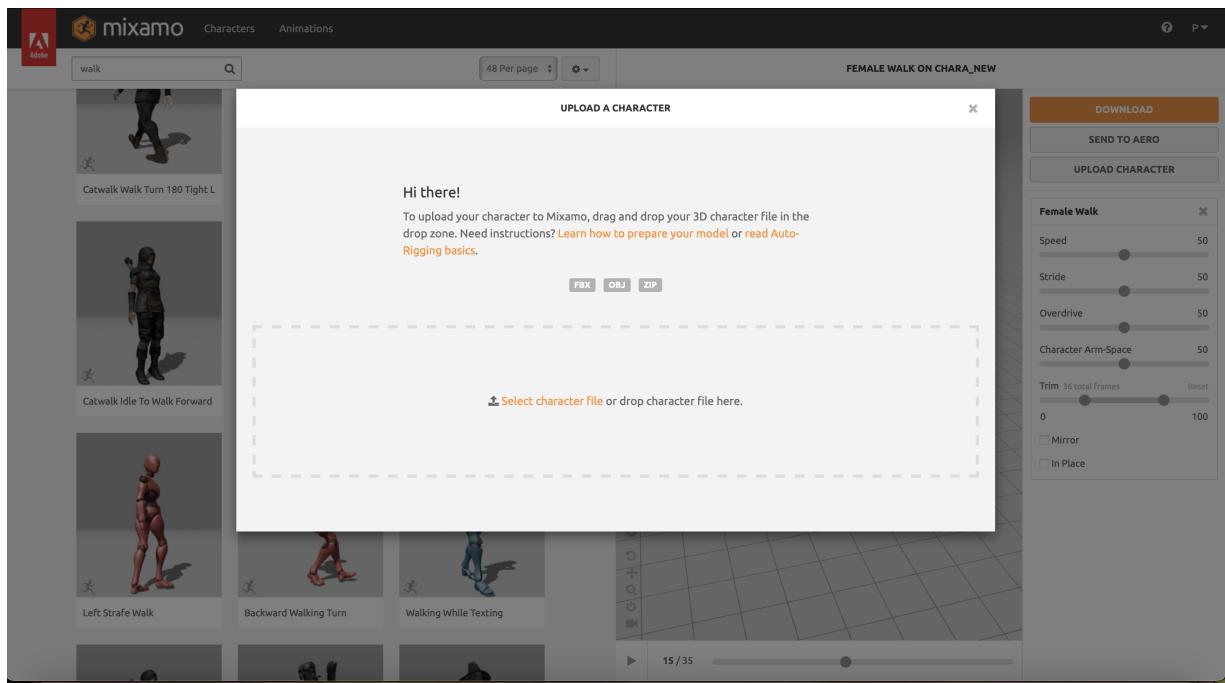


Abb. 3.18: Hochladen des 3D Charakters

Beim Hochladen kann man das Modell als FBX, OBJ oder ZIP-Datei hochladen. Der zuvor erstellte Charakter wurde als glTF von Nomad exportiert. Anschließend wurde

das Modell in Blender importiert und erneut als FBX in Blender exportiert. Nach dem Export wurde das 3D-Modell auf Mixamo hochgeladen. Durch das Auto-Rigging von Mixamo kann das Modell geriggt werden und anschließend mit verschiedenen Animationen versehen werden.

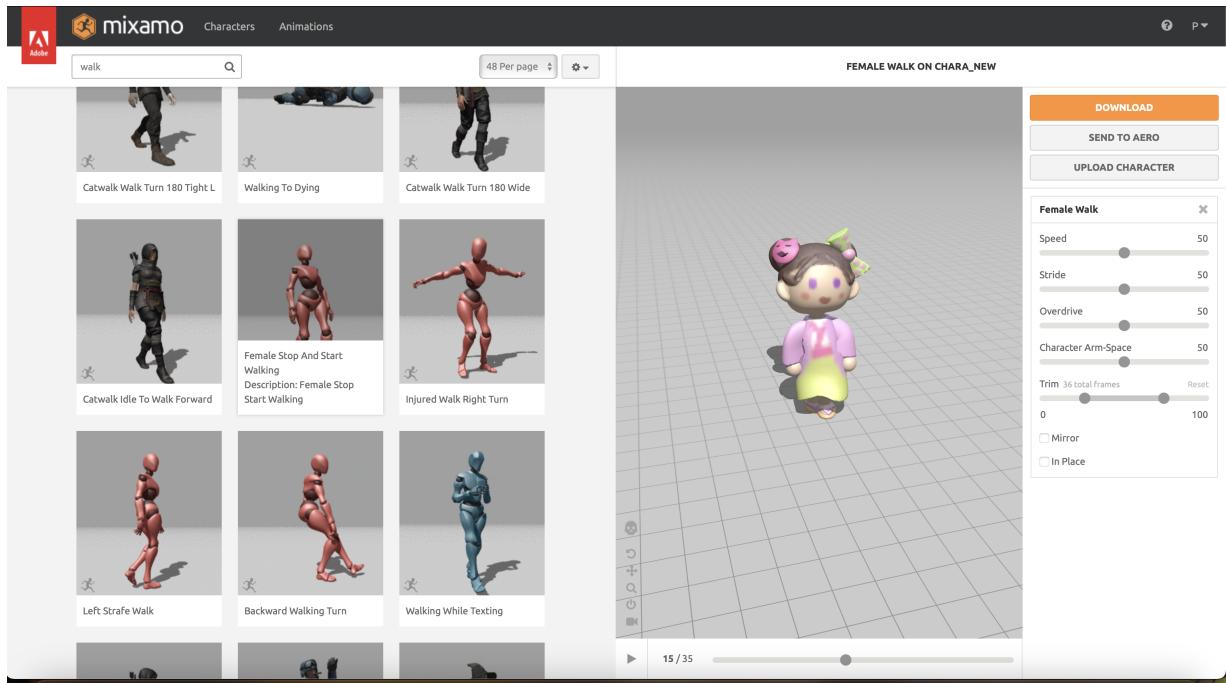


Abb. 3.19: Animation Library von Mixamo

In der Abbildung 3.19 sind die verschiedene Animationen zu sehen, die dem Charakter zugeordnet werden können. Auf der rechten Seite sind zusätzliche Optionen sichtbar, wie beispielsweise die Geschwindigkeit der Laufbewegung. Es ist wichtig, dass die Laufanimation im Spiel an Ort und Stelle ausgeführt wird, da die Animation beim Bewegen des Charakters im Loop abgespielt wird. Andernfalls würde der Charakter immer wieder zur Ausgangsposition zurückkehren, und die Laufbewegung im Spiel würde dabei ins Stocken geraten.

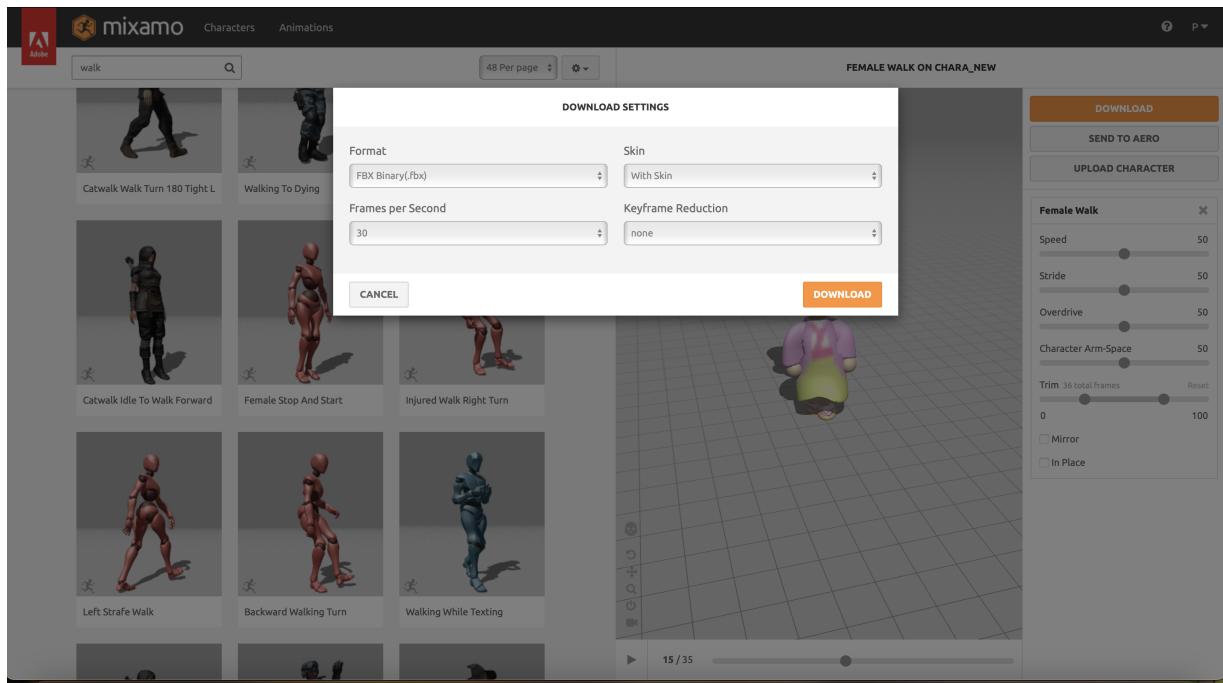


Abb. 3.20: Mixamo exportieren der Animation

In der Abbildung 3.20 ist der Exportvorgang zu sehen. Als Format wird erneut FBX verwendet, wobei auch die Hautinformationen (Skin) mitgenommen werden. Anschließend kann die Anzahl der Frames pro Sekunde festgelegt werden; hier wurden 30 FPS gewählt. Nach den Einstellen kann das Modell mit der Animation heruntergeladen werden.

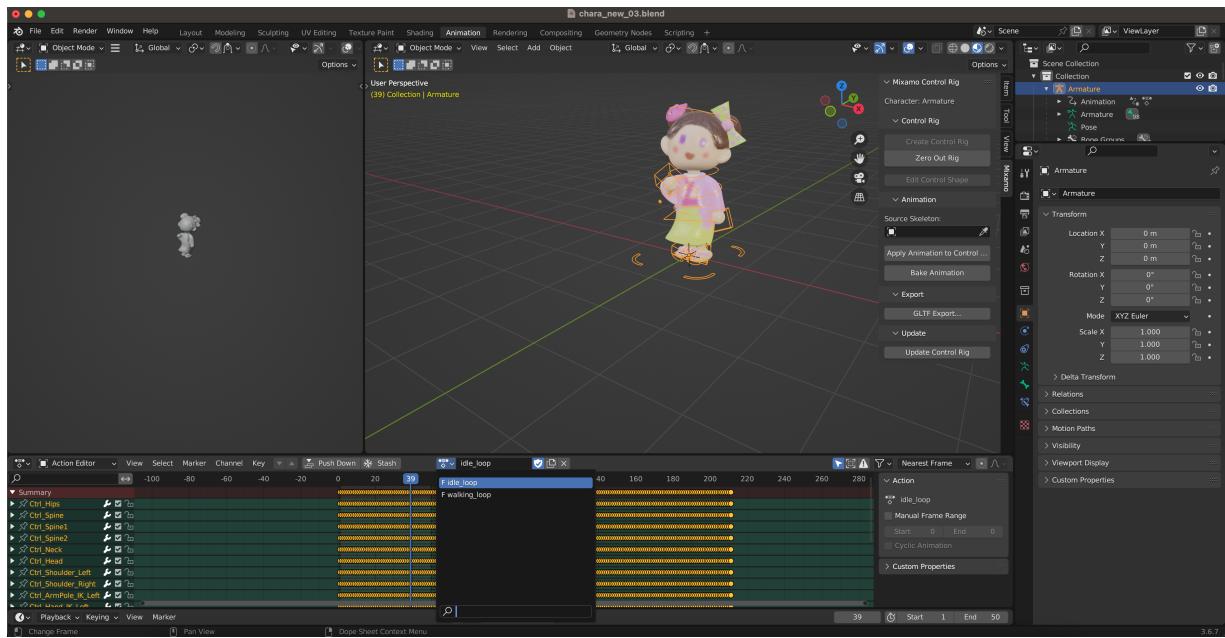


Abb. 3.21: Mixamo und Blender Actions

In Blender ist es dann möglich, die Animationen zu importieren und mithilfe des Mixamo auto control-rig-add-on das Modell zu riggen und die Animationen einzubinden. Zurzeit gibt es Probleme mit der neusten Blender Version und dem Mixamo Plugin, deswegen ist es ratsam die derzeitige LTS Version (v3.6.7) von Blender zu benutzen. Weitere Infos zu dem Auto Rigging Addon findet man auf der Seite <https://substance3d.adobe.com/plugins/mixamo-in-blender/>.

In der Abbildung 3.21 sieht man, dass in Blender der Figur die Animationen "Walking" und "Idle" zugewiesen wurden, indem die Animationen als Aktionen für ein Modell hinzugefügt werden. Das bedeutet, dass das 3D-Modell nun die beiden Aktionen "Walking" und "Idle" enthält. Außerdem ist die Bezeichnung des Actions sehr wichtig. Wenn die Actions nicht richtig mit "loop" benannt werden, könnte es Probleme mit der Animation in Godot geben. Denn mit der Bezeichnung "loop", erreicht man automatisch einen Schleifenvorgang beim importieren des Charakters in Godot. Die Animationen sollten alle bei Frame 0 beginnen, um später beim Importieren in Godot ein stotterfreies Abspielen zu ermöglichen.

3.2.4 Blender und Godot

Dieses Unterkapitel behandelt die Verwendung von Blender und Godot.

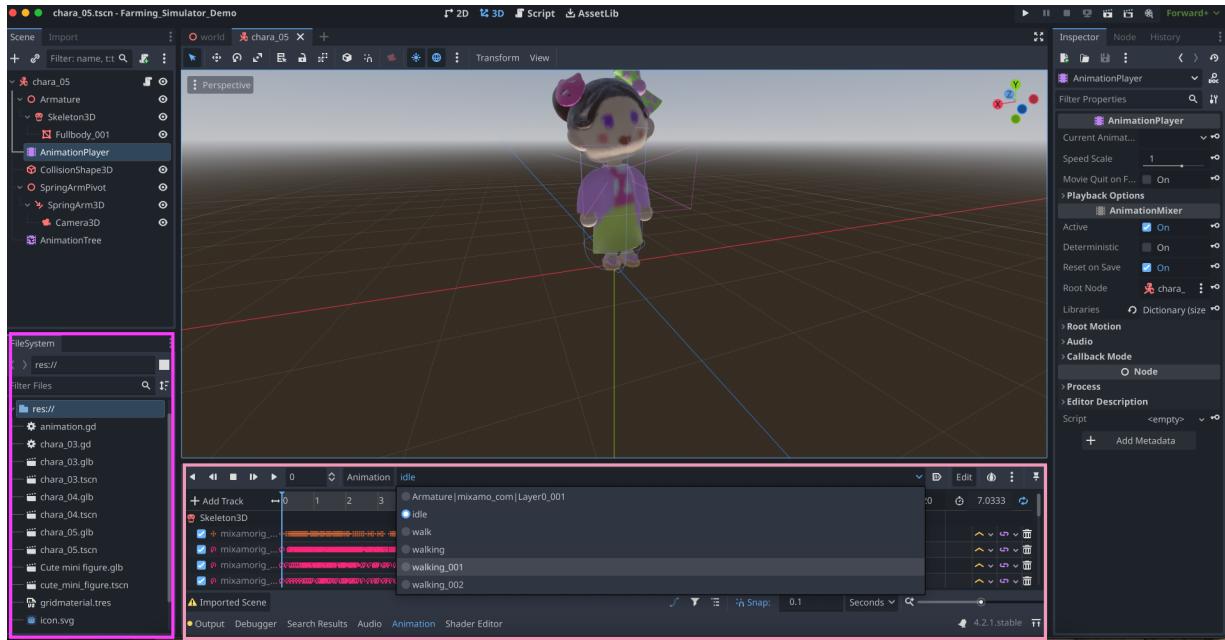


Abb. 3.22: Importieren des 3D-Modells und Animationsplayer in Godot

Um einen 3D-Charakter in Godot zu importieren, muss dieser zuerst im glTF-Format exportiert werden. Der Export erzeugt eine Datei mit der Endung ".glb", die dann durch einfaches Drag and Drop in das Verzeichnis von Godot importiert werden kann. Nach dem Import erstellt man eine Szene für den 3D-Charakter. Nach dem Import überprüft man, ob die Animationen und Materialien korrekt angezeigt werden. Wie erwartet laufen die Animationen in einer "loop"-Schleife, da in Blender die jeweiligen Aktionen mit der Benennung "loop" versehen wurden.

3.2.5 Umsetzung der Charakteranimationen: "Idle" und "Walk" in Godot

In diesem Unterkapitel wird die Umsetzung der Charakteranimationen "Walk" und "Idle" in Godot beschrieben.

Das Spiel wurde mit Godot v4.2.1.stable.official entwickelt und sollte dementsprechend auch mit dieser Godot-Version geöffnet werden. Das Öffnen mit einer anderen Version von Godot könnte dazu führen, dass die 3D-Objekte nicht korrekt dargestellt werden und beschädigt auftreten.

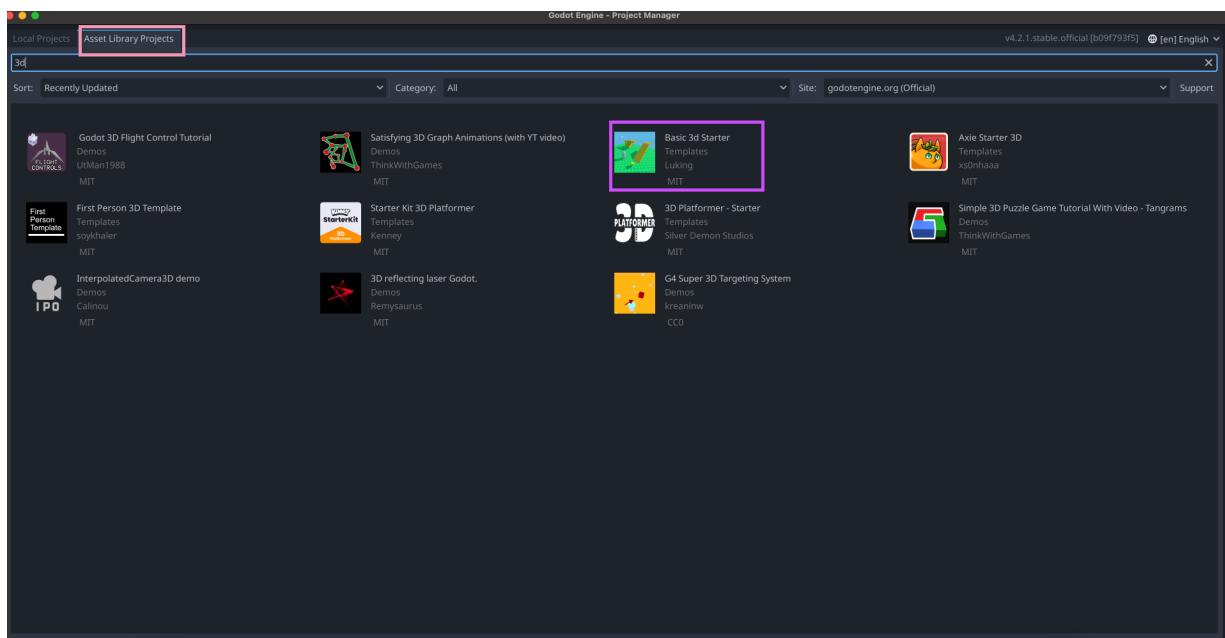


Abb. 3.23: Godot Asset Library: Basic 3D Starter

Bevor der Charakter ins Spiel importiert wurde, hat man als ersten Schritt aus der Godot Asset Library das Basic 3D Starter heruntergeladen. Dieses Template diente als Grundlage für die Erstellung des Spiels. Es bietet Anfängern ein Beispieltemplate mit einer Kapsel als Charakter und einer 3D-Plattform. Darüber hinaus enthält das Template weitere Elemente wie animierte Plattformen, ein beeinflussbares Objekt (Würfel) und Textboxen.

In Abbildung 3.22 ist zu sehen, dass nach dem import ein inherited Scene für den Charakter erstellt wurde. In Godot werden erstellte Szenen als ".tscn" bezeichnet.

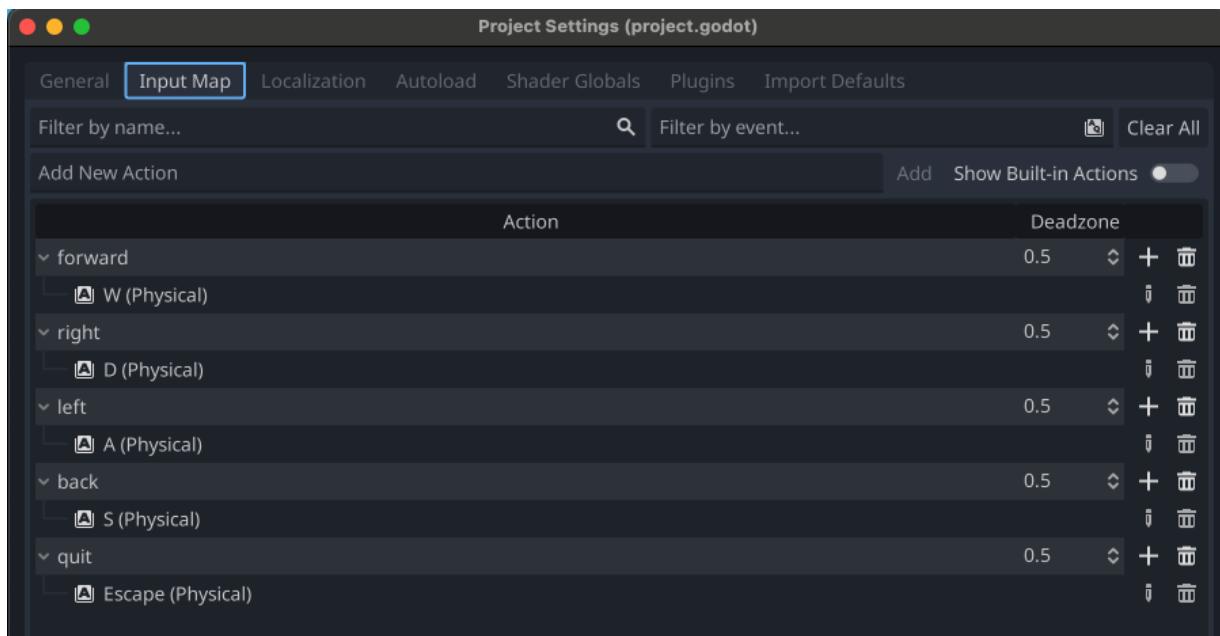


Abb. 3.24: Godot Project Settings

Bevor man mit dem Programmieren und dem Erstellen der anderen Nodes beginnt, legt man in den Project Settings die Input Map fest. Hier können Tasturbefehle auf Actions verteilt werden, wie beispielsweise Vorwärts (W), Rückwärts (S), Rechts (A) und Links (D). Der Tasturbefehl "quit" wird später für das Beenden des Fensters oder des Spiels verwendet.

In der Abbildung 3.22 ist auch das Root Node des Charakters zu sehen. Hierbei handelt es sich um einen CharacterBody3D. Man erkennt zudem, dass der Charakter von einem CollisionShape3D umgeben ist. Dieses dient dazu, mit anderen Objekten in Godot zu kollidieren. Für den Charakter wurde ein CapsuleShape3D ausgewählt und entsprechend skaliert.

Nach der Erstellung des CollisionShape3D wird die Kamera eingerichtet. Diese wird als Root ein Node3D namens SpringArmPivot haben. Diese wird verwendet, um die Kamera zu drehen. Dem SpringArmPivot wird dann ein SpringArm3D zugewiesen, und anschließend bekommt das SpringArm3D eine Camera3D zugewiesen. Das SpringArm3D dient dazu, die Länge des Abstands zwischen der Kamera und dem Charakter festzulegen. Außerdem wird der SpringArm3D mit der Kamera auf die Schulterhöhe des Charakters in der y-Achse verschoben.

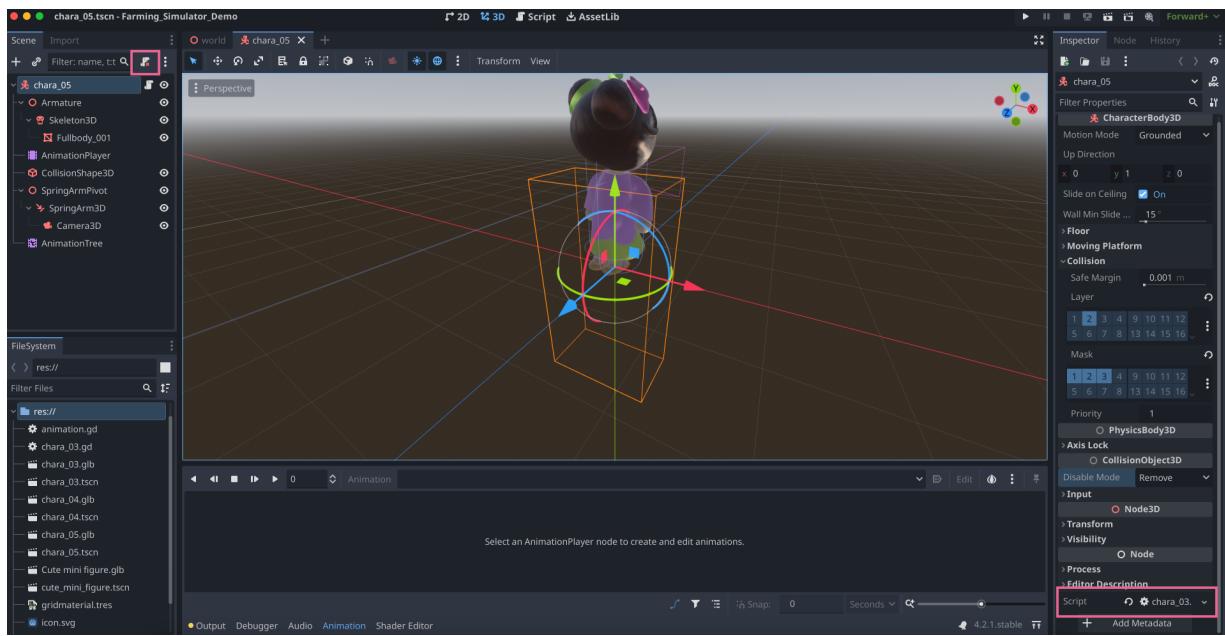


Abb. 3.25: Godot Script

Nun fügt man dem CharacterBody3D ein neues Skript hinzu. In der Abbildung 3.25 sind verschiedene Möglichkeiten dargestellt, wie ein Skript eingebunden werden kann. Man kann in Godot oben links bei der Szene ein Skript zum dazugehörigen Node erstellen oder rechts unten im Inspektor. Als Template nimmt man das, was bereits mit dem CharacterBody3D mitgeliefert wird.

Nachdem das Skript erstellt wurde, zieht man die Charakter-Szene in die World-Szene, um zu überprüfen, ob der Charakter und die Kamerabewegungen funktionieren.

List. 3.1: Character movement

```

1 extends CharacterBody3D
2
3
4 @onready var armature = $Armature
5 @onready var spring_arm_pivot = $SpringArmPivot
6 @onready var spring_arm = $SpringArmPivot/SpringArm3D
7 @onready var anim_tree = $AnimationTree
8
9 const SPEED = 5.0
10 const LERP_VAL = .15
11
12 # Get the gravity from the project settings to be synced with
   RigidBody nodes.
13 var gravity =
   ProjectSettings.get_setting("physics/3d/default_gravity")
14
15 func _ready():
   Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
16
17
18 func _unhandled_input(event):
19   if Input.is_action_just_pressed("quit"):
20     get_tree().quit()
21
22   if event is InputEventMouseMotion:
23     spring_arm_pivot.rotate_y(-event.relative.x * .005)
24     spring_arm.rotate_x(-event.relative.y * .005)
25     spring_arm.rotation.x = clamp(spring_arm.rotation.x, -PI/4, PI/4)
26
27 func _physics_process(delta):
28   # Add the gravity.
29   if not is_on_floor():
30     velocity.y -= gravity * delta
31
32   # Get the input direction and handle the movement/deceleration.
33   # As good practice, you should replace UI actions with custom
   gameplay actions.
34   var input_dir = Input.get_vector("left", "right", "forward",
35     "back")
35   var direction = (transform.basis * Vector3(input_dir.x, 0,
36     input_dir.y)).normalized()
36   direction = direction.rotated(Vector3.UP,
37     spring_arm_pivot.rotation.y)
37   if direction:
38     velocity.x = lerp(velocity.x, direction.x * SPEED, LERP_VAL)
39     velocity.z = lerp(velocity.z, direction.z * SPEED, LERP_VAL)

```

```
40     armature.rotation.y = lerp_angle(armature.rotation.y,
41             atan2(-velocity.x, -velocity.z), LERP_VAL)
42     else:
43         velocity.x = lerp(velocity.x, 0.0, LERP_VAL)
44         velocity.z = lerp(velocity.z, 0.0, LERP_VAL)
45
46     anim_tree.set("parameters/BlendSpace1Dblend_position",
47             Vector2(velocity.x, velocity.z).length() / SPEED)
48
49     move_and_slide()
```

In dem obigen Codeausschnitt sieht man, wie sich der Charakter bewegt. Hier kommen wieder die Input Maps zum Einsatz, die zu Beginn festgelegt wurden. Zu Beginn bewegt sich der Charakter ohne eine Animation durch die Welt. Als nächstes werden die Kameraeinstellungen vorgenommen. Gewünscht ist, dass die Kamera sich mit der Mausbewegung unabhängig vom Charakter bewegt. Deshalb wurden zu Beginn der SpringArmPivot, der SpringArm3D und die Camera3D zur Szene hinzugefügt. Mit dem InputEventMouseMotion-Event werden die Rotationen des Arms verändert. Um eine endlose Rotation zu verhindern, wird die clamp() Methode verwendet. Nun muss die Rotation der Kamera an die des Charakters angepasst werden, was bei der direction geschieht.

Als nächstes versuchen wir, die Animation einzubauen. Dazu muss zunächst programmiert werden, dass der Charakter sich in die Richtung dreht, in die er sich auch bewegt. Dies wird durch das Rotieren des Armature-Knotens erreicht. Für viele Aktionen wird eine Interpolation verwendet, für die ein LERP-Value erstellt wird. Der Charakter wird rotiert, wenn die Ausrichtung (direction) nicht Null ist.

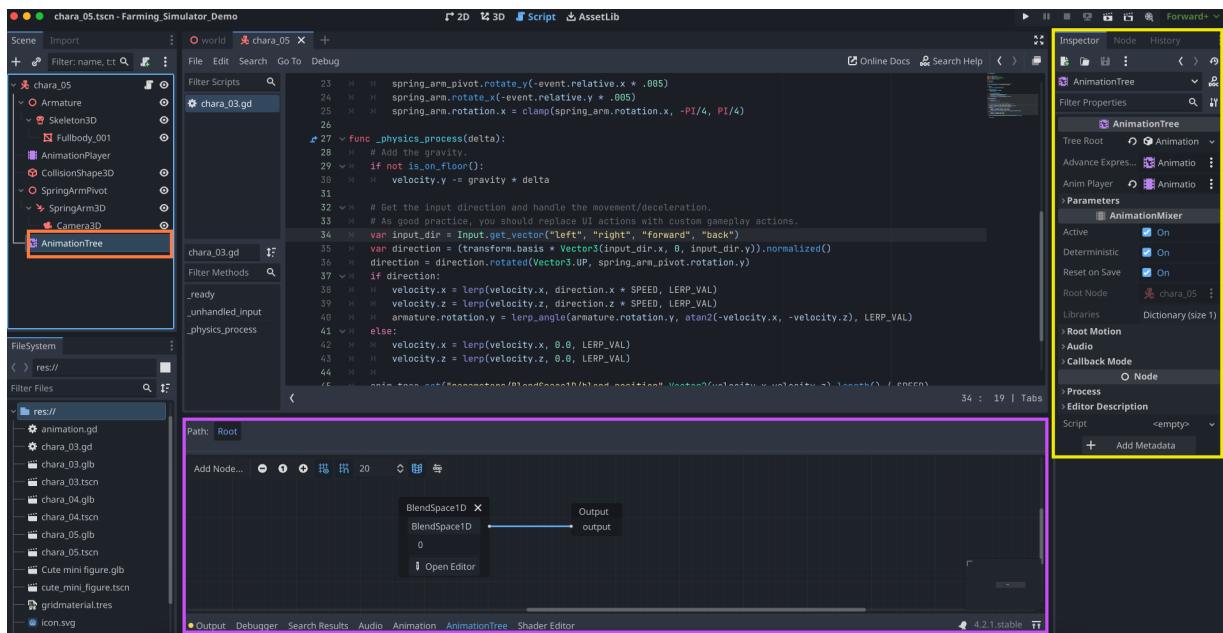


Abb. 3.26: Godot AnimationTree

Wenn der Charakter sich jetzt mit der Ausrichtung bewegt, ist der nächste Schritt das Hinzufügen der Animation. Dies wird durch einen AnimationTree-Knoten erreicht, wie in Abbildung 3.26 zu sehen ist. Zuerst fügt man den AnimationPlayer hinzu und erstellt ein BlendSpace1D.

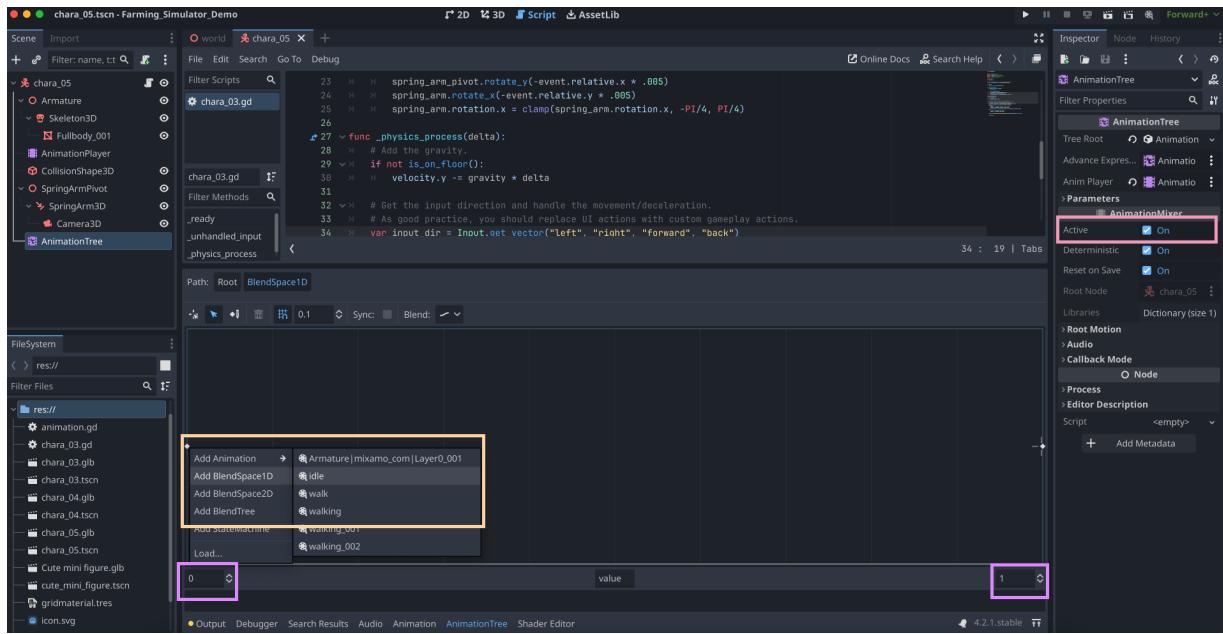


Abb. 3.27: Godot AnimationTree Editor

Danach öffnet man den Editor für das BlendSpace1D und fügt jeweils für Rechts und Links durch Rechtsklick die entsprechende Animation hinzu. Links fügt man die "Idle"-

Animation hinzu und rechts die "Walk"-Animation. Man setzt den Wert für Links auf 0 und für Rechts auf 1. Diese Werte bestimmen die Länge des Geschwindigkeitsvektors des Charakters. Anschließend verbindet man, wie in Abbildung 3.26 gezeigt, den BlendSpace1D mit dem Output und aktiviert den AnimationTree.

Um die Animation flüssig abzuspielen, muss die Geschwindigkeit interpoliert werden, und im Skript muss der Pfad (property Path) des AnimationTrees als Parameter hinzugefügt werden. Nun bewegt sich der Charakter flüssig und wechselt seine Animation je nach Geschwindigkeit zwischen "Idle" und "Walk".

3.2.6 Umsetzung des Inventarsystems: Das erste Prototyp

In diesem Kapitel wird die praktische Umsetzung des Inventarsystems behandelt. Dabei werden die einzelne Elemente wie Inventar Daten, Slot Daten und Item Daten näher beschrieben. Das beschriebene Inventarsystem, ist ein Prototyp und wird später mit den selbst erstellten Materialen und Items umgesetzt.

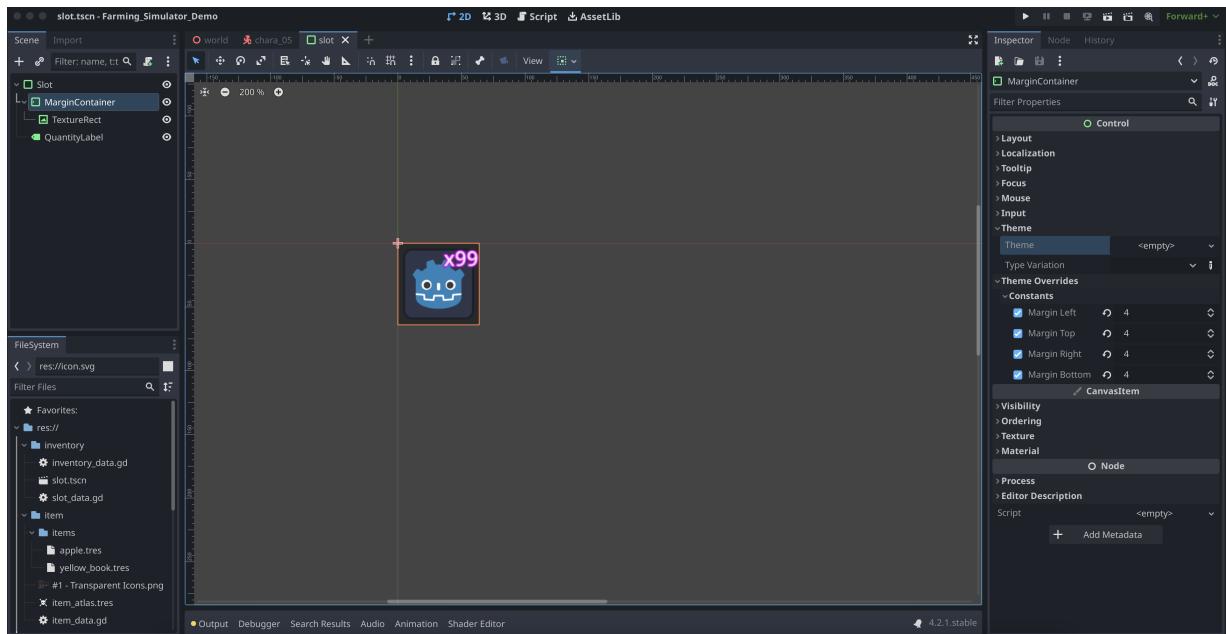


Abb. 3.28: Godot Slot UI

In Kapitel 3.1.4 hat man die grobe Datenstruktur von dem inventory festgestellt. Das Inventory besteht aus Slots, welche dann wiederum Item Daten enthalten. Zunächst wurden die Slots nur mit den User Interface erstellt, um die grobe Vorstellung zu haben, wie die Slots ausgerichtet sind und im Spiel integriert werden sollten. Als Placeholder wurde das das godot.svg Icon verwendet.

List. 3.2: inventory.gd

```

1    extends PanelContainer
2
3    const Slot = preload("res://inventory/slot.tscn")
4
5    @onready var item_grid: GridContainer = $MarginContainer/ItemGrid
6
7    func set_inventory_data(inventory_data: InventoryData) -> void:
8        inventory_data.inventory_updated.connect(populate_item_grid)
9        populate_item_grid(inventory_data)
10
11   func clear_inventory_data(inventory_data: InventoryData) -> void:
12       inventory_data.inventory_updated.disconnect(populate_item_grid)
13
14   #inventory updates
15   func populate_item_grid(inventory_data: InventoryData) -> void:
16       #clear the item grid of any children
17       for child in item_grid.get_children():
18           child.queue_free()
19
20       #slot for every slotdata -> child to the itemgrid
21       for slot_data in inventory_data.slot_datas:
22           var slot = Slot.instantiate()
23           item_grid.add_child(slot)
24
25           slot.slot_clicked.connect(inventory_data.on_slot_clicked)
26
27           # check slot data and update parameter
28           if slot_data:
29               slot.set_slot_data(slot_data)

```

Das Skript inventory.gd implementiert das Inventarsystem für das Spiel.

Das Skript erweitert die Klasse für den PanelContainer Node und stellt einen Container für die UI Elemente dar. Die Funktion set_inventory_data stellt die Inventar Daten für das Inventarsystem fest. Diese stellt auch eine Verbindung durch das Signal inventory_updated des Inventar Daten Objekts mit der poulated_item_grid her.

Die Funtion ruft das populate_item_grid auf um das Raster mit den Inventar Daten zu füllen. Das populate_item_grid löscht die vorhandenen children des Rasters. Außerdem durchläuft es alle SlotData im Inventar und jeder Slot wird als child zu Raster hinzugefügt. Die Funktion gibt auch ein Signal slot_clicked des jeweiligen Slots an die Funktion on_slot_clicked der Inventar Daten weiter. Wenn nun die Slot Daten gelten, werden die Slot Daten für die jeweiligen Slots gesetzt und geupdated.

Die Funktion clear_inventory_data trennt den Signal inventory_updated von der Funktion populated_item_grid.

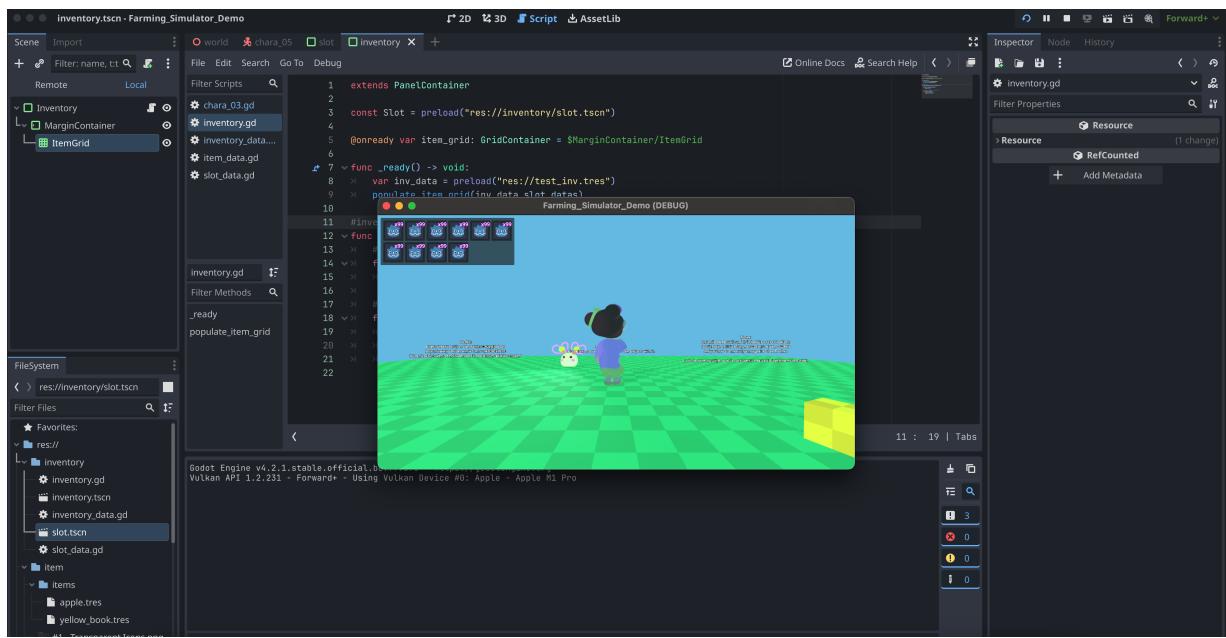


Abb. 3.29: Godot Populate Slot Data

Auf der Abbildung 3.29 sieht man oben die Vermehrung der Inventar Daten im Inventar im User Interface des Spiels. Hier wurden die Slots hinzugefügt mit dem Prototyp Texture godot.svg.

List. 3.3: inventory_data.gd

```
1  extends Resource
2  class_name InventoryData
3
4  signal inventory_updated(inventory_data: InventoryData)
5  signal inventory_interact(inventory_data:
6      InventoryData, index:int, button:int)
7
8  @export var slot_datas: Array[SlotData]
9
10 func grab_slot_data(index: int) -> SlotData:
11     var slot_data = slot_datas[index]
12
13     #check if grabbed
14     if slot_data:
15         slot_datas[index] = null
16         #update inventory if grabbed
17         inventory_updated.emit(self)
18         return slot_data
19     else:
20         return null
21
22 func drop_slot_data(grabbed_slot_data: SlotData, index: int) ->
23     SlotData:
24     var slot_data = slot_datas[index]
25
26     var return_slot_data: SlotData
27
28     if slot_data and
29         slot_data.can_fully_merge_with(grabbed_slot_data):
30             slot_data.fully_merge_with(grabbed_slot_data)
31     else:
32         slot_datas[index] = grabbed_slot_data
33         return_slot_data = slot_data
34
35     inventory_updated.emit(self)
36     return return_slot_data
37
38     #slot_datas[index] = grabbed_slot_data
39     #inventory_updated.emit(self)
40     #swapping if something has an item_data in the slot
41     #return slot_data
42
43
44     #to drop single item into slot
45     func drop_single_slot_data(grabbed_slot_data: SlotData,
```

```
43     index: int) -> SlotData:
44         var slot_data = slot_datas[index]
45
46         if not slot_data:
47             slot_datas[index] =
48                 grabbed_slot_data.create_single_slot_data()
49         elif slot_data.can_merge_with(grabbed_slot_data):
50             slot_data.fully_merge_with(grabbed_slot_data.
51                 create_single_slot_data())
52
53         inventory_updated.emit(self)
54
55         if grabbed_slot_data.quantity > 0:
56             return grabbed_slot_data
57         else:
58             return null
59
60 #consume consumable item_data with right click
61 func use_slot_data(index: int) -> void:
62     var slot_data = slot_datas[index]
63
64     if not slot_data:
65         return
66
67     if slot_data.item_data is ItemDataConsumable:
68         slot_data.quantity -= 1
69         if slot_data.quantity < 1:
70             slot_datas[index] = null
71
72     print(slot_data.item_data.name)
73     CharacterManager.use_slot_data(slot_data)
74
75     inventory_updated.emit(self)
76
77 #pick up the pick_up item into slot_data (walk to the item
78 func pick_up_slot_data(slot_data: SlotData) -> bool:
79
80     # search mergable slot and take that instead
81     for index in slot_datas.size():
82         if slot_datas[index] and
83             slot_datas[index].can_fully_merge_with(slot_data):
84             slot_datas[index].fully_merge_with(slot_data)
85             inventory_updated.emit(self)
86             return true
```

```
87     for index in slot_datas.size():
88         if not slot_datas[index]:
89             slot_datas[index] = slot_data
90             inventory_updated.emit(self)
91         return true
92
93     return false
94
95 #click on inventoriesystem get signal
96 func on_slot_clicked(index:int, button:int) -> void:
97     #print("inventory interact")
98     inventory_interact.emit(self, index, button)
```

Das Skript inventory_data.gd fügt die Inventardaten in das Inventarsystem des Spiels hinzu.

Die Klasse InventoryData ist die Repräsentation für die Inventardaten und erweitert die Ressource Klasse. Das Signal inventory_updated löst dich aus, wenn das Inventar verändert wird. Das inventory_interact wird ausgeführt wenn das Inventar geklickt wird.

Das slot_datas ist ein Array und speichert die jeweiligen Items. Die grab_slot_data Funktion wird aufgerufen um bestimmte Slot Daten zu greifen oder anzunehmen. Diese ruft auch das Signal inventory_updated aus, um die Slot Daten aus dem Inventar zu entfernen, wenn diese schon vorhanden sind.

Die Funktion drop_slot_data wird verwendet um ein Slot Data herauswerfen zu können. Diese Funktionachtet auch darauf, ob die Slot Daten mit den schon im Inventar vorhanden Slot Daten zusammengefasst werden kann. Außerdem ruft sich auch das Signal inventory_updated aus, um das Inventar zu aktualisieren.

Die drop_single_slot_data Funktion wird verwendet, um einzelne Slot Daten am jeweiligen Index herauszuschmeißen. Hier wird wieder geprüft, ob die vorhanden Slot Daten mit den neu hinzugefügten Slot Daten zusammengefasst werden können. Das Signal inventory_updated wird auch hier ausgeführt.

Die Funktion pick_up_slot_data wird verwendet, um Slot Daten aufzusammeln, wenn der Charakter sich zu einem Item bewegt. Hier wird wieder geschaut, dass die aufgenommen Slot Daten mit den bereits vorhandenen Slot Daten im Inventar zusammengefügt werden. Wenn das Slot schon voll ist, wird ein neuer Slot für die aufgesammelten Slot Daten erzeugt.

Die use_slot_data Funktion wird für Slot Daten verwendet, welche benutzt werden können. Das heißt, wenn diese SlotDaten verwendet werden, wird auch im Inventar die Slot Daten in der Menge reduziert. Hier wird wieder das Inventar mit inventory_updated aktualisiert.

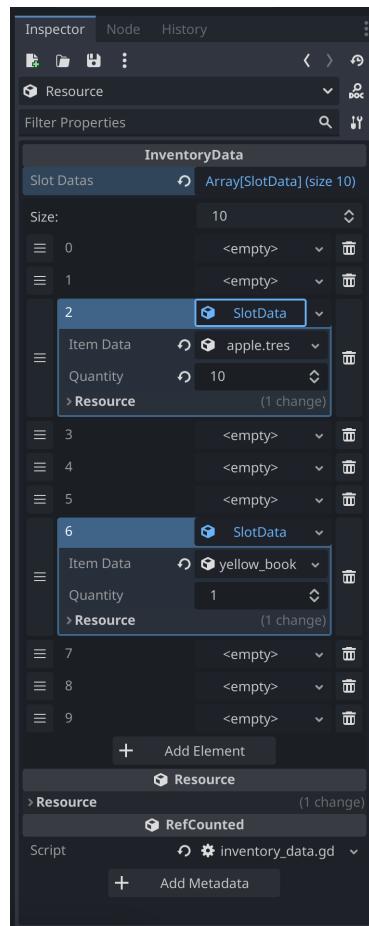


Abb. 3.30: Charakter Inventarsystem: Inventar Daten, Slot Daten

In der Abbildung 3.30 ist das Inventar vom Charakter in Godot zu sehen. Abgebildet sind die Slot Daten vom InventoryData. Man kann die Slots mit Item Daten füllen, indem man die erstellten item.tres Daten als Item Data hinzufügt und die Menge von den jeweiligen Slot Daten bestimmt.

List. 3.4: inventory_interface.gd

```
1  extends Control
2
3  #send signal to main for pick up item
4  signal drop_slot_data(slot_data: SlotData)
5
6  #force close inventory if to far from interactable
7  signal force_close
8
9  #variable to store slot_data
10 var grabbed_slot_data: SlotData
11 var external_inventory_owner
12
13 @onready var player_inventory: PanelContainer = $PlayerInventory
14 @onready var grabbed_slot: PanelContainer = $GrabbedSlot
15 @onready var external_inventory: PanelContainer =
16     $ExternalInventory
17
18 func _physics_process(delta:float) -> void:
19     if grabbed_slot.visible:
20         grabbed_slot.global_position =
21             get_global_mouse_position() + Vector2(5,5)
22
23     if external_inventory_owner \
24         and external_inventory_owner.
25             global_position.distance_to(CharacterManager.
26                 get_global_position()) > 4:
27         force_close.emit()
28
29 func set_player_inventory_data(inventory_data: InventoryData) ->
30     void:
31     inventory_data.inventory_interact.
32     connect(on_inventory_interact)
33     player_inventory.set_inventory_data(inventory_data)
34
35 #inventory of the interactable
36 func set_external_inventory(_external_inventory_owner) -> void:
37     #print(_external_inventory_owner)
38     external_inventory_owner = _external_inventory_owner
39     var inventory_data = external_inventory_owner.inventory_data
40
41     inventory_data.inventory_interact.
42     connect(on_inventory_interact)
43     external_inventory.set_inventory_data(inventory_data)
```

```
44     external_inventory.show()
45
46 #to disconnect and clear the external_inventory
47 func clear_external_inventory() -> void:
48
49     if external_inventory_owner:
50         var inventory_data =
51             external_inventory_owner.inventory_data
52
53         inventory_data.inventory_interact.
54             disconnect(on_inventory_interact)
55         external_inventory.clear_inventory_data(inventory_data)
56
57         external_inventory.hide()
58
59         external_inventory_owner = null
60
61
62 func on_inventory_interact(inventory_data: InventoryData,
63                             index:int, button:int) -> void:
64     # check if interact with slots
65     #print("%s %s %s" % [inventory_data, index, button])
66
67     #no slot_data grabbed
68     match [grabbed_slot_data, button]:
69         [null, MOUSE_BUTTON_LEFT]:
70             grabbed_slot_data = inventory_data.
71             grab_slot_data(index)
72             # _ can be anything, if we do have something in the
73             # slot_data
74         [_, MOUSE_BUTTON_LEFT]:
75             grabbed_slot_data =
76                 inventory_data.drop_slot_data(grabbed_slot_data,
77                                             index)
78
79             #case for consuming an consumable item_data
80             [null, MOUSE_BUTTON_RIGHT]:
81                 inventory_data.use_slot_data(index)
82                 # _ can be anything, if we do have something in the
83                 # slot_data
84                 # single drop of item with right click
85             [_, MOUSE_BUTTON_RIGHT]:
86                 grabbed_slot_data = inventory_data.
87                 drop_single_slot_data(grabbed_slot_data, index)
88
89             #print(grabbed_slot_data)
```

```

85         update_grabbed_slot()

86
87     #update display
88 func update_grabbed_slot() -> void:
89     if grabbed_slot_data:
90         grabbed_slot.show()
91         grabbed_slot.set_slot_data(grabbed_slot_data)
92     else:
93         grabbed_slot.hide()

94
95
96 func _on_gui_input(event: InputEvent) -> void:
97     if event is InputEventMouseButton \
98         and event.is_pressed() \
99         and grabbed_slot_data:
100
101     match event.button_index:
102         MOUSE_BUTTON_LEFT:
103             drop_slot_data.emit(grabbed_slot_data)
104             #print("drop data")
105             grabbed_slot_data = null
106             # drop single slot_data by right clicking
107         MOUSE_BUTTON_RIGHT:
108             drop_slot_data.emit(grabbed_slot_data.
109             create_single_slot_data())
110             if grabbed_slot_data.quantity < 1:
111                 grabbed_slot_data = null
112             update_grabbed_slot()

113
114
115 #drop grabbed slot_data if inventory closed
116 func _on_visibility_changed() -> void:
117     if not visible and grabbed_slot_data:
118         drop_slot_data.emit(grabbed_slot_data)
119         grabbed_slot_data = null
120         update_grabbed_slot()

```

Im inventory_interface.gd Skript werden die Funktionalitäten von der Oberfläche des Inventarsystems des Farming Simulators implementiert.

Die Klasse InventoryPanel erweitert das Control Element und steuert das Inventarpanel.

Die Funktion _physics_process wird im jedem Frame aufgerufen und aktualisiert die Position des ausgewählten Slots. Außerdem wird auch auf die Entfernung des externen Inventars geschaut, um diese zu schließen, wenn der Charakter weiter weg läuft.

Die set_external_inventory Funktion setzt die Daten des externen Inventars und verbindet sie mit den benötigten Signalen. Das set_player_inventory_data macht das gleiche wie die vorher genannte Funktion, nur das es hier um das Charakter Inventar handelt.

Die Funktion clear_external_inventory entfernt die Daten von den externen Inventar und trennt die Verbindung zu den Signalen.

Die on_inventory_interact Funktion wird aufgerufen, wenn mit dem Inventar interagiert wird, wie das ablegen, aufheben oder verbrauchen der Slot Daten. Hier wird auf die Inventar Daten zugegriffen.

Die Funktion update_grabbed_slot updated die Darstellung des aktuell angeklickten Slots in der Benutzeroberfläche.

Die _on_gui_input Funktion führt die Eingaben des Nutzers in der Benutzeroberfläche, wie zum Beispiel das Ablegen eines Objektes.

Die Funktion _on_visibility_changed wird aufgerufen, wenn die Sichtbarkeit des Inventars verändert wird, wie das schließen und das öffnen des Inventars. Hier wurde auch eingebaut, dass ein geöffnetes Objekt, nach dem Schließen des Inventars fallen gelassen wird.

List. 3.5: hotbar_inventory.gd

```
1 extends PanelContainer
2
3 #use hotbar
4 signal hot_bar_use(index: int)
5
6 const Slot = preload("res://inventory/slot.tscn")
7
8 @onready var h_box_container: HBoxContainer =
9     $MarginContainer/HBoxContainer
10
11 func _unhandled_key_input(event: InputEvent) -> void:
12     if not visible or not event.is_pressed():
13         return
14
15     #return index, emitting 0 through 5
16     if range(KEY_1, KEY_7).has(event.keycode):
17         hot_bar_use.emit(event.keycode - KEY_1)
18
19 # set up connection to the inventory_data
20 func set_inventory_data(inventory_data: InventoryData) -> void:
21     inventory_data.inventory_updated.connect(populate_hot_bar)
22     populate_hot_bar(inventory_data)
23     hot_bar_use.connect(inventory_data.use_slot_data)
24
25 #hotbar update
26 func populate_hot_bar(inventory_data: InventoryData) -> void:
27     #clear the item grid of any children
28     for child in h_box_container.get_children():
29         child.queue_free()
30
31     # pass only 6 elements
32     for slot_data in inventory_data.slot_datas.slice(0, 6):
33         var slot = Slot.instantiate()
34         h_box_container.add_child(slot)
35
36         # check slot data and update parameter
37         if slot_data:
38             slot.set_slot_data(slot_data)
```

Das Skript hotbar_inventory.gd implementiert die Hotbar ins Inventarsystem des Farming Simulators.

Die Klasse HotbarPanel erweitert das PanelContainer Element und steuert die Hotbar.

Die Funktion _unhandled_key_input wird aktiviert, wenn ein Ereignis nicht ausgeführt wird. Überprüft wird, die Sichtbarkeit des Hotbars und ob die Tasteneingaben von Index 0-5 ausgelöst werden. Wenn diese ausgelöst werden, wird ein Signal hot_bar_use mit dem jeweiligen Index ausgeführt.

Die set_inventory_data Funktion setzt die Daten und verbindet die jeweiligen benutzten Signale, wie das Aktualisieren des Inventars und die Verbindung zu der Inventar Oberfläche.

Die Funktion populate_hot_bar updated die Hotbar, wie das Füllen der Slots mit den Inventar Daten. Zuerst werden die Kinder des Containers entfernt und dann werden die Slots hinzugefügt. Hier wird geschaut, dass nur null bis sechs Slot Daten erstellt werden.

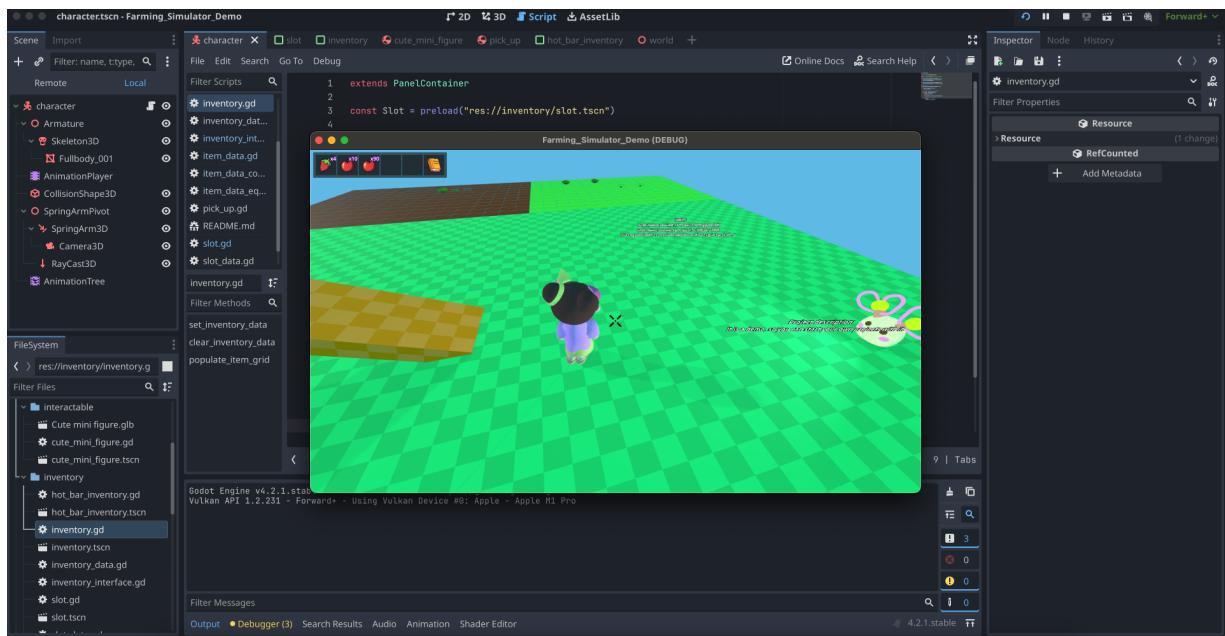


Abb. 3.31: Hotbar Inventar

In der Abbildung 3.31 ist die Hotbar zu sehen und die jeweiligen Slot Daten von Index null bis fünf. Man kann die hotbar mit Tab Aktion öffnen und schließen. Ist im Inventar von Index null bis fünf ein verbrauchbarer Gegenstand, kann man diese durch das Auslösen der Tasten eins bis sechs verwenden.

List. 3.6: slot.gd

```

1  extends PanelContainer
2
3  signal slot_clicked(index:int, button:int)
4
5  @onready var texture_rect: TextureRect = $MarginContainer/TextureRect
6  @onready var quantity_label: Label = $QuantityLabel
7
8
9  func set_slot_data(slot_data: SlotData) -> void:
10    #get reference
11    var item_data = slot_data.item_data
12    #update texture
13    texture_rect.texture= item_data.texture
14    #set tooltip text
15    tooltip_text = "%s\n%s" % [item_data.name, item_data.description]
16
17    #show or hide quantity label
18    if slot_data.quantity > 1:
19      quantity_label.text = "x%s" % slot_data.quantity
20      quantity_label.show()
21    #hide quantity label none stackable item
22    else:
23      quantity_label.hide()
24
25
26
27  func _on_gui_input(event):
28    #check if right or left click and if it is pressed
29    if event is InputEventMouseButton \
30        and (event.button_index == MOUSE_BUTTON_LEFT \
31        or event.button_index == MOUSE_BUTTON_RIGHT) \
32        and event.is_pressed():
33      slot_clicked.emit(get_index(), event.button_index)

```

Das Skript slot.gd stellt die Logik des jeweiligen Slots im Inventar dar.

Die Klasse Slot erweitert das PanelContainer Element und steuert die Slots im Inventar.

Die Funktion set_slot_data setzt die Daten von einem Slot und aktualisiert diese auch in der Oberfläche. Außerdem hat sie auch die Slot Daten als Parameter, mit der Menge und einem Tooltip als Text. Geschaut wird, ob ein Item stapelbar ist oder nicht, dementsprechend wird bestimmt, ob die Menge ein- oder ausgeblendet.

Die _on_gui_input Funktion wird aktiviert, wenn mit der Maus geklickt wird. Diese Funktion prüft auch, um welche Maustaste es sich handelt und sendet das Signal

slot_clicked mit der ausgelösten Maustaste und des Slots Indexes.

List. 3.7: slot_data.gd

```

1  extends Resource
2  class_name SlotData
3
4  const MAX_STACK_SIZE: int = 99
5
6  @export var item_data: ItemData
7  @export_range(1, MAX_STACK_SIZE) var quantity: int = 1: set =
8      set_quantity
9
10 #check if it can add one item
11 func can_merge_with(other_slot_data: SlotData) -> bool:
12     return item_data == other_slot_data.item_data \
13         and item_data.stackable \
14         and quantity < MAX_STACK_SIZE
15
16 #can merge stackable data: look if exceeded MAX_STACK_SIZE and
17 #not merge just swap
18 func can_fully_merge_with(other_slot_data: SlotData) -> bool:
19     return item_data == other_slot_data.item_data \
20         and item_data.stackable \
21         and quantity + other_slot_data.quantity <=
22             MAX_STACK_SIZE
23
24 # merge with stackable data
25 func fully_merge_with(other_slot_data: SlotData) -> void:
26     quantity += other_slot_data.quantity
27
28 #create single slot data to drop
29 func create_single_slot_data() -> SlotData:
30     var new_slot_data = duplicate()
31     new_slot_data.quantity = 1
32     quantity -= 1
33     return new_slot_data
34
35 func set_quantity(value: int) -> void:
36     quantity = value
37     #check quantity if stackable
38     if quantity > 1 and not item_data.stackable:
39         quantity = 1
40         push_error("%s_is_not_stackable, setting_quantity_to_1" %
41             item_data.name)

```

Das Skript slot_data definiert die Datenstruktur der Slot Daten im Inventar.

Die Klasse SlotData definiert die Daten des Inventarslots. Diese enthält die Referenz des item_data und die Menge von den Slot Objekten. Die maximale Stackgröße wird

hier auch bestimmt.

Die Methode can_merge_with überprüft das zusammenführen von zwei Slot Daten.

Die Funktion can_fully_merge_with stellt fest ob ein anderer Slot vollständig zusammengeführt werden kann, ohne das die Maximale Stackgröße erreicht wird.

Die fully_merge_with Methode führt die Slot Daten eines anderen Slots zusammen.

Die Methode create_single_slot_data erzeugt eine Kopie von den Slot Daten, wenn nur ein Objekt aus dem Stapel entfernt wird.

Die Funktion set_quantity setzt die Menge der Slot Daten und prüft ob es stapelbar ist.

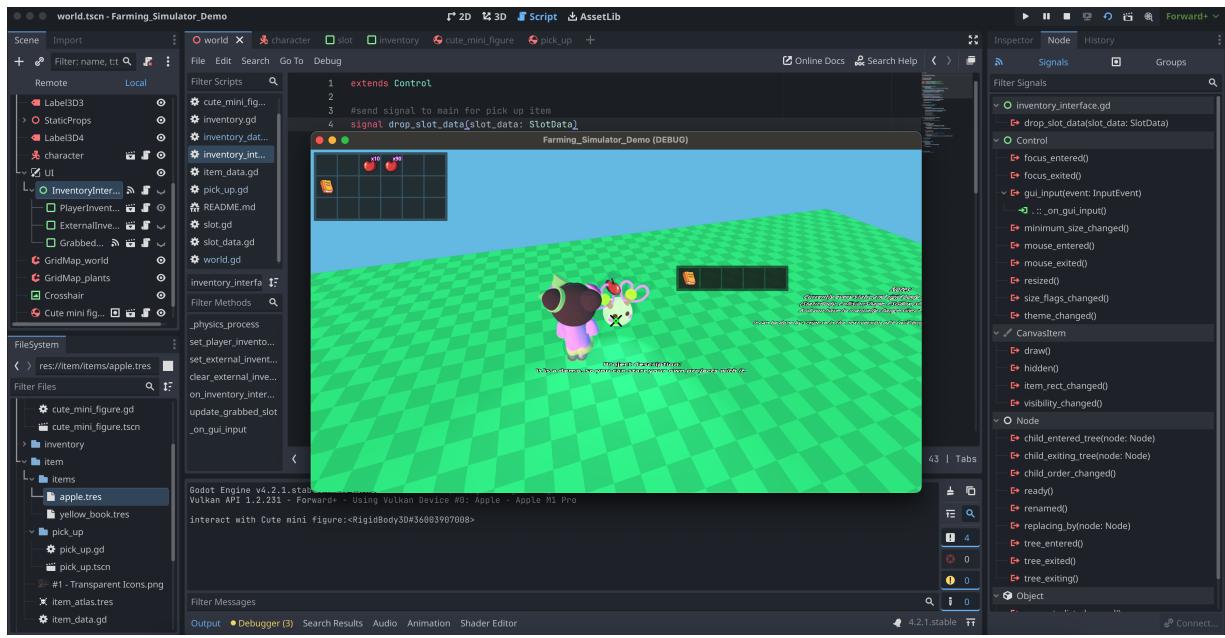


Abb. 3.32: Slot Data Beispiel: Quantität

In der Abbildung 3.32 sieht man, wie der Charakter mit einem interagierbaren Gegenstand interagiert. Zu sehen ist das Inventar vom Charakter und des externen Inventars. Außerdem erkennt man auch die Menge der Slot Daten im Inventar. Es ist möglich die Objekte von einem Inventar zum anderen Inventar hin und her zu schieben.

List. 3.8: pickup_item.gd

```
1  extends RigidBody3D
2
3  @export var slot_data: SlotData
4
5  @onready var sprite_3d: Sprite3D = $Sprite3D
6
7  func _ready() -> void:
8      sprite_3d.texture = slot_data.item_data.texture
9
10
11 func _physics_process(delta: float) -> void:
12     #rotate a little
13     sprite_3d.rotate_y(delta)
14
15
16 func _on_area_3d_body_entered(body: Node3D) -> void:
17     #print("body entered")
18     if body.inventory_data.pick_up_slot_data(slot_data):
19         queue_free()
```

Das Skript pick_up.gd stellt ein Gegenstand und dessen Verhalten als 3D-Objekt im Spiel dar.

Die Klasse ItemObject erweitert das RigidBody3D Node.

In der _ready Funktion wird das 3D-Sprite des Items mit der Textur des Objektes geupdated. Das Aussehen wird mit der Textur dargestellt.

Die Funktion _physics_process wird für jedes Frame ausgeführt und lässt den Gegenstand langsam rotieren.

Die _on_area_3d_body_entered Funktion wird aufgerufen, wenn der Charakter in das RigidBody3D eintritt. Wenn der Charakter das Objekt aufnehmen kann wird das Objekt entfernt und im Inventar des Charakters aufgenommen.

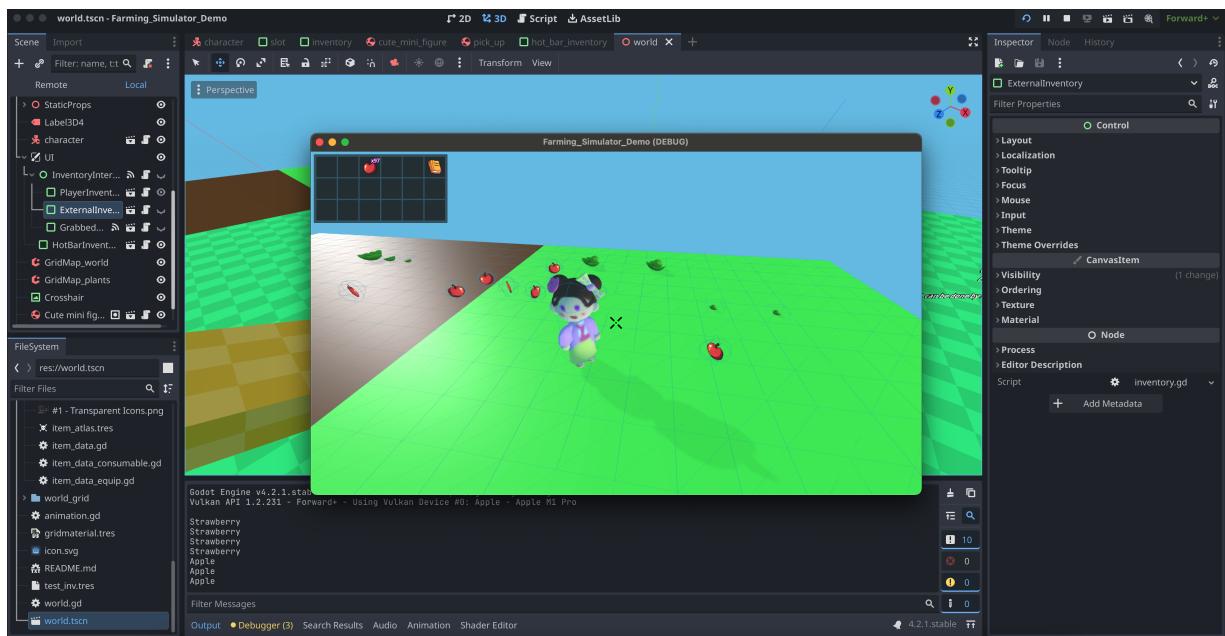


Abb. 3.33: PickUp Items

In der Abbildung 3.33 ist die PickUp Funktion dargestellt. Auf den Boden wurden die Gegenstände herausgeworfen und können, durch das eintreten des Gegenstandes aufgesammelt werden.

List. 3.9: item_data.gd

```
1  extends Resource
2  class_name ItemData
3
4  @export var name: String = ""
5  @export_multiline var description: String = ""
6  @export var stackable: bool = false
7  @export var texture: AtlasTexture
8
9  func use(target) -> void:
10    pass
```

Das Skript item_data.gd stellt die Datenstruktur von Item Daten dar.

Die Klasse ItemData erweitert das Ressource Element, heißt sie kann geladen und gespeichert werden. Die Variablen, welche exportiert wurden enthalten den Namen des Items, die Beschreibung eines Items und einen bool Wert um zu bestimmen, ob das Objekt stapelbar ist oder nicht. Außerdem wird die Textur des Objekts als AtlasTextur angegeben.

Die Funktion use(target) ist erstellt worden, um das Objekt zu verwenden. Das pass steht, für keine Funktion vorhanden.

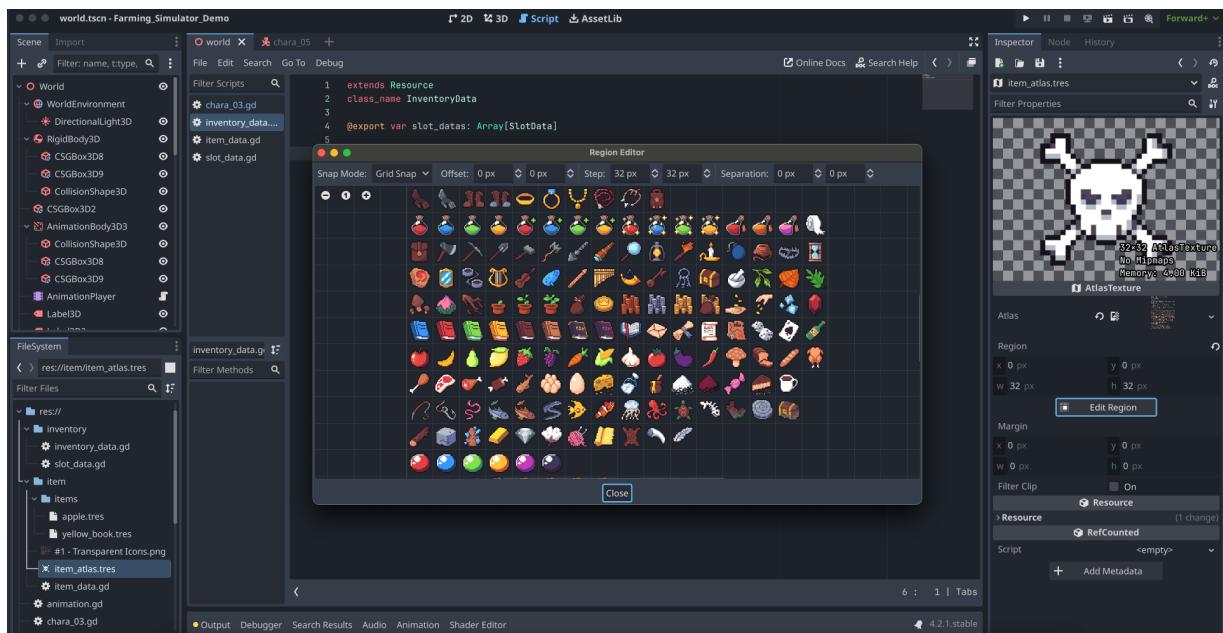


Abb. 3.34: Item Data Atlas Texture

In der Abbildung 3.34 sieht man eine Beispiel AtlasTexture, welches als Prototyp verwendet wurde. Später wurden dann selber erstellte Items als AtlasTextur verwendet. Den Bereich einer Textur kann man selber festlegen, um später ein ItemData zu erstellen. Hier wurde der Bereich 32x32 Pixel, für ein Item ausgewählt. Die hier benutzen Texturen sind frei zugänglich und können auf der Seite <https://shikashipx.itch.io/shikashis-fantasy-icons-pack> heruntergeladen werden.

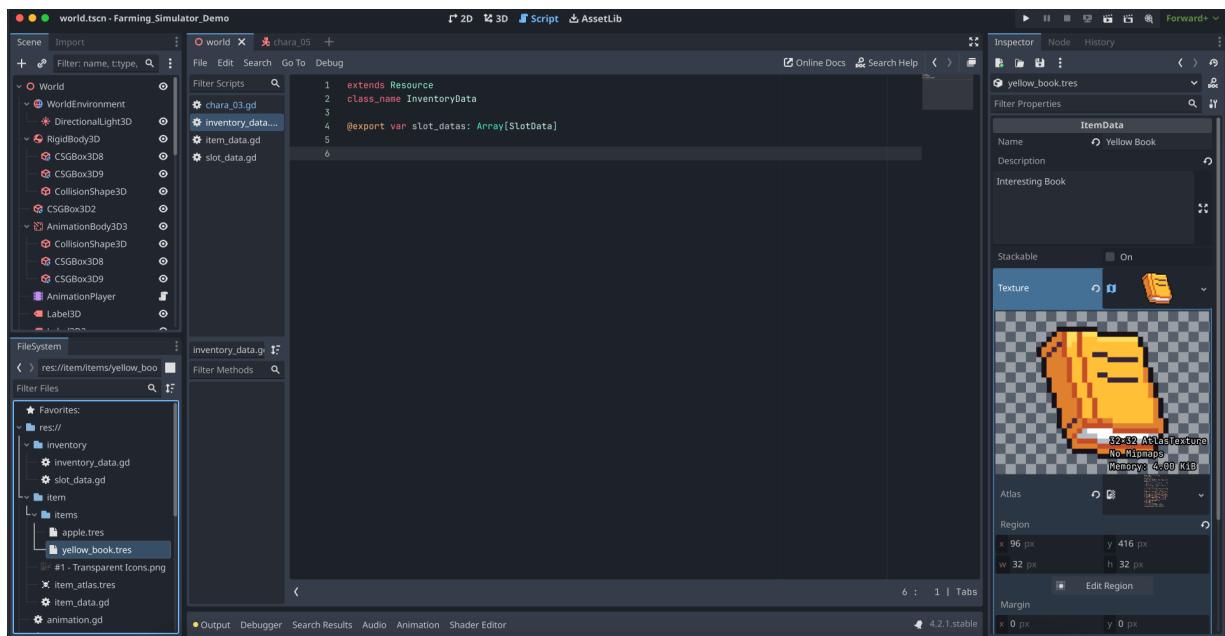


Abb. 3.35: Item Data: Yellow Book

In der Abbildung 3.35 sieht man wie so ein ItemData erstellt wird. Man kann den Namen, die Beschreibung und die Stapelbarkeit des Items festlegen. In Edit Region kann man den bestimmten Bereich wählen. Da man ein yellow_book.tres Datei hat und ein gelbes Buch haben möchte, wählt man den Bereich, wo sich das gelbe Buch befindet, im AtlasTexture aus. Wichtig ist auch das man die ItemData als Make Unique kennzeichnet. Die erstellten item.tres Dateien kann man später in die Slot Daten der Inventory-Data reinziehen und benutzen.

List. 3.10: item_data_consumable.gd

```
1  extends ItemData
2
3  class_name ItemDataConsumable
4
5  @export var heal_value: int
6
7  func use(target) -> void:
8      #heal the player
9      if heal_value != 0:
10          target.heal(heal_value)
```

Das Skript item_data_consumable.gd wurde für die verbrauchbaren Items erstellt.

Die ItemDataConsumable Variable erweitert die ItemData Klasse.

Die Funktion use(target) wird aufgerufen, wenn ein verbrauchbarer Gegenstand verwendet wird. Wird ein Item konsumiert, wird die Funktion heal aufgerufen und der healvalue wird gesteigert.

List. 3.11: character.gd

```
1  extends CharacterBody3D
2
3  @export var inventory_data: InventoryData
4
5  @onready var armature = $Armature
6  @onready var spring_arm_pivot = $SpringArmPivot
7  @onready var spring_arm = $SpringArmPivot/SpringArm3D
8  @onready var anim_tree = $AnimationTree
9  @onready var ray = $SpringArmPivot/RayCast3D
10 @onready var camera = $SpringArmPivot/SpringArm3D/Camera3D
11
12 const SPEED = 5.0
13 const LERP_VAL = .15
14 const JUMP_VAL = 5.0
15 const RAY_DISTANCE = 4
16
17 signal toggle_inventory()
18
19 # Get the gravity from the project settings to be synced with
   RigidBody nodes.
20 var gravity =
21     ProjectSettings.get_setting("physics/3d/default_gravity")
22 var health: int = 5
23
24 func _ready():
25     CharacterManager.character = self
26     Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
27
28 func _unhandled_input(event):
29     if Input.is_action_just_pressed("quit"):
30         get_tree().quit()
31
32     if event is InputEventMouseMotion:
33         spring_arm_pivot.rotate_y(-event.relative.x * .005)
34         spring_arm.rotate_x(-event.relative.y * .005)
35         spring_arm.rotation.x = clamp(spring_arm.rotation.x,
36                                         -PI/4, PI/4)
37
38         # Ray movement with mouse
39         ray.rotate_x(-event.relative.y * .005)
40         ray.rotation.x = clamp(ray.rotation.x, PI/4, PI/2)
41
42         # Character movement with camera
43         armature.rotate_y(-event.relative.x * .005)
```

```

43
44     if Input.is_action_just_pressed("inventory"):
45         toggle_inventory.emit()
46
47     if Input.is_action_just_pressed("interact"):
48         interact()
49
50     func _physics_process(delta):
51         # Add the gravity.
52         if not is_on_floor():
53             velocity.y -= gravity * delta
54
55         # Get the input direction and handle the
56         # movement/deceleration.
57         # As good practice, you should replace UI actions with
58         # custom gameplay actions.
59         var input_dir = Input.get_vector("left", "right", "forward",
60                                         "back")
61         var direction = (transform.basis * Vector3(input_dir.x, 0,
62                                                 input_dir.y)).normalized()
63         direction = direction.rotated(Vector3.UP,
64                                       spring_arm_pivot.rotation.y)
65         if direction:
66             velocity.x = lerp(velocity.x, direction.x * SPEED,
67                               LERP_VAL)
68             velocity.z = lerp(velocity.z, direction.z * SPEED,
69                               LERP_VAL)
70             armature.rotation.y = lerp_angle(armature.rotation.y,
71                                              atan2(-velocity.x, -velocity.z), LERP_VAL)
71         else:
72             velocity.x = lerp(velocity.x, 0.0, LERP_VAL)
73             velocity.z = lerp(velocity.z, 0.0, LERP_VAL)
74
75         if Input.is_action_pressed("jump"):
76             if is_on_floor():
77                 velocity.y = JUMP_VAL
78
79             anim_tree.set("parameters/BlendSpace1D/blend_position",
80                         Vector2(velocity.x, velocity.z).length() / SPEED)
81
82             move_and_slide()
83
84     func interact() -> void:
85         if ray.is_colliding():
86             print("interact_with_", ray.get.collider())
87             ray.get.collider().player_interact()
88

```

```
81
82     #drop location in front of character (using ray position)
83     func get_drop_position() -> Vector3:
84         var direction = -ray.global_transform.basis.z
85         return ray.global_position + direction
86
87     func heal(heal_value: int) -> void:
88         health += heal_value
```

Das character.gd Skript wurde erweitert auf ein RayCast3D, um mit Objekten zu interagieren.

Die Funktion interact() wird für das interagieren mit der Umgebung verwendet.

Die get_drop_position Methode gibt die Position des RayCasts des Characters von vorne zurück und bestimmt, wo das Objekt was man rauswirft, fallen gelassen wird.

Die heal() Methode heilt den Charakter um den Wert des heal_value Parameters.

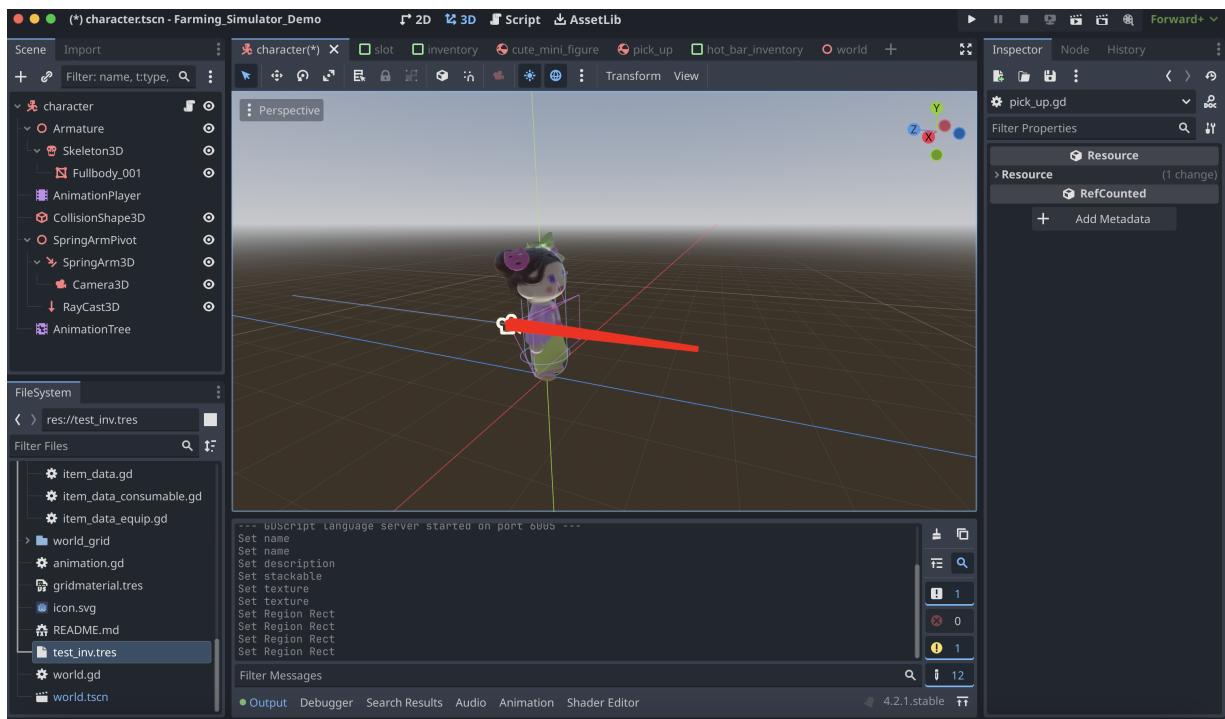


Abb. 3.36: Charakter mit RayCast3D

In der Abbildung 3.36 ist der Charakter mit dem RayCast3D abgebildet. Bei dem erstellen des Rays, wird auch ein Crosshair Element erstellt, um die Postion des Rays im Spiel anzeigen zu können.

List. 3.12: character_manager.gd

```
1  extends Node
2
3  var character
4
5  func use_slot_data(slot_data: SlotData) -> void:
6      #pass character as an argument
7      slot_data.item_data.use(character)
8
9  func get_global_position() -> Vector3:
10     return character.global_position
```

Das character_manager.gd Skript erbt von der Klasse Node.

Die Funktion use_slot_data(slot_data:SlotData) wird aufgerufen, wenn das Inventar verwendet wird. Sie ruft die use() Methode des ItemData im SlotData auf und über gibt charakter als Argument.

Die get_global_position Funktion liefert die globale Position des Charakters zurück, welches mit dem Inventar verbunden ist.

List. 3.13: cute_mini_figure.gd

```
1  extends RigidBody3D
2
3  #interact with inventory
4  signal toggle_inventory(external_inventory_owner)
5
6  @export var inventory_data: InventoryData
7
8  #throw the signal when interact
9  func player_interact() -> void:
10     toggle_inventory.emit(self)
```

Das Skript cute_mini_figure.gd erweitert die Klasse RigidBody3D.

Das Signal `toggle_inventory(external_inventory_owner)` wird für das öffnen und schließen des Inventars verwendet.

Die Funktion `player_interact` wird beim Interagieren mit einem Objekt verwendet. Diese sendet dann ein Signal um das Inventar zu öffnen oder zu schließen und gibt sich selbst als Argument wieder.

Kapitel 4

Ergebnis und Ausblick

In diesem Kapitel wird zusammengefasst, was erreicht wurde, welche Schwierigkeiten aufgetreten sind und was noch für die nähere Zukunft implementiert werden könnte.



Abb. 4.1: Farming Simulator

Das Ziel dieses Projektes war ein Farming Simulator, mit einer zeitgemäßen Game Engine (Godot) zu entwickeln. Man kann mehrere positive Errungenschaften in der Entwicklung des Farming SImulators auflisten. Es ist möglich, Samen in einen bestimmten Bereich einzupflanzen und nach den vollständigen Wachstum aufzusammeln. Gelungen ist auch das Inventarsystem, welches verschiedene Slot Daten enthält und eine Mengenübersicht liefert. Außerdem kann man mit einem externen Inventarsystem interagieren. Was aber nicht gelungen ist, sind die Werkzeuge einzubauen, wie eine Hacke um selber braune Felder zu produzieren. Es können gerade nur bestimmte braune

Stellen zum einpflanzen benutzt werden. Erreicht wurde auch ein Charakter zu erstellen. Dieser kann im Spiel laufen, springen und mit Objekten interagieren. Leider konnte man das Gesicht aus zeitlichen Gründen nicht animieren. Das Haus sollte eigentlich als Speicher Zone dienen, welches aber aus zeitlichen Gründen auch nicht erfüllt wurde.

In der näheren Zukunft könnte man, die vorgenommenen Features implementieren. Man könnte auch ein Verkaufsstand mit einem Nicht-Spieler-Charakter erstellen um weitere Samen oder Werkzeuge zu kaufen. Eine Health Bar wäre auch eine gute Erweiterung, um das konsumieren eines Gemüses oder Obstes ein Sinn zu geben. Ein Hauptmenü konnte auch aus zeitgründen und mangelnde Erfahrung nicht umgesetzt werden und könnte in der Zukunft umgesetzt werden. Eine gute Idee wäre auch ein Charakter Editor, um mehrere Charaktere auswählen zu können und sie auf den Geschmack des Benutzers anpassen zu können. Außerdem könnte man auch eine Hintergrundmusik einbauen, damit es eine schöne und beruhigende Stimmung gibt. Des weiteren könnte man, weitere Nicht-Spieler-Charaktere, als Bewohner von einer Stadt erstellen. Dabei kommt man zu den Punkt die Map zu vergrößern und weitere Häuser, Bäume und andere Elemente einzubauen. Man könnte auch das Spiel zu einem Multi-Player-Modus erweitern um mit Freunden zusammen zu spielen.

Wie in Kapitel 3.1.1 ist vieles nicht möglich, weil die Lernkurve sehr schleifend ist. Viele Godot Features kann man nur durch viel Erfahrung benutzen. Außerdem ist für die Entwicklung des Spiels zwei Entwickler nicht ausreichend, also sollte das Team erweitert werden. Für den Anfang ist das Projekt Farming Simulator gut gelungen und man kann die Basis Farming Aktionen ausführen. Insgesamt kann man den Farming Simulator Prototyp als gelungen einstufen.

Literatur

1. JOHNSON, J. *Godot 4 Game Development Cookbook: Over 50 solid recipes for building high-quality 2D and 3D games with improved performance*. Packt Publishing, 2023. ISBN 9781838827250. Auch verfügbar unter: <https://books.google.de/books?id=FNS-EAAAQBAJ>.
2. *Godotengine download macOs*. [o. D.]. Auch verfügbar unter: <https://godotengine.org/download/macos/>. Zugegriffen: 2023-25-12.
3. BRADFIELD, C. *Godot 4 Game Development Projects: Build five cross-platform 2D and 3D games using one of the most powerful open source game engines*. Packt Publishing, 2023. ISBN 9781804615621. Auch verfügbar unter: <https://books.google.de/books?id=GrfLEAAAQBAJ>.
4. *Godot Docs – 4.2 branch — docs.godotengine.org* [<https://docs.godotengine.org/en/stable/index.html>]. [o. D.]. [Accessed 30-12-2023].
5. *AnimalCrossing* [<https://i.pinimg.com/originals/07/f7/ec/07f7ec705585cd7a9c.jpg>]. [o. D.]. [Accessed 14-01-2024].
6. SAVIOR. *Inventory profiles Mod (1.20.4, 1.19.4) - inventory sorting tweaks*. 2023. Auch verfügbar unter: <https://www.9minecraft.net/inventory-profiles-mod/>. [Accessed 01-02-2024].
7. *Getting Started | Nomad* — nomadsculpt.com/manual/gettingstarted. [o. D.]. [Accessed 16-01-2024].
8. *Mixamo Startpage*. [o. D.]. Auch verfügbar unter: <https://www.mixamo.com/#/>. [Accessed 01-02-2024].