

Chroma Core

Dillon Metzler, Mat.Nr.: 431015



Hochschule Kempten
University of Applied Sciences

Projektdokumentation
Semester: SS 23

Entstanden im Rahmen der
Veranstaltung **Game-Prototyping**
an der Hochschule Kempten
bei Prof. Dr.-Ing. René Bühling

Inhaltsverzeichnis

1. Überblick	1
2. Making of Best-Of.....	2
2.1. Spieler Controller.....	2
2.1.1. Konzeption.....	3
2.1.2. Implementierung.....	5
2.2. Enterhaken	7
2.2.1. Konzeption.....	7
2.2.2. Implementierung.....	8
2.3. Sonstige Tätigkeiten	9
2.4. Probleme	9
3. Abschluss	10
4. Quellenverzeichnis	11
4.1. Bild- & Videoverweis	11

1. Überblick

Im Rahmen unseres Projekts Chroma Core habe ich mich mit der Programmierung der Kern-Mechaniken für die Steuerung des Spielercharakters und die Implementierung der Animationen für diesen beschäftigt. Das Projekt wurde mit der Godot-Engine Version 4.0 entwickelt. Dabei wurde die Programmiersprache C# verwendet. Da ich mit der Godot-Engine bisher keine Erfahrung gemacht habe, dafür aber viel Erfahrung in C# besitze, entschied ich mich im Zusammenhang unseres Projekts dazu, diese Aufgaben zu übernehmen. Dabei habe ich mich intensiv mit der Engine auseinandergesetzt und dadurch viele Erfahrungen bei der Verwendung der Godot-Engine zur Entwicklung eines 2D Plattformers sammeln und ebenso meine Fähigkeiten in C# verbessern können.

Programmierung:

- Bewegung des Spielers
 - Nach links/rechts bewegen
 - Springen
- Passive Bewegungsmechaniken
 - Variable Sprung Höhe
 - Apex-Modifikator
 - Sprung-Pufferung
 - Coyote-Zeit
 - Beschränkte Fallgeschwindigkeit
 - Beschleunigung und Abbremsung
→ Reibung
- Aktive Bewegungsmechaniken
 - Dash
 - Enterhaken zum Schwingen
- Interaktionen mit der Spielwelt und anderen Objekten
- Kamera mit Verzögerung

Implementierung der Animationen des Spielcharakters:

- Idle
- Schieben
- Laufen
- Schwingen
- Springen
- Sterben
- Fallen
- Dashen

2. Making of Best-Of

2.1. Spieler Controller

Bevor ich mit der eigentlichen Konzeption des Spieler-Controllers angefangen habe, setzte ich mich erst einmal intensiv mit der Godot-Engine auseinander, um herauszufinden, welche Möglichkeiten die Engine bietet, ein 2D Spiel zu entwickeln. Dabei gibt es im Wesentlichen zwei Optionen, einen Charakter-Controller zu implementieren. Ersteres wäre ein Controller basierend auf der Physik-Engine von Godot zu entwickeln. Dabei nutze man die RigidBody2D Node, welche bereits jegliche simulierte Physik, die gefordert ist, wie zum Beispiel Gravitation, implementiert. Dabei wendet man Kräfte auf ein Objekt an, welche die Transformationskomponenten beeinflussen. Kollisionen mit anderen Objekten werden von der Engine entsprechend berechnet und simuliert. Zwar funktioniert dies recht gut, doch stellte ich fest, dass man dabei nicht die gewünschte Kontrolle über das Objekt hat. Gerade bei einem Plattformer ist eine präzise Bewegung essenziell wichtig, da dies eines der Kernelemente ist. Schließlich habe ich mich für die zweite Option entschieden. Diese basiert auf die CharacterBody2D Node, welche speziell darauf ausgelegt ist, eigene Bewegungsmechaniken zu implementieren. Ebenso implementiert diese bereits viele Helfermethoden, die bei der Entwicklung eines Plattformers hilfreich sind, wie zum Beispiel die Erkennung, ob ein Objekt auf dem Boden oder in der Lust ist. Im Gegensatz zu der ersten Option hat dabei ein Objekt eine Geschwindigkeit (Velocity), über welche man die Bewegung steuert, und wird auf keiner Weise von der Physik-Engine beeinflusst. Bei der Geschwindigkeit handelt es sich intern um einen Vector2D (X/Y). Da wir uns in einem 2-Dimensionalen Koordinatensystem befinden, sorgt ein positiver x-Wert für eine Bewegung nach rechts und ein positiver y-Wert für eine Bewegung nach unten. Man gibt einem Objekt eine Geschwindigkeit und ruft dann entweder die Methode MoveAndCollide oder MoveAndSlide auf. Folglich bewegt sich das Objekt konstant mit der gesetzten Geschwindigkeit. Dabei bestimmen die Methoden, wie sich das Objekt bei einer Kollision mit einem anderen verhält. MoveAndCollide sorgt dafür, dass das Objekt umgehend gestoppt wird, sobald es mit einem anderen Objekt kollidiert. Bei MoveAndSlide hingegen gleitet das Objekt entlang des anderen Objektes, anstatt sofort zu stoppen. Da MoveAndSlide für eine weichere bzw. glaubwürdigere Kollision sorgt, habe ich mich dazu entschieden, diese Methode zu verwenden. Dies ermöglicht ebenso, dass man per Code selbst entscheiden kann, was bei einer Kollision geschehen soll. Beispielsweise könnte das Objekt bei einer Kollision von dem anderen abprallen und sich in eine andere Richtung bewegen. Das bedeutet zwar, dass die komplette Physik selbst programmiert werden muss, doch hat man dabei die größtmögliche Freiheit ein Objekt zu manipulieren und kann jegliche Bewegungsmechaniken umsetzen, ohne dabei eventuell von der Physik-Engine eingeschränkt zu werden.

2.1.1. Konzeption

Nachdem ich gewusst hatte, wie die Bewegungsmechaniken umgesetzt werden konnten, machte ich mich zunächst daran, ein Konzept zu entwickeln, um mir die Implementierung zu erleichtern.

Nach Absprache mit Stefan, welcher sich unter anderem um das Projektmanagement gekümmert hat und somit auch um die Aufgabenverteilung, einigten wir uns darauf, dass der Spieler-Controller als State-Pattern umgesetzt werden soll. Da der Spieler-Controller voraussichtlich recht komplex ausfallen sollte, konnte mithilfe des verwendeten Patterns eine gute Übersichtlichkeit und problemlose Erweiterbarkeit garantiert werden. Abgesehen von einer Liste mit Anforderungen, mit Mechaniken des PlayerControllers, hatte ich bei der Umsetzung keine Einschränkungen und somit die volle Verantwortung.

Der Spieler kann sich nach links oder rechts bewegen sowie springen.

Dabei gibt es einige passive Plattformer Mechaniken, die für ein besseres Spielgefühl sorgen.

Variable Sprung Höhe:

Hält der Spieler die Sprungtaste während des Springens gedrückt, so springt der Spieler maximal hoch. Wird die Sprungtaste vorher losgelassen, endet der Sprung vorher.

Folglich erreicht der Spieler nicht die maximale Sprunghöhe.

Apex-Modifikator:

Am höchsten Punkt des Sprungs erfährt der Spieler eine gewisse Zeit, keine Gravitation und bekommt einen Geschwindigkeits-Boost abhängig von seiner Bewegungsrichtung.

Sprung Pufferung:

Betätigt der Spieler die Sprungtaste während einem Sprung oder während dem Fallen nach einem Sprung, springt der Spieler, sobald er den Boden berührt, erneut.

Coyote-Zeit:

Zeitfenster, in dem der Spieler nach dem Verlassen einer Plattform, noch in der Lage ist zu springen.

Beschränkte Fallgeschwindigkeit:

Die Fallgeschwindigkeit kann einen bestimmten Wert nicht überschreiten.

Beschleunigung und Abbremsung des Spielers:

Der Spieler stoppt nicht abrupt, sondern erfährt Reibung.

Des Weiteren hat der Spieler zwei aktive Bewegungsmechaniken. Einerseits einen Dash, bei dem er sich eine kurze Zeit mit einer sehr hohen Geschwindigkeit bewegt und dementsprechend eine kurze waagrechte Distanz überbrücken kann. Andererseits einen Enterhaken, an dem er sich entlang schwingen kann und somit beispielsweise Abgründe überqueren kann.

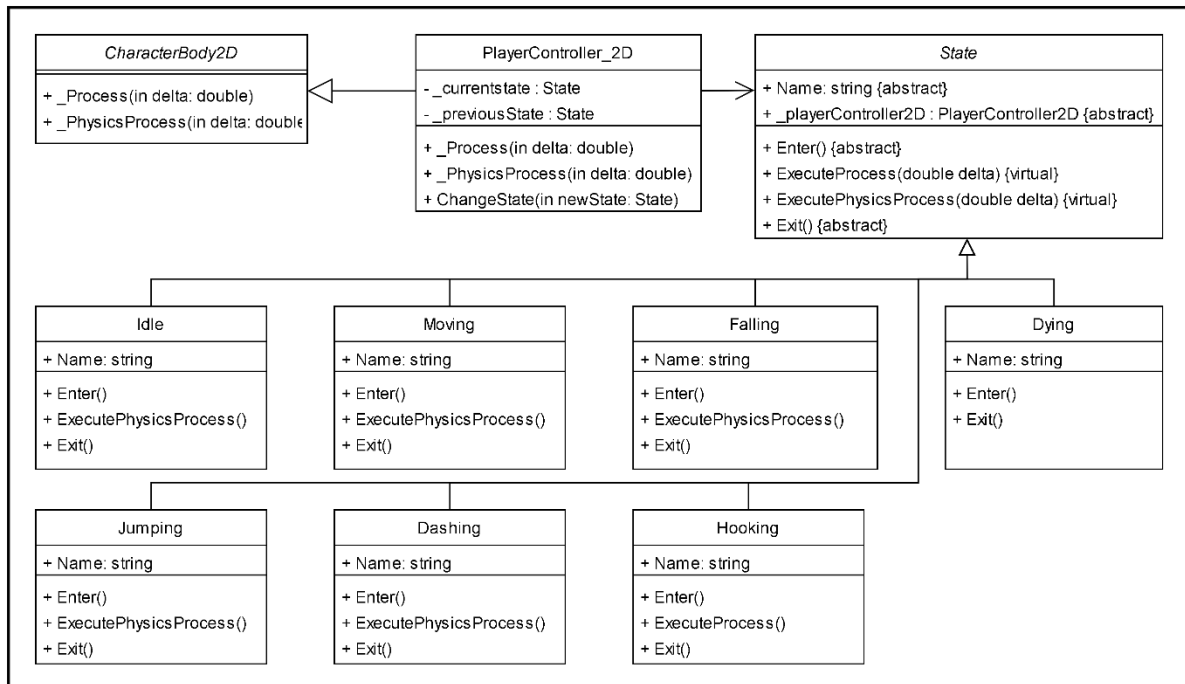


Abb. 1: Konzept für die Implementierung des Spieler-Controllers im State-Pattern

Unter Berücksichtigung der Klassen von Godot erbt der **PlayerController** von der **CharacterBody2D** Node. Es gibt zwei Update-Methoden. Einerseits `Process`, welche pro Frame aufgerufen wird und somit für jegliche nicht physikbasierte Aktualisierungen verwendet werden soll. Andererseits `PhysicsProcess`, welche eine festgelegte Anzahl an Ticks pro Sekunde aufgerufen wird. Diese wird verwendet, um jegliche physikalisch basierte Änderungen durchzuführen. Ebenso hat der **PlayerController** einen aktuellen Zustand. Ein Zustand hat wiederum einen Namen, eine Referenz auf den **PlayerController** und mindestens eine `Enter`- sowie eine `Exit`-Methode. Da nicht jeder Zustand Änderungen in der `Process` oder `PhysicsProcess`-Methode durchführt, müssen diese nicht zwingend implementiert werden. Insgesamt gibt es sieben Zustände: **Idle**, **Moving**, **Falling**, **Dying**, **Jumping**, **Dashing** und **Hooking**. Entsprechend werden die Update-Methoden der Zustände in den Update-Methoden des **PlayerControllers** als erstes aufgerufen. Nachdem die `PhysicsProcess`-Methode den Zustand aktualisiert hat, wird die Methode `MoveAndSlide` aufgerufen. Dies ermöglicht den Zuständen Änderungen, wie zum Beispiel eine Anpassung der Geschwindigkeit, durchzuführen. Es werden also alle Bewegungsmechaniken in den dafür vorgesehenen Zuständen implementiert. Die Übergänge zwischen den Zuständen werden im Zustand selbst implementiert. Ein neuer Zustand kann mittels der `ChangeState`-Methode gesetzt werden. Dabei werden die entsprechenden `Enter`- und `Exit`-Methoden des neuen sowie des vorherigen Zustandes aufgerufen.

2.1.2. Implementierung

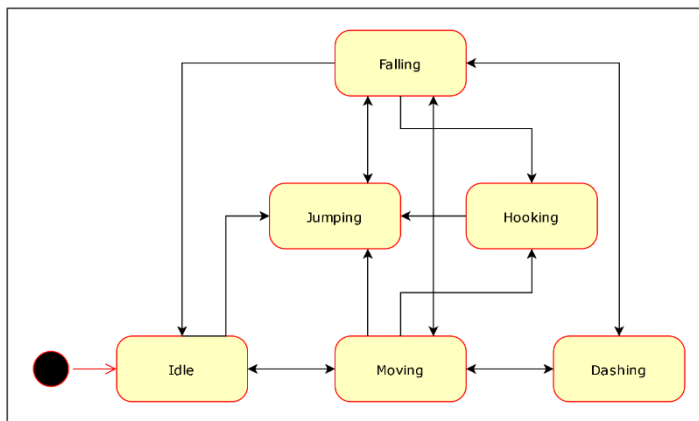


Abb. 2: Zustandsdiagramm für die Bewegungsmechaniken der Spielfigur

Im Idle Zustand befindet sich der Spieler, wenn er sich auf dem Boden befindet und sich nicht bewegt. Es gibt einen Übergang zum Jumping Zustand und einen zum Moving Zustand. Drückt man die Sprungtaste, wechselt der Zustand zu Jumping. Drückt man eine der Bewegungstasten, wechselt der Zustand zu Moving.

Im Moving Zustand befindet sich der Spieler, wenn die Bewegungstasten gedrückt werden und er sich auf dem Boden befindet. Dabei erfährt der Spieler eine Beschleunigung. Wird keine Taste mehr gedrückt, so wird eine Abbremsung auf den Spieler angewendet. Sobald die Geschwindigkeit auf der x-Achse null ist, wechselt der Zustand zu Idle. Wird die Sprungtaste gedrückt, wechselt der Zustand zu Jumping. Wird die Dastaste gedrückt, wechselt der Zustand zu Dashing, sofern dieser verfügbar ist. Hat der Spieler keinen Kontakt zum Boden, wechselt der Zustand zu Falling. Falls ein passendes Ziel vorhanden ist, kann durch Drücken der Enterhakentaste, in den Hooking Zustand gewechselt werden.

Im Falling Zustand befindet sich der Spieler, wenn er keinen Kontakt zum Boden hat. Ebenso kann sich der Spieler während dem Fall nach links oder rechts bewegen. Die Geschwindigkeit des Spielers auf der y-Achse wird auf einen positiven Wert gesetzt. Dabei beschleunigt die Fallgeschwindigkeit bis zu einem bestimmten Wert, welcher nicht überschritten werden kann. Die meisten passiven Plattformer Mechaniken sind ebenfalls in diesem Zustand implementiert und lassen sich über den Konstruktor deaktivieren bzw. beeinflussen.

Sprung Pufferung:

Zur Umsetzung der Sprung Pufferung, wird überprüft, ob die Sprungtaste, während dem Fallen gedrückt wurde. Sobald der Spieler Kontakt zum Boden hat, wird entsprechend in den Jumping Zustand gewechselt.

Coyote-Zeit:

Nachdem in den Falling Zustand gewechselt wurde, wird ein Timer gestartet. Drückt man innerhalb einer festgelegten Zeit die Sprungtaste, so wechselt der Zustand zu Jumping.

Apex-Modifikator:

Nach dem Wechsel des Zustandes von Jumping zu Falling wird für einen kurzen Zeitraum, beim Erreichen der höchsten Position, die Geschwindigkeit auf der x-Achse mit einem Modifikator beschleunigt sowie die Geschwindigkeit auf der y-Achse auf null gesetzt.

Variable Sprung Höhe:

Beim Anwenden der Gravitation wird überprüft, ob die Sprungtaste gedrückt wird. Falls ja, wird die Fallbeschleunigung ohne Modifikator auf die Geschwindigkeit addiert. Falls nein, wird auf die

normale Fallbeschleunigung ein zusätzlicher Modifikator multipliziert, um das Fallen schneller zu machen.

Ebenso kann der Spieler, während dem Falling Zustand Dashing, in dem die Dashtaste gedrückt und sich in eine Richtung bewegt wird. Falls ein passendes Ziel vorhanden ist, kann durch Drücken der Enterhaketaste, in den Hooking Zustand gewechselt werden. Befindet sich der Spieler auf dem Boden und wird eine der Bewegungstasten gedrückt, wechselt der Zustand zu Moving und falls sich der Spieler nicht bewegt, wechselt der Zustand zu Idle.

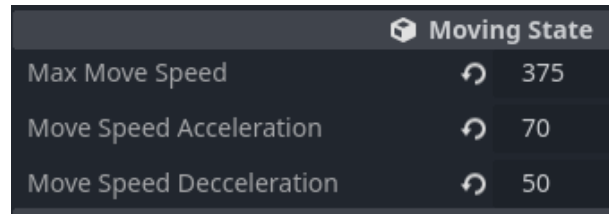
In den Jumping Zustand wechselt der Spieler, sobald die Sprungtaste im Idle oder im Moving Zustand gedrückt wurde. Ebenso kann der Falling Zustand zu Jumping wechseln, da im Falling Zustand die Sprung Pufferung implementiert ist. Die Mechanik des Jumping Zustandes kann auch wiederverwendet und mit anderen Mechaniken kombiniert werden. So wechselt beispielsweise der Hooking Zustand immer zu Jumping, um den Spieler nach dem Benutzen des Enterhakens etwas in die Luft fliegen zu lassen. Der Spieler verbleibt im Jumping Zustand nur wenige Frames, da in diesem lediglich die Bewegungsgeschwindigkeit auf der y-Achse entsprechend auf einen negativen Wert gesetzt wird. Sobald sich der Spieler nicht mehr auf dem Boden befindet, wird in den Falling Zustand gewechselt. Da der Spieler zu Beginn des Falling Zustandes eine negative Geschwindigkeit auf der y-Achse hat, bewegt sich dieser nach oben, da er sich aber im Falling Zustand befindet, nimmt diese ab und folglich fällt der Spieler, bei einer positiven Geschwindigkeit auf der y-Achse, nach unten.

Im Dashing Zustand befindet sich der Spieler, wenn die Dashtaste im Moving oder Falling Zustand gedrückt wurde. Dabei wird die Geschwindigkeit auf der x-Achse auf einen hohen Wert gesetzt, damit sich der Spieler schnell bewegt. Ebenso wird die Geschwindigkeit auf der y-Achse auf null gesetzt, damit sich der Spieler nicht vertikal bewegen kann. Sobald der Timer sein Ende erreicht hat, wechselt der Zustand entweder zu Moving, falls sich der Spieler auf dem Boden befindet oder zu Falling, falls sich der Spieler in der Luft befindet.

In dem Hooking Zustand befindet sich der Spieler, nachdem der Enterhaken erfolgreich an einen entsprechenden Ankerpunkt geschossen wurde. Die eigentliche Schwing-Mechanik ist allerdings nicht in diesem Zustand umgesetzt, da die Mechanik des Enterhakens nicht auf Basis der CharakterBody2D Node umgesetzt wurde. Der Hooking Zustand ist daher eher ein Pseudozustand, in dem nichts ausgeführt wird, dass die Bewegung des Spielers beeinflusst. Sobald der Enterhaken gelöst wurde, wechselt der Zustand zu Jumping, damit der Spieler am Ende einen kleinen Geschwindigkeitsschub nach oben erlangt.

Zusätzlich zu den Zuständen, in denen die Bewegungsmechaniken umgesetzt wurden, gibt es noch den Dying Zustand. In diesem befindet sich der Spieler, sobald er stirbt. Zum Beispiel, wenn er in eine Falle läuft oder von einem Laser getroffen wird. In den Zustand kann von allen anderen Zuständen aus gewechselt werden. Nachdem die Todesanimation abgespielt wurde, wird der Spieler bei dem letzten Checkpoint wiederbelebt und wechselt in den Falling Zustand.

Jegliche Werte, wie zum Beispiel die maximale Geschwindigkeit oder die Sprungkraft können über eine Datei eingestellt werden. Diese können während dem Spielen geändert werden. Somit sind Änderungen sofort ersichtlich, was das Balancing deutlich erleichtert.



Moving State		
Max Move Speed	↺	375
Move Speed Acceleration	↺	70
Move Speed Deceleration	↺	50

Abb. 3: Ausschnitt aus der PlayerController2D_Data.tres Datei

2.2. Enterhaken

Nachdem der Spieler-Controller mit den geforderten Bewegungsmechaniken fertig gewesen ist, habe ich mich mit der Implementierung des Enterhakens auseinandergesetzt. Dieser sollte immer eine feste Länge haben, sodass der Spieler nur daran schwingen kann. Ebenso sollte ein Indikator erscheinen, wenn man diesen benutzen kann, da dies nur an bestimmten Stellen in der Spielwelt möglich sein sollte.

2.2.1. Konzeption

Da ich zu Beginn keine wirkliche Idee hatte, wie ich an den Enterhaken rangehen sollte, recherchierte ich im Internet, wie man einen Enterhaken in Godot umsetzen könnte. Vergebens musste ich aber feststellen, dass es keine wirklich guten Referenzen für einen Enterhaken mit den genannten Anforderungen gibt. Auch die Suche nach passenden Enterhaken, welche in der Unity Engine umgesetzt wurden, sahen zum einen nicht gut aus und zum anderen konnten diese nicht in der Godot Engine umgesetzt werden, da die Unity Engine andere Komponenten zur Verfügung stellt. Schließlich habe ich mehrere Prototypen entwickelt, welche mehr oder weniger gut funktionieren.

Meine erste Idee war es, einfach zwei Nodes zu verwenden, wobei eine als Kind untergeordnet ist. Der Spieler wird dabei auf die Position der Child Node gesetzt und das Schwingen, wird über das

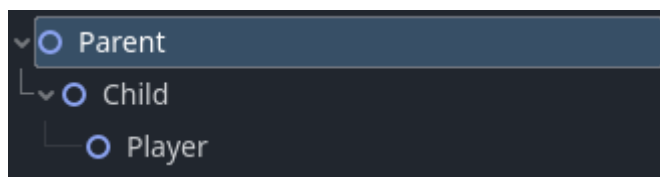


Abb. 4: Prototyp bestehend aus Node2D's

Beeinflussen der Rotation der Eltern Node erzeugt. Dieser Ansatz funktioniert zwar, allerdings nicht sehr gut. Das Schwingen fühlte sich schlecht an und sah sehr robotermäßig aus.

Mein zweiter Ansatz bezieht sich auf ein Video, welches ich bei meiner Recherche gefunden habe. In dem Video (DualWielded, 05.06.2022) wird ein Enterhaken als Seil, bestehend aus lauter kleinen physikalischen Objekten implementiert. Im Video wird die Unity Engine genutzt, daher musste ich ein Seil in der Godot Engine umsetzen. Dies war relativ einfach da, das Prinzip gleich bleibt. Dabei besteht das Seil in der Godot Engine aus vielen kleinen RigidBody2D's. Der Spieler würde wieder auf die Position des Endes gesetzt werden, welches frei schwingen kann. Das bedeutet man steuert nicht den Spieler, sondern eine PhysicsBody2D Node mithilfe von Kräften, welche auf die Node angewendet wird.

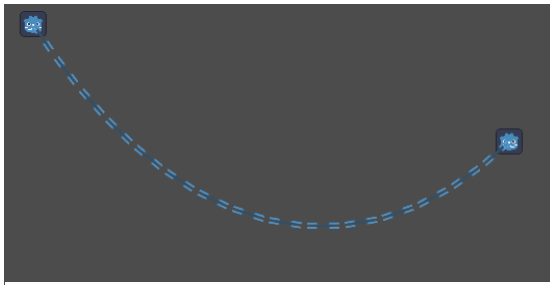
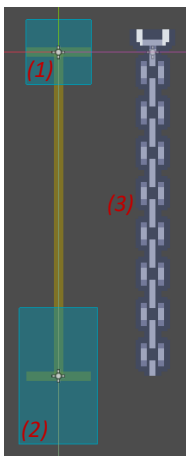


Abb. 5: Implementierung eines Seils in der Godot Engine bestehend aus einzelnen RigidBody's

Zwar ist dieser Ansatz, im Gegensatz zum ersten, schon eine deutliche Verbesserung, doch gab es Aspekte, welche dazu geführt haben, dass ich diese Idee wieder verwarf. Das Seil ist elastisch, aufgrund der vielen RigidBody's, welche durch PinJoint2D's verbunden sind. Diese verbinden zwei RigidBody2D's miteinander und bewirken, dass diese immer einen bestimmten Abstand zueinander haben. Diesen

Dehnungseffekt konnte man zwar durch viele Feinheiten in den Eigenschaften verringern, doch nicht komplett weg bekommen, sodass sich die einzelnen Segmente des Seils immer etwas ausdehnen konnten. Ebenso war die Bewegung des Endstücks nicht sehr präzise, da dies beim Anwenden von Kräften stark hin und her wackeln konnte. Dies ließ sich zwar auch mit Feinheiten in den Einstellungen korrigieren, allerdings wieder nicht so gut. Letzten Endes war das Verhalten des Seils zu unvorhersehbar und daher entschloss ich, den Ansatz nicht zu verwenden.

Schließlich bin ich nach längerem ausprobieren von verschiedenen Komponenten der Godot Engine auf die DampedSpringJoint2D Node gestoßen. Dieser Joint verbindet zwei PhysicBody2D Node's miteinander, sodass diese immer einen bestimmten Abstand zueinander haben. Wird der Abstand überschritten, bewegen sich die Nodes entsprechend aufeinander zu und verhalten sich, wie eine Feder. Die Stärke, wie schnell sie sich aufeinander zubewegen sowie den Abstand, welchen sie zueinander einhalten sollen, kann man einstellen. Mit entsprechenden Anpassungen kann dieser Joint, einen passenden Enterhaken simulieren. Die Mechanik des Enterhakens besteht folglich aus zwei



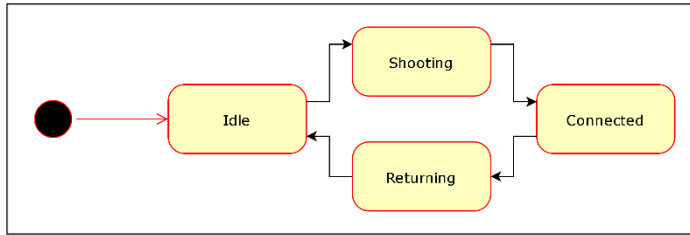
Nodes. Das Endstück, welches am Ziel, und das Anfangsstück, welches am Spieler positioniert wird. Das Endstück ist ein StaticBody2D, da sich dieser nie bewegt und das Anfangsstück ein RigidBody2D. Um nun ein Schwingen zu simulieren, wird lediglich auf das Anfangsstück eine Kraft nach links oder rechts angewendet. Da das Anfangsstück aufgrund des Joints immer den gleichen Abstand zum Endstück behält, schwingt dieses folglich auf einer Kreisbahn nahezu perfekt um das Endstück herum. Um den Enterhaken visuell anzuzeigen, wird eine Line2D Node verwendet, welche aus zwei Punkten besteht und entsprechend auf das Endstück sowie das Anfangsstück positioniert werden.

Abb. 6: Enterhaken bestehend aus einer DampedSpringJoint2D Node mit Endstück (1) Anfangsstück (2), sowie einer Line2D Node (3)

2.2.2. Implementierung

Ebenso wie beim PlayerController, verwendete ich für den Enterhaken das State-Pattern. Da der Enterhaken aus zwei Teilen besteht, einer mechanischen und einer visuellen Komponente, konnten die Abläufe problemlos und unabhängig voneinander in den Zuständen implementiert werden. Insgesamt besteht der Enterhaken aus vier Zuständen, siehe Abb. 9. Im Idle Zustand befindet sich der Enterhaken, wenn er nicht benutzt wird, entsprechend wird die visuelle Komponente deaktiviert.

Abb. 9: Zustandsdiagramm des Enterhakens



Sobald ein passendes Ziel übergeben wird, wechselt der Zustand zu Shooting. Dabei bewegt sich der Enterhaken in die Richtung des Ziels. Ist er am Ziel angekommen, wechselt er zu Connected. In diesem Zustand wird die

Steuerung des Spielers deaktiviert und die Position der Spielfigur kontinuierlich auf die Position des Endstücks gesetzt. Ebenso wird die Steuerung des Endstücks aktiviert, damit der Spieler, das Schwingen steuern kann. Durch Drücken der Taste, die den Enterhaken löst, wechselt der Zustand zu Returning. Im Returning Zustand, bewegt sich der Enterhaken zurück zum Spieler, und wechselt wieder in den Idle Zustand und kann erneut ausgelöst werden.

2.3. Sonstige Tätigkeiten

Abseits von der Implementierung der Steuerung des Spielers und dessen Bewegungsmechaniken habe ich mich noch um andere Aufgaben gekümmert.

- Implementierung eines Icons für den Dash, um dem Spieler anzuzeigen, wann dieser verfügbar ist
- Implementierung einer Kamera, welche die Spielfigur mit einer leichten Verzögerung verfolgt
- Implementierung der Animationen für die Spielfigur
- Erstellung des Präsentationsvideos

2.4. Probleme

Direkte Probleme hatte ich bei der Umsetzung des Spieler-Controllers sowie des Enterhakens nicht, da das meiste eher darauf zurückzuführen ist, dass ich mich nicht genug mit der Godot Engine ausgekannt habe. In den meisten Fällen bestand der Arbeitsfortschritt aus Trial-and-Error, wie bei der Umsetzung des Enterhakens, wodurch ich die Engine und dessen Komponenten besser verstand.

Bei der Implementierung einer Kamera, die den Spieler mit einer leichten Verzögerung verfolgt, gab es Probleme. Aufgrund der niedrigen Auflösung sind bereits kleinste Veränderungen der Kameraposition sichtbar. Diese funktioniert durch einen Lerp zwischen der Position der Kamera und des Spielers. Dabei war das Bild nicht sehr flüssig bzw. der Spieler stotterte/zitterte ein wenig. Das Problem ließ sich auf den Lerp zurückführen. Dabei hat Martin festgestellt, dass bei dem Lerp der Godot Engine größere Ergebnisse auftreten konnten als der maximale Wert, welchen man angegeben hat. Schließlich konnte das Problem durch einen vorherigen Clamp gelöst werden. Lediglich wenn der Spieler auf einer bewegenden Plattform steht, kommt es noch zu einem minimalen stottern. Abgesehen davon gab es einen Effekt, welcher aufgetreten ist, sobald der Spieler den Enterhaken benutzt hat. Dabei flackerte das Bild kurz und man sah die Spielfigur oder den Enterhaken an der vorherigen Position auftauchen, siehe Abb. 8. Das lag daran, dass der Spieler, nachdem der Enterhaken sein Ziel erreicht hat, auf die Position des Endstücks des Enterhakens gesetzt wird. Im Code wird zuerst die Position des Enterhakens

auf die Position des Spielers gesetzt und dann, solange der Enterhaken genutzt wird, die Position des Spielers auf die Position des Endstücks gesetzt. Die Lösung für dieses Problem bestand darin, eine Verzögerung von 0.05s einzubauen, bevor die Position des Spielers zum ersten Mal auf die Position des Endstücks gesetzt wird.

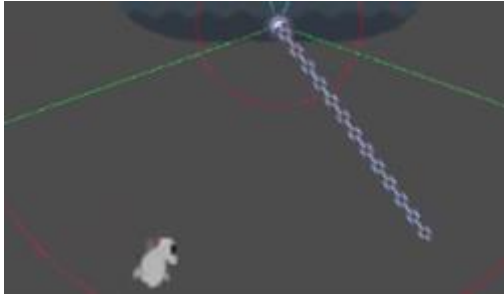


Abb. 8: Kamera Flacker-Effekt beim benutzen des Enterhakens - Spieler an vorheriger Position des Enterhakens

Bei der Implementation der Animationen für die Spielfigur gab es Probleme mit den Sprite Atlassen. Diese habe ich zum einen mit einer Auflösung im Bereich von 10000 x 10000 übergeben bekommen, was für ein Pixel Art Spiel viel zu groß ist und unnötigen Speicherverbrauch verursacht und zum anderen eigentlich keine Sprite Atlasse sind, da die einzelnen Bilder im Atlas willkürlich und ohne gleichmäßige Abstände angeordnet sind. Dies machte es unmöglich, die Animationen in diesem Zustand in der Engine zu importieren. Aus zeitlichen Gründen habe ich mich dazu entschieden, die Atlasse selbst zu korrigieren. Dazu habe ich zunächst die fehlerhaften Atlasse auf eine passende Auflösung herunterskaliert und anschließend jeden einzelnen Sprite passend zu einem Sprite Atlas angeordnet.



Abb. 9: Sprite Atlas für Idle mit Fehler

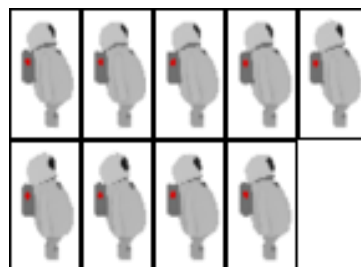


Abb. 10: Sprite Atlas Idle korrigiert

3. Abschluss

Insgesamt war es ein arbeitsintensives Projekt, das mir sehr großen Spaß bereitet und mir einen tiefen Einblick in die Godot Engine gegeben hat. Unter anderem konnte ich dabei meine Fähigkeiten mit der Programmiersprache C# erweitern und erneut in der Teamarbeit feststellen, wie wichtig Planung, Strukturierung, Kommunikation, klare Schnittstellen und Zeitmanagement in der Zusammenarbeit sind. Was mir besonders gut gefallen hat, war unser Kommunikationsfluss innerhalb des Teams. Abseits der zweiwöchentlichen Termine an der Hochschule, gab es wöchentliche Teammeetings als auch ein wöchentliches Einzelgespräch mit Stefan, dem Projektleiter. Dadurch wusste man stets, wer sich mit welchen Aufgaben beschäftigt und wie der aktuelle Stand der Dinge ist. Ebenso konnten auftretende Probleme gut im Team kommuniziert und schließlich gemeinsam gelöst werden. Insgesamt gestaltete sich das Arbeiten im Team sehr angenehm.

4. Quellenverzeichnis

Stand der Weblinks: 20.06.2023

4.1. Bild- & Videoverweis

- DualWielded. (05.06.2022). I Added A Grappling To My Game... - Incaved Devlog. [Video]. YouTube. <https://www.youtube.com/watch?v=NcQjW92fRY>