Epiales Dokumentation

Teammitglieder

Lena Sophie Musse (lm138)
Marco de Jesus António (md131)
Marian Syska (ms495)
Simon Geupel (sg184)

Betreuung durch

Stefan Radicke
Martin Fuchs
Don-Oliver Matthies

Software Entwicklungsprojekt
Modulnummer: 113400
Stage Projektnummer: 3750

Inhaltsverzeichnis

Einführung und Ziele

Aufgabenstellung

Die Aufgabenstellung bestand in der Unterstützung der Studioproduktion Epiales. Darin sollte ein 2D- Sidescroll Spiel mit Kampfmechaniken entwickelt werden. Die Entwicklung des gesamten Spiels ist jedoch zeitlich in einem Semester nicht umsetzbar. Aufgrund dessen bestand das Ziel darin, die Grundlagen für eine Weiterführung des Projekts zu schaffen. Ziel war die Implementierung eines grundlegenden Gameloops, eines Levels mit Jump & Run Passagen und Kampfsektionen, sowie Grundmechaniken des Hauptcharakters und wenige, einfache Gegnertypen.

Qualitätsziele

Die Qualitätsziele wurden im Sommersemester durch die Studenten des Studiengangs Audiovisuelle Medien erarbeitet und sollten als Maßstab der Qualitätskontrolle dienen. Das Spiel wurde regelmäßig durch das Team der Gamedesigner auf die Qualitätsziele überprüft. Die Qualitätsziele sind:

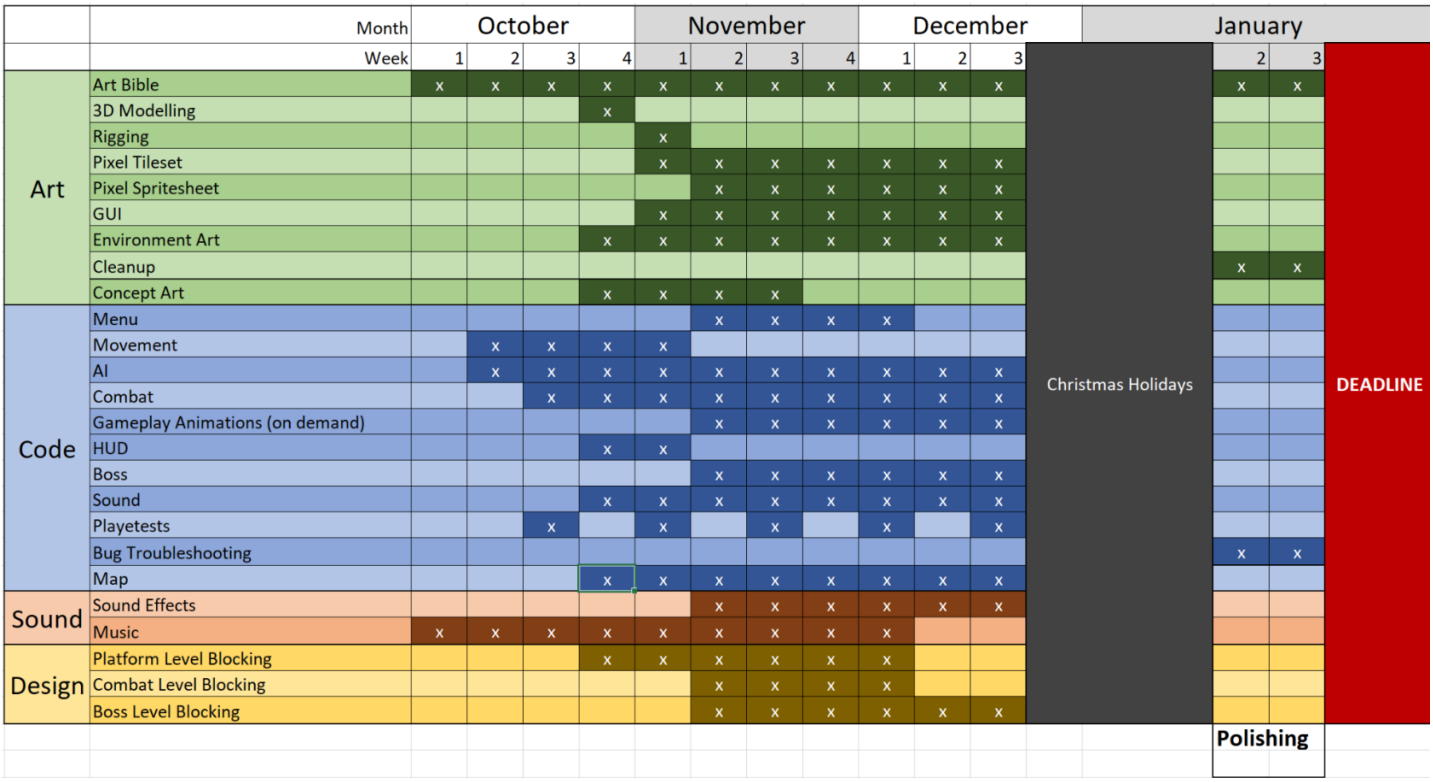
- flüssiges Platformer Gameplay in der Art des Spiels “Celeste”
- Intensives Kampfgefühl in der Art des Spiels “Whiskers and Wags”.
- Übermächtiges, immersives Spielgefühl das stilvolles Spielen belohnt

Stakeholder

Stakeholder waren in diesem Projekt die Professoren des Studiengangs Herr Martin Fuchs und Herr Don-Oliver Matthies. Durch wöchentliche Stakeholder Reviews wurden diese auf den neuesten Stand des Projekts gebracht, Probleme besprochen und gemeinsam Lösungen gefunden. Dabei wurde nach dem Update des Gesamtprojekts in Teilgruppen des Art- und des Developmentteams genauer auf die respektiven Problemstellungen eingegangen.

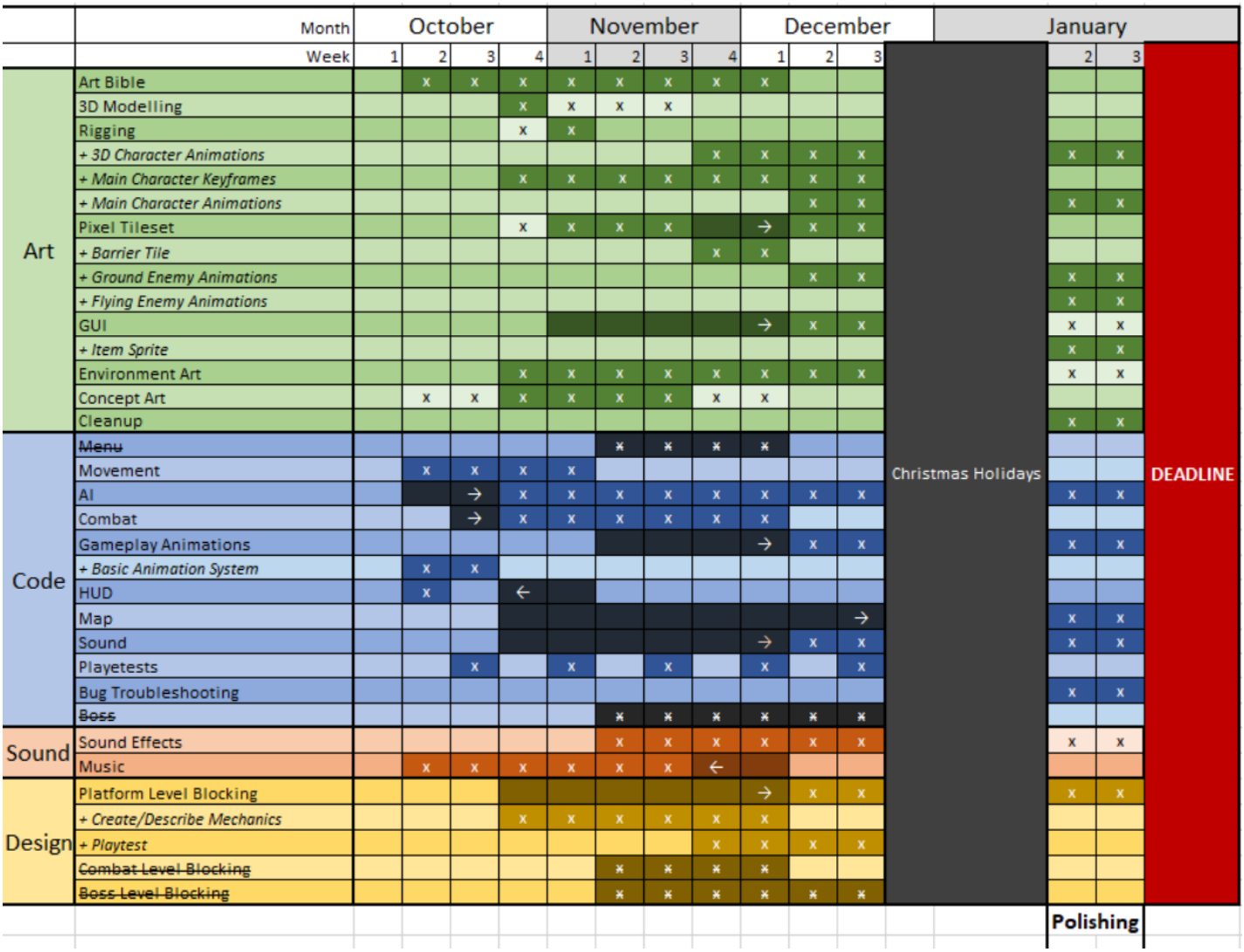
Projektplan

Ursprünglicher Projektplan zu Beginn des Semesters



	Month	October				November				December				January		
	Week	1	2	3	4	1	2	3	4	1	2	3		2	3	
Art	Art Bible	x	x	x	x	x	x	x	x	x	x	x	Christmas Holidays	x	x	DEADLINE
	3D Modelling				x											
	Rigging					x										
	Pixel Tileset					x	x	x	x	x	x	x				
	Pixel Spritesheet						x	x	x	x	x	x				
	GUI					x	x	x	x	x	x	x				
	Environment Art				x	x	x	x	x	x	x	x				
	Cleanup													x	x	
	Concept Art				x	x	x	x								
Code	Menu						x	x	x	x						
	Movement		x	x	x	x										
	AI		x	x	x	x	x	x	x	x	x	x				
	Combat			x	x	x	x	x	x	x	x	x				
	Gameplay Animations (on demand)						x	x	x	x	x	x				
	HUD				x	x										
	Boss						x	x	x	x	x	x				
	Sound				x	x	x	x	x	x	x	x				
	Playetests			x		x		x		x		x				
	Bug Troubleshooting													x	x	
	Map				x	x	x	x	x	x	x	x				
Sound	Sound Effects						x	x	x	x	x	x				
	Music	x	x	x	x	x	x	x	x	x						
Design	Platform Level Blocking				x	x	x	x	x	x						
	Combat Level Blocking						x	x	x	x						
	Boss Level Blocking						x	x	x	x	x	x				
														Polishing		

Angepasster Projektplan



Legende

- Heller -> dazu gekommen
- Dunkler -> weggefallen
- *+Kursiv* -> neues Arbeitspaket
- Durchgestrichen = Arbeitspaket weggefallen
- Pfeil = verschoben

Randbedingungen

Technische Randbedingungen

Godot

Das Projekt wurde mit der open-source engine godot umgesetzt. Die Engine ist verglichen mit Unity oder Unreal Engine eher schlank und übersichtlich, was einen leichten Einstieg ermöglicht. Trotzdem sind alle Tools, welche ein 2D-Sidescroller benötigt, vorhanden.

GDScript

Auch die in Godot standardmäßig verwendete Skriptsprache GDScript ist sehr einsteigerfreundlich und bietet eine intuitive Programmierung. Trotzdem war es anfangs etwas schwierig, nach Sprachen wie Java und C#, sich in die Skriptsprache einzudenken und die Konventionen zu beachten.

Plattformunabhängigkeit

Das Spiel soll am Ende auf Windows, macOS und Linux installiert werden können und außerdem plattformunabhängig im Browser spielbar sein. Godot bietet die Möglichkeit, das Spiel am Ende entsprechend zu bauen.

Organisatorische Randbedingungen

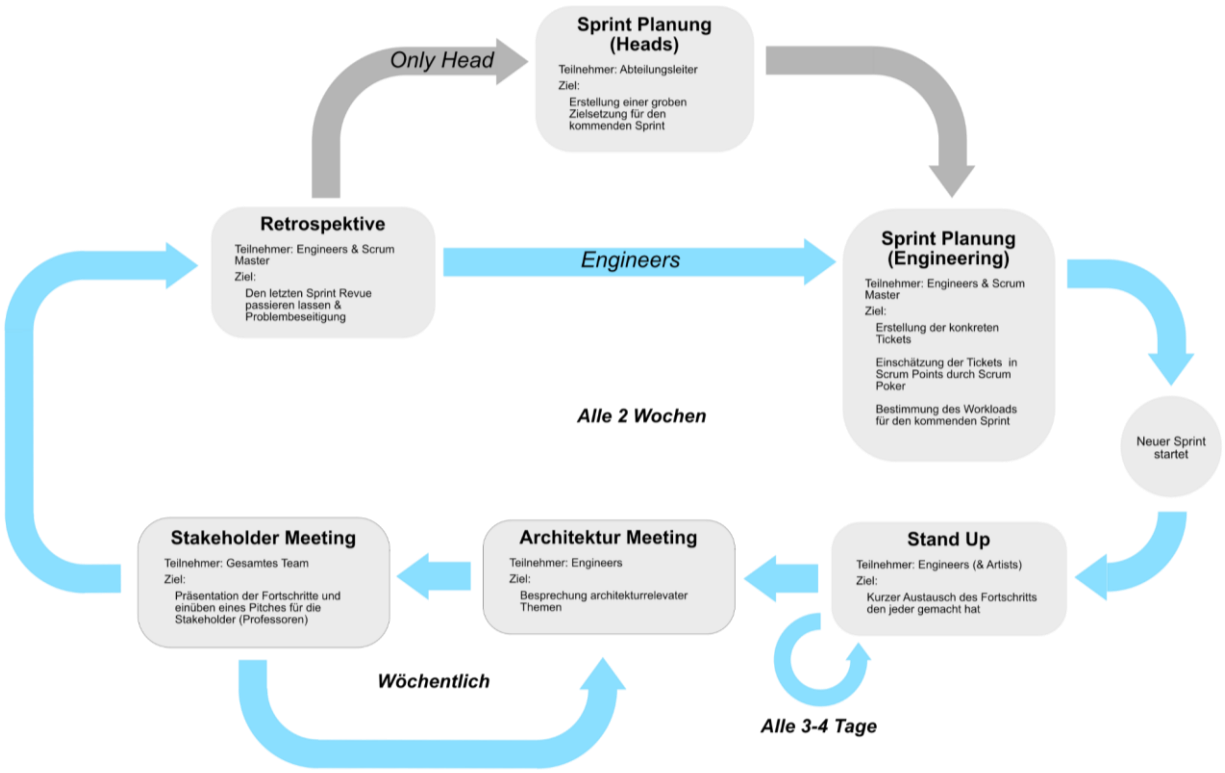
Team

Das Engineering-Projekt Team bestand aus vier MI-Studenten, drei davon wiesen Vorerfahrungen in einem vorhergegangenen Projekt auf, einer davon leicht fortgeschrittenere Vorerfahrungen in mehreren Projekten auf. Zudem wurde das Projekt noch extern von einem AM-Studenten mit leichter Erfahrungen und einer CSM-Studentin mit fortgeschrittener Erfahrung ergänzt. Extern standen ein Game Design Team bestehend aus drei AM-Studenten, ein Artist Team bestehend aus 11 AM-Studenten und ein Ton und Sound Team bestehend aus 2 AM-Studenten bereit. Für alle externen Teams war dies das erste Projekt. Die Arbeitszeit aller Beteiligten richtete sich nach der jeweiligen Studienordnung. So wendeten die Artists 16 Wochenstunden, die Entwickler 8 Wochenstunden auf.

Ressourcen

In der Hochschule der Medien standen allen Mitwirkenden, die das Gebäude betreten durften, ein Raum mit Arbeitsmaschinen, Whiteboards und Grafiktabletts zur Verfügung. Für jegliche zusätzliche Kommunikation stand ein Discord-Server zur Verfügung. Zur Koordinierung des Projektes wurde ein Confluence Bereich aufgesetzt. Es stand ein Web-Space zum Deployen des Spieles zur Verfügungen. Desweiteren stand der MI-GitLab-Server der Hochschule der Medien zur Versionierung des Projektes zur Verfügung.

Ablauf



Zur Synchronisation der Teams wurde ein **Scrum-Workflow** vorausgesetzt. Die Scrum-Sprints wurden auf 2 Wochen angesetzt. Für das Engineering-Projekt-Team wurden zwei-wöchige Stand-Ups eingeplant. Zusätzlich fanden notwendige Scrum-Planning Meetings statt. Benötigte Tickets (Aufgaben) und Sprints wurden über **Hack N Plan** verwaltet.

Darüber hinaus wurde wöchentlich, jeden Mittwoch, ein Stakeholder-Meeting festgesetzt. In dem Meeting wurde von AM-Seite ein Stakeholder, der über Fortschritte in Kenntnis gesetzt wird, simuliert und ein Pitch eingeübt.

Zeitlich war das Projekt gebunden an die Länge des Semesters. Über das Projekt hinweg mussten festgesetzte Deadlines eingehalten werden. Zunächst musste ein Prototyp des Projektes bis zum 22.12.2021 fertig sein. Darüber hinaus musste ein fertige Version für die AM-Abgabe bis zum 23.01.2022 bereitgestellt werden und für die Präsentation auf der Media Night am 27.01.2022 konnten noch kleinere Änderungen erfolgen. Zu guter Letzt ist eine weiter Version für die MI-Abgabe zum 17.02.2022 angesetzt.

Bei der Entwicklung musste eine eventuelle Fortsetzung des Projektes von nicht eingearbeiteten Nachfolgern mitberücksichtigt werden.

Konventionen

Dokumentation

Die Dokumentation hält sich an den Aufbau des deutschsprachigen arc42-Templates.

Sprache

Der Quellcode, sowie Kommentare, Commit Messages und Issue Tracking wurden auf Englisch geschrieben. Auch das Ingame-UI ist für eine universelle Nutzung englisch.

Organisatorisches geschah meist auf Deutsch und Englisch, da auch ein paar internationale Studenten am Projekt beteiligt waren.

Coding Conventions

Es wurden größtenteils die GDScript Coding conventions verwendet (https://docs.godotengine.org/de/stable/tutorials/scripting/gdscript/gdscript_styleguide.html). Teilweise war dies durch viel Erfahrung in Sprachen wie Java aber sehr gewöhnungsbedürftig.

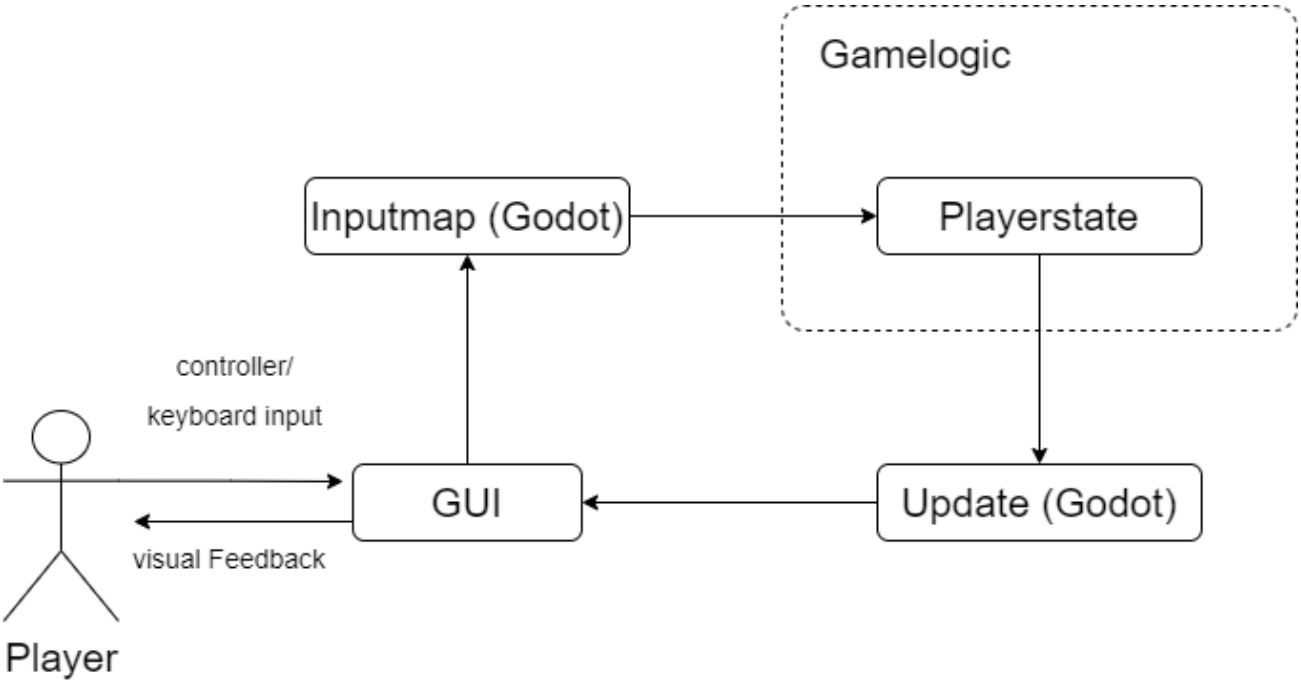
Benennung der Branches

Bei der Benennung der Branches wurde das Muster “Zweck/Beschreibung”. War ein Branch einem Issue in Hack’and’Plan zugeordnet, begann die Beschreibung des Branches mit der Issue Nummer, gefolgt von dem Namen des Tickets.

Kontextabgrenzung

Entwickelt wurde ein Offline Singleplayer 2D-Sidescroll - Platformer. Es wurde bewusst auf Mehrspieler Mechaniken und Serverstrukturen verzichtet. Entsprechend ist auch die Softwarearchitektur von Epiales aufgebaut.

Fachlicher Kontext



Der Spieler kann über die Benutzeroberfläche mit dem Spiel interagieren. Dabei werden sowohl Controller als auch Tastatureingaben akzeptiert. In den einzelnen States des Spielers werden die jeweiligen Eingaben verarbeitet und auf entsprechende Aktionen, wie zum Beispiel Bewegungen, Sprünge oder Angriffe des Spiels, gemapped. Dabei sind je nach State manche Eingaben erlaubt oder eingeschränkt. Beispielsweise ist der Leap Jump nur möglich wenn sich der Spieler bereits im Dash befindet.

Lösungsstrategie

Technologien

Neben den festgesetzten Bedingungen(Hack N Plan, Discord, Confluence etc.) hat sich das Team für folgende Technologien entschieden.

Epiales wurde auf der Basis von der Gameengine **Godot** entwickelt. Ausschlagspunkt für diese Entscheidung war hauptsächlich die gute Unterstützung von Godot für die 2D-Spieleentwicklung, die leichte Benutzung für Einsteiger und die gute Verwendung mit Versionierungsverwaltungen. So bietet es ein gutes 2D Physiksystem zur Entwicklung eines komplexen Charakters und Einbindung von Tilemaps.

Die Entscheidung **Git/GitLab** als Versionierungs-Tool zu verwenden fiel leicht, da ein MI-GitLab Server zur Verfügung stand und alle Partizipierenden bereits Vorerfahrungen gemacht hatten.

Entwurfs- und Architekturmuster

Es wurde versucht möglichst die Gesamtstruktur als **Schichtenmodell** aufzubauen und Elemente unabhängig von ihrer Umgebung zu implementieren. Benötigte Abhängigkeiten sollten per **Dependency Injection** übergeben werden. Für die Kommunikation zwischen zwei Elementen sollte möglichst ein **Beobachtungsmuster** angewandt werden, wie es im Schadenssystem in Form eines Empfängers und Senders implementiert wurde. Dies ermöglicht eine unabhängigen Austausch. Desweiteren wurde eine Trennung zwischen Spieldaten und Anwendungsdaten geplant um Möglichkeiten zu bieten Spieldaten zu speichern.

In mehreren Szenarien erwies sich eine **State Machine** als nutzbringende Lösung (Bsp. Spieler, Gegner etc). Die **State Machine** bot dabei auch eine Möglichkeit Sub State Machines einzubinden, um komplexere Abläufe zu modellieren.

Für die “künstliche Intelligenz” der Gegner wurde eine einfache Separation der Logik und des Verhaltens gewählt um eine leichte Anpassung des Verhaltens zu ermöglichen. Um das Verhalten zu modellieren wurde ein rein **situationsgetriebener Ansatz** implementiert. Die Gegner reagieren lediglich auf äußere Veränderung. Sie besitzen kein Gedächtnis, lernen nicht dazu und planen nicht in die Zukunft.

Organisatorische Entscheidungen

Workflow

Selbst mit hilfe von GitLab, ist das Zusammenarbeiten von sechs Programmierern an einem Spiel gar nicht so einfach. Zu Beginn wurde der Main Branch gesperrt und alle Merge-Requests sollten regelmäßig vom Lead-Programmer gemerged werden. Allerdings unterschieden sich die Branches mit der Zeit so sehr, dass Merge Konflikte immer aufwändiger zu lösen wurden. Schließlich entschieden wir uns den Gitflow-Workflow anzuwenden und den Development Branch dazwischen zu schalten, wobei jeder die Berechtigung hatte zu pushen. Dadurch war eine viel bessere Synchronisation möglich. Auf den Main wurde dann immer wieder eine Bugfreie Version des Development Branches gepusht.

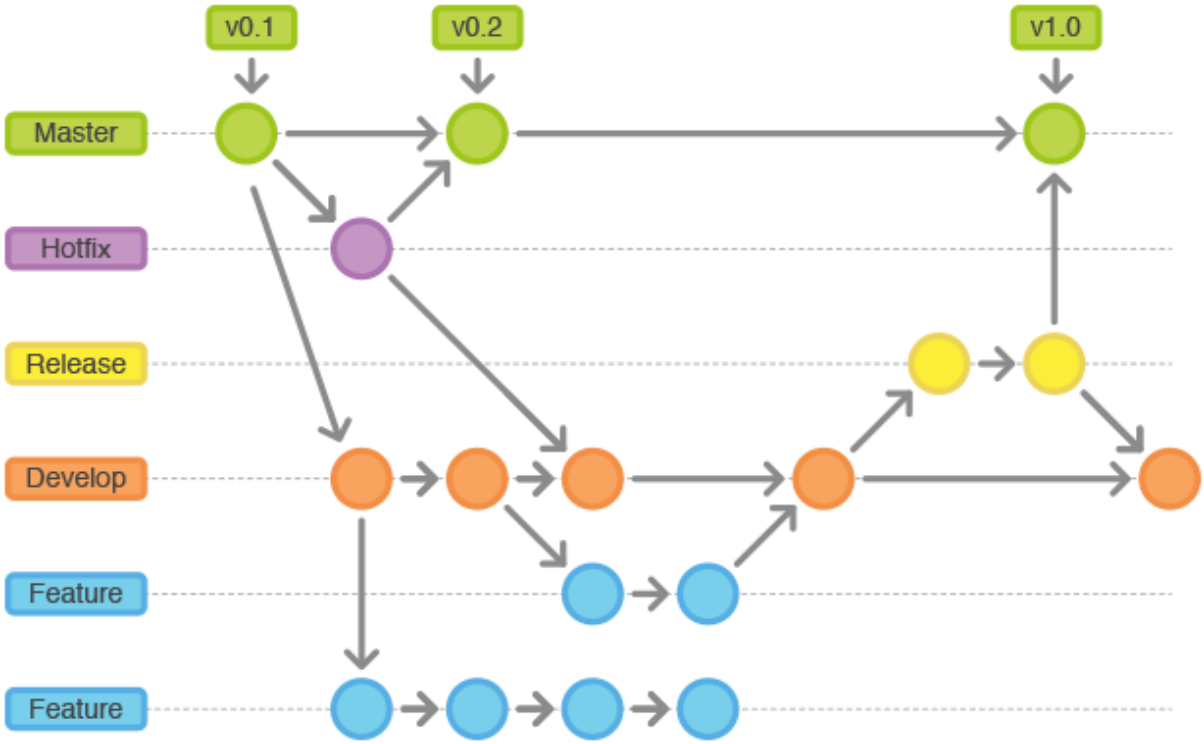


Abbildung: **Gitflow-Workflow** (https://blog.seibert-media.net/blog/2014/03/31/git-workflows-der-gitflow-workflow-teil-1/)

Debugging

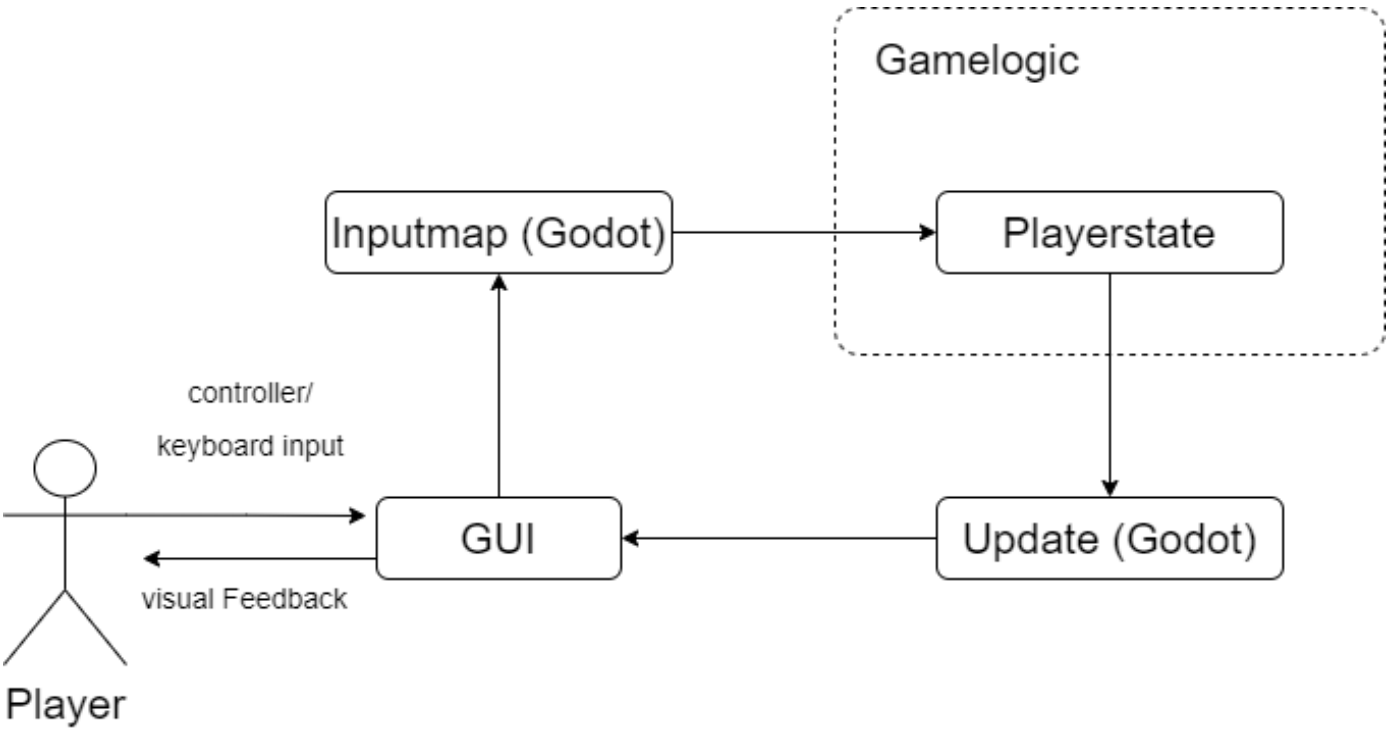
Zum Debuggen wurden verschiedene Methoden verwendet.
Der Godot eigene Debugger ist ein hilfreiches Werkzeug. Er wurde oft verwendet, um verhalten von Klassen und Methoden im Detail nachzuvollziehen. Leider war es nicht immer möglich ihn zu benutzen. Dies war der Fall, wenn beispielsweise die Ursache für ein Problem schwer zu lokalisieren war.
In diesem Fall waren printf-debugging und logging sehr hilfreich. Während der Entwicklung wurden bestimmte Ereignisse, wie zum Beispiel der Wechsel zwischen states geloggt. So konnten viele Fehlerquellen gefunden werden. Da allerdings regelmäßig print-Befehle im Code vergessen und später gesucht wurden, führten wir die Konvention ein, den Ort des print Aufrufs zu loggen.

Playtests

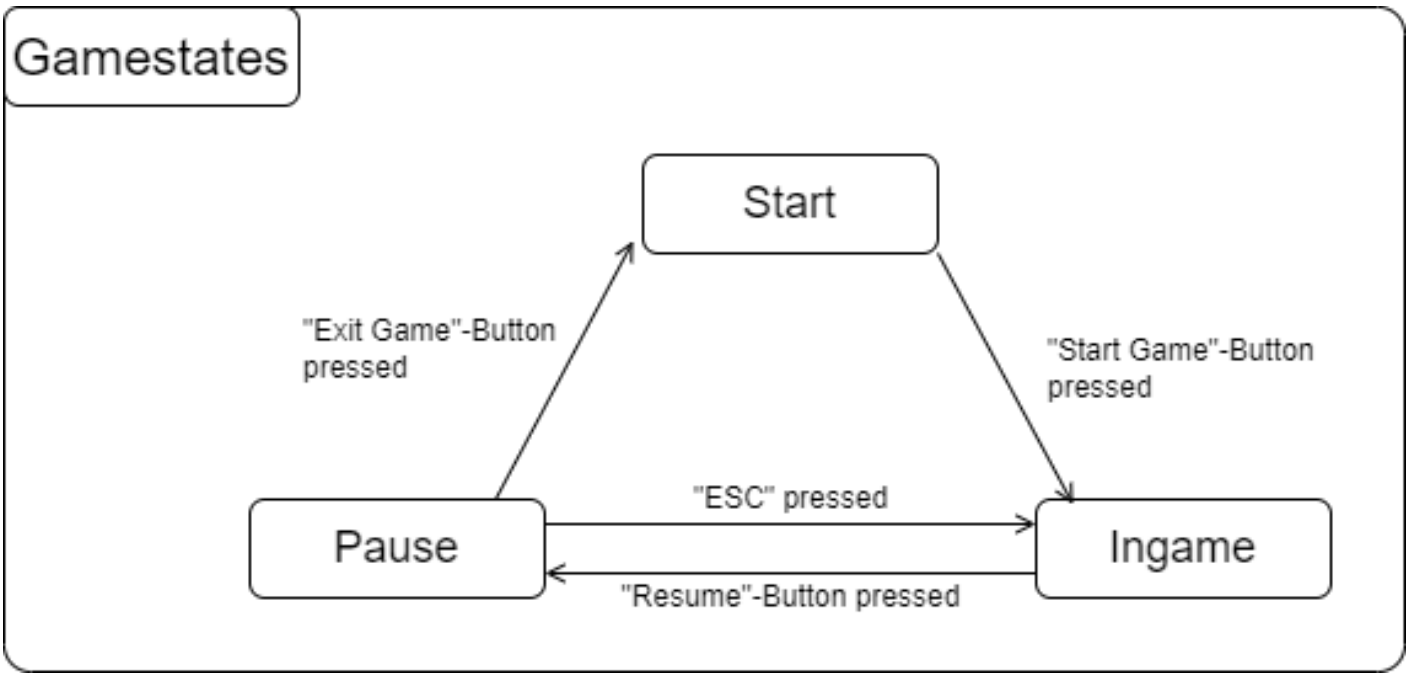
Playtests sind ein sehr effektives Mittel, um schon während der Entwicklung eine Qualitätssicherung zu realisieren. Aus diesem Grund wurden alle implementierten Features von den Game Designern getestet. So konnten viele Bugs gefunden werden.
Wenn ein Feature nicht die Erwartungen der Playtester erfüllte, wurde im nächsten Sprint ein weiteres Mal das Feature bearbeitet. So wurden iterativ Features fertig gestellt.
Der letzte “Playtest” fand durch Spieler und Spielerinnen an der Medianight statt. So konnten neben Bugs auch grundsätzliche Probleme, wie die Selbsterklärbarkeit des Spiels festgestellt werden.

Bausteinsicht

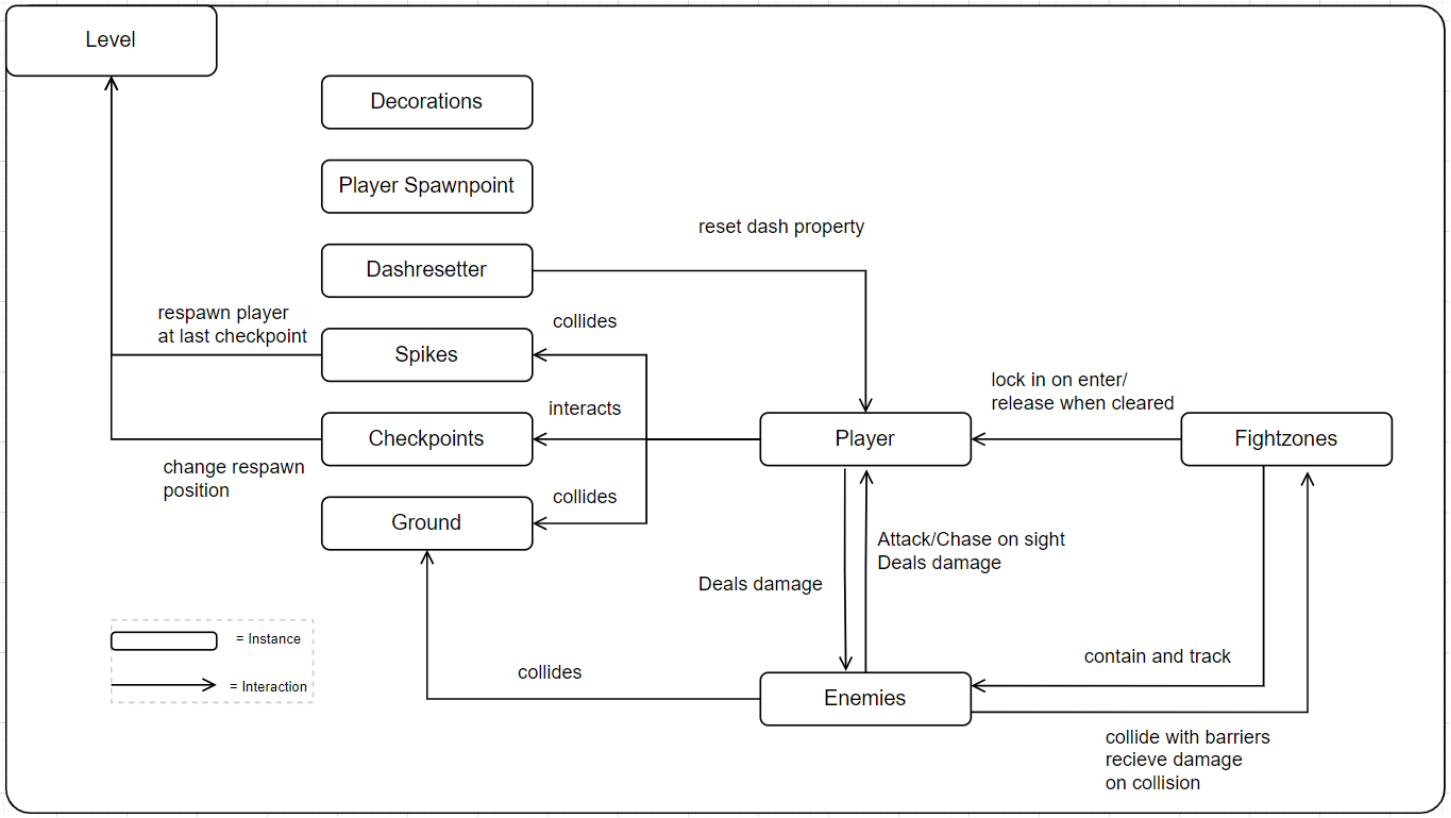
Kontextabgrenzung



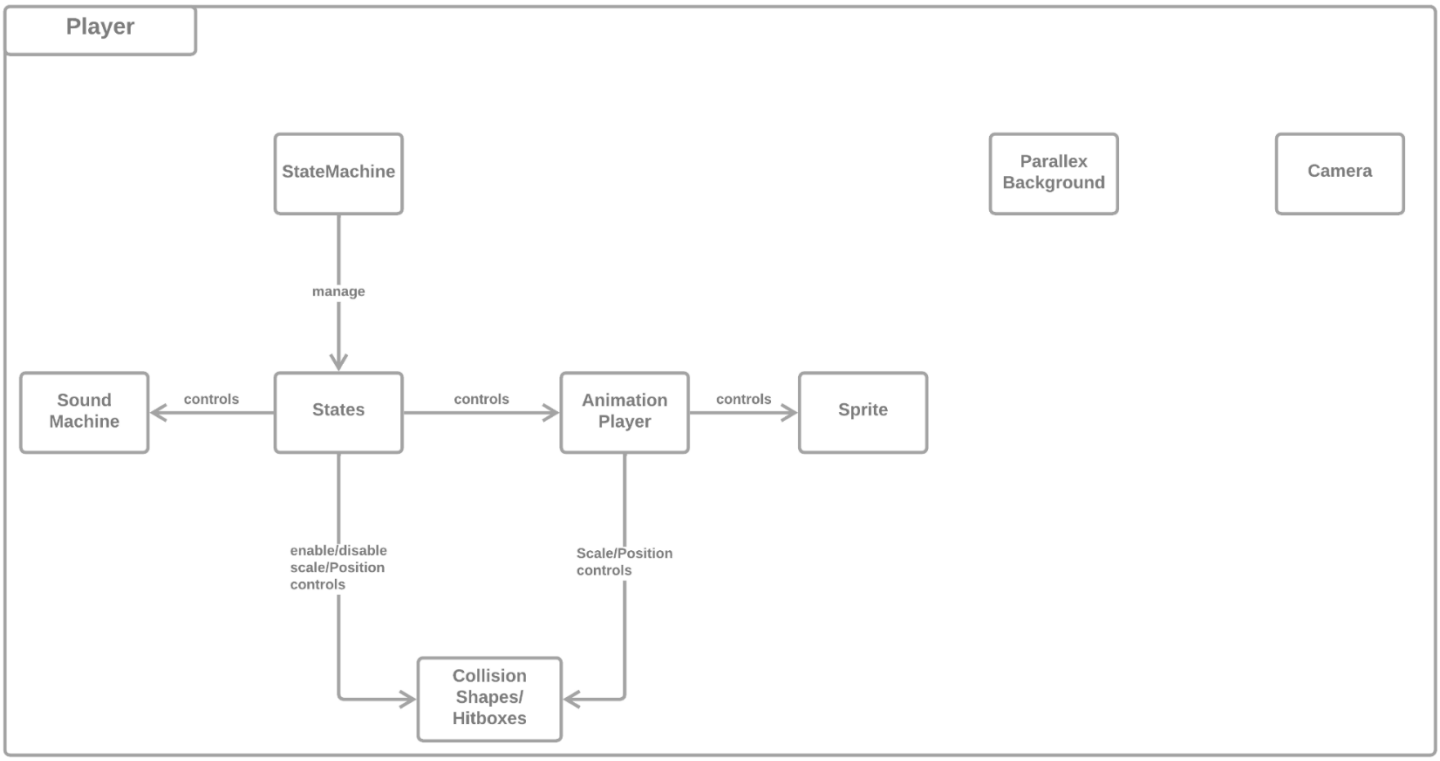
Ebene 1 - Gamestates



Ebene 2 - Level

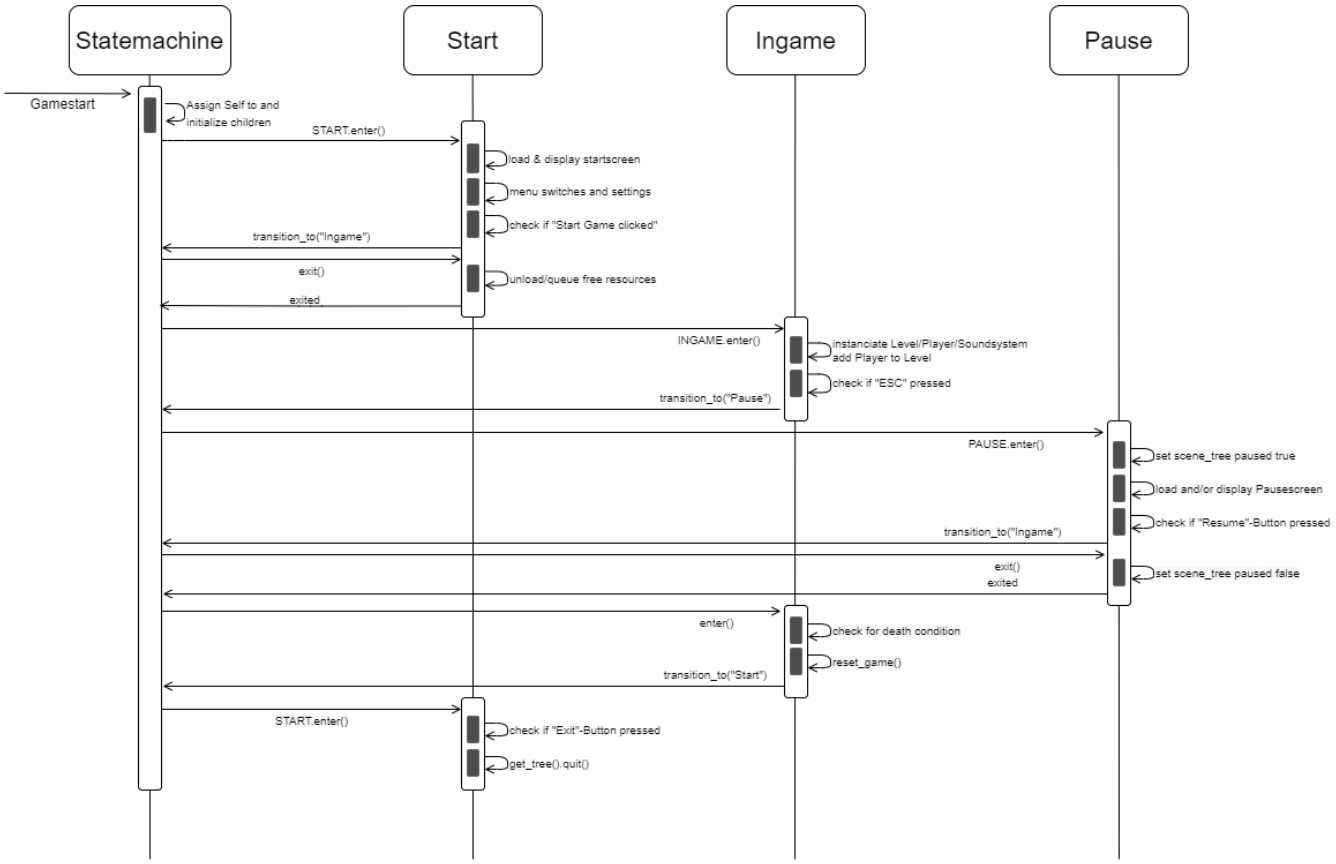


Ebene 3 - Player

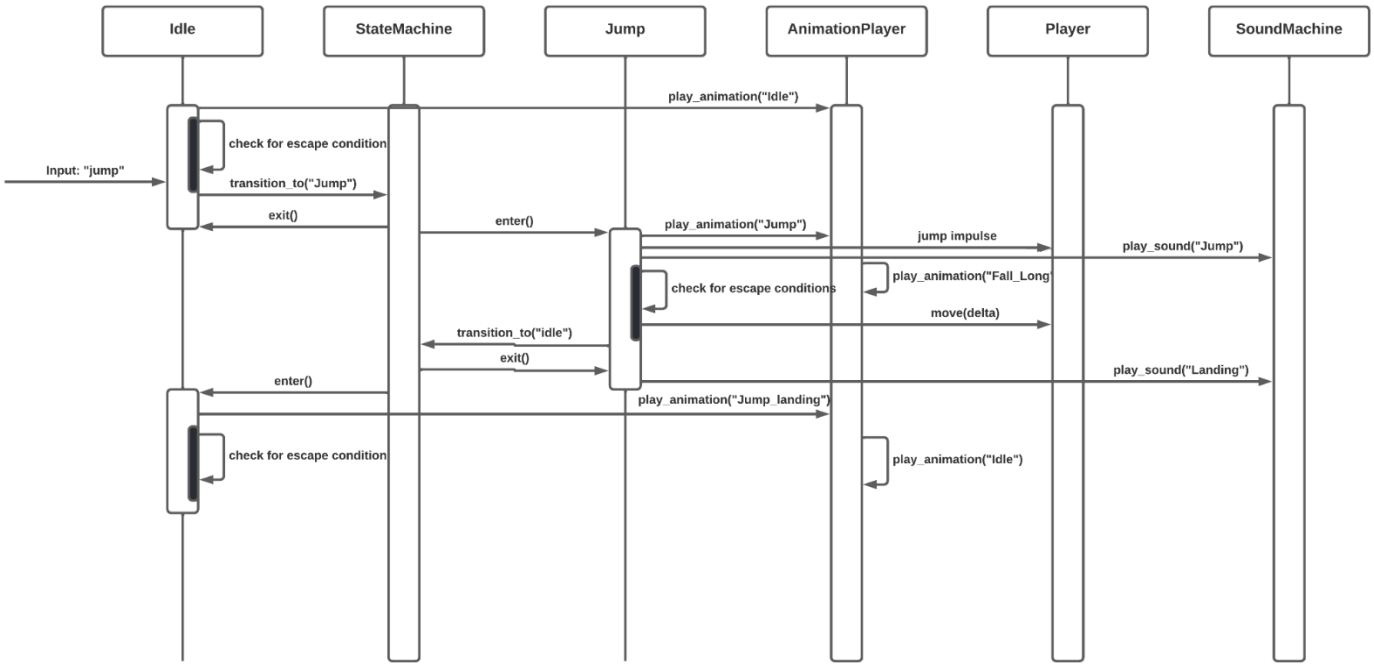


Laufzeitsicht

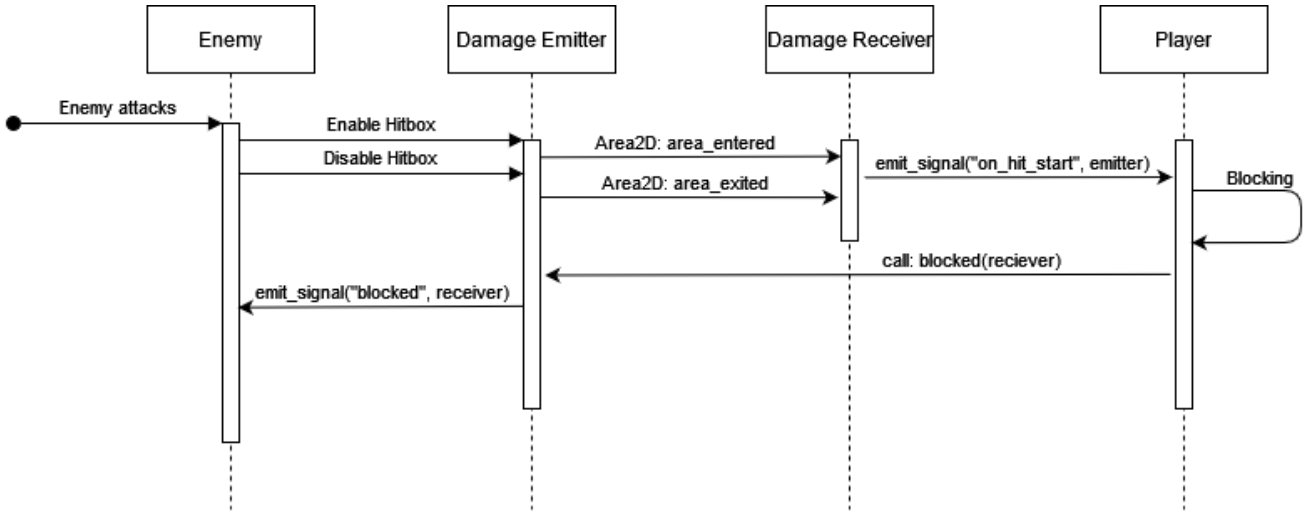
Szenario 1: Gameloop



Szenario 2: Player Jump



Szenario 3: Hitbox Signaling



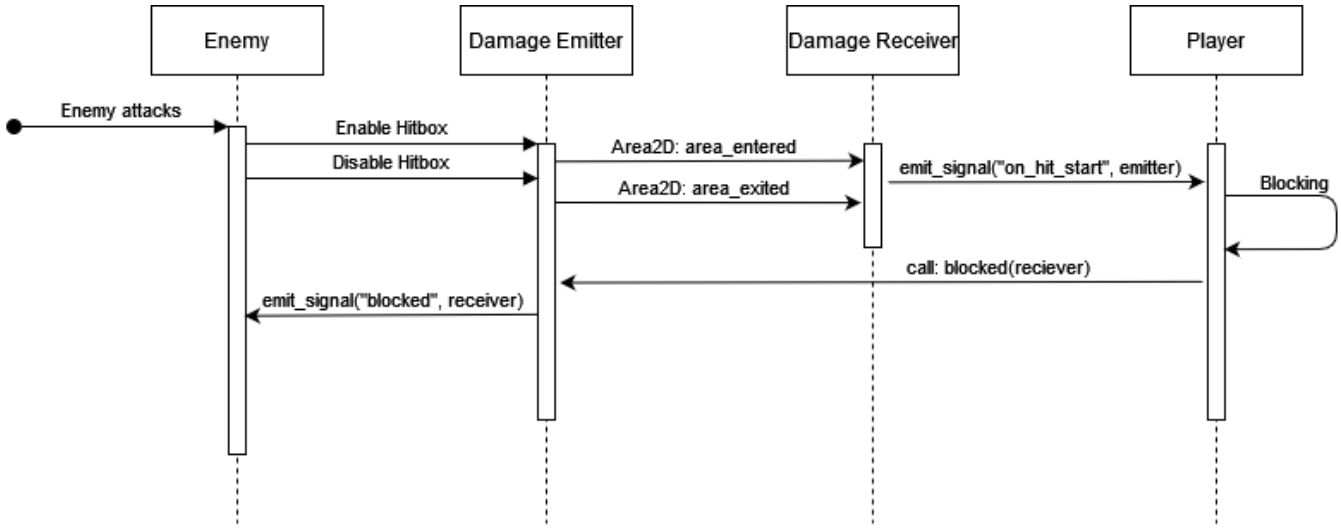
Querschnittliche Konzepte

Export Variablen

Beim Anlegen von Klassen wurde darauf geachtet, technische Details in Form von Export Variablen zu parametrisieren. Dies hat den Vorteil, dass bei der Entwicklung, als auch beim Balancing des Spiels, über den Godot-Editor Werte, wie bspw. die Schwerkraft, verändert werden können. So ist ein einfacher Arbeitsablauf auch für Personen ohne Programmiererfahrung gewährleistet.

Signaling

Beispielhaft lässt sich die Verwendung von Signalen gut an dem implementierten System aus Damage Emittern und Damage Receivern erläutern.
Um die Kommunikation zwischen Gegnern und Spielern zu realisieren wurden verschiedene Signale angelegt, die ausgesendet werden, wenn beispielsweise ein Damage Emitter mit der Area2D eines Damage Receivers kollidiert.
Im unteren Sequenzdiagramm ist der Ablauf eines erfolgreich geblockten Angriffs eines Gegners zu sehen. Der Damage Receiver des Players empfängt ein ***area_entered*** signal von Godot und prüft, ob dieses von einem Damage Emitter kommt.
Wenn ja, wird ein eigenes Signal ausgesendet: ***on_hit_start***. Mit diesem Signal wird der Emitter mitgesendet, mit dem der Receiver kollidierte.
Das Signal ***on_hit_start*** ist im Player verbunden mit entsprechenden Methoden, die Angriff und Blocken verarbeiten. In diesem Fall wurde erfolgreich geblockt und auf dem Damage Emitter des Gegners wird die Methode ***blocked*** gerufen, der der eigene Damage Receiver übergeben wird.
Der Damage Emitter sendet daraufhin das Signal ***blocked*** aus, und gibt diesem den Damage Receiver des Players mit.



Entwurfsentscheidungen

Architekturprinzipien und Design Patterns

Modularisierung

Das Spiel ist in einzelne Module unterteilt um zu gewährleisten, dass der Code übersichtlich und verständlich bleibt. Jede Klasse hat ihren Zuständigkeitsbereich in dem sie Variablen bearbeitet. So kümmert sich die Klasse Player zum Beispiel um die Inputs der Spielenden, die Bewegung des Spielers und den erlittenen Schaden. Zur kommunikation werden entweder direkt Methoden der Klasse aufgerufen oder die Klassen signalisieren das eintreten eines bestimmten Ereignisses untereinander.

Statemachine

Statemachines mit verschiedenen States finden sich im Projekt an vielen Stellen wieder. Im Gamecontroller, Player sowie Gegner. Die einmal implementierte Statemachine wird dabei immer wiederverwendet, während verschiedene States alle von der Klasse ***State*** erben.
Ein Statemachine Objekt ist immer einem ***Owner*** zugeordnet, dessen State sie verwaltet. Je nach aktuellem State kann dann das Verhalten sowie Austrittsbedingungen festgelegt werden. Diese Bedingungen sind meist an Userinputs oder Timer geknüpft. In jedem State ist klar definiert, in welchen State übergangen werden kann. Die ***exit*** Methode wird in jedem Fall beim Verlassen eines States aufgerufen, weshalb sie zum “aufräumen” verwendet wird.

Input Queue

Eine Input Queue wird beim Spieler für zwei Dinge verwendet. Zum Einen für die Bewegungssteuerung mit Tastatur und zum Anderen für Angriffe.

Bei der Bewegungssteuerung wird als Queue ein zweielementiges Array verwendet, in dem die Bewegungsrichtung immer am Anfang eingefügt wird. Kommt es vor, dass der Spieler nach Links und Rechts gleichzeitig steuert, dann bewegt sich der Charakter in die Richtung des letzen Tastendrucks.

Bei den Angriffen besteht die Queue aus einem vierelementigen Array. Hier werden die letzen kampfbestimmenden Eingaben gespeichert. Dies geschieht auf eine Weise, dass es möglich ist, eine Angriffskombo auszuführen, ohne im exakt richtigen Moment die entsprechende Eingabe zu geben. Die Queue wird bei Eingaben geleert, die einen Angriff abbrechen können.

Gegner und AI

Im Prototyp wurden zwei Gegnertypen implementiert. Während der ***Bird*** Enemy sehr simpel gehalten ist, ist der ***Puppet*** durch Trennung von Logik und Verhalten etwas komplexer und besser anpassbar.
Anfangs wurde auch für den Puppet ein simplerer Ansatz gewählt. Dies führte jedoch dazu, dass Anpassungen des Verhaltens ausgehend vom Game Design Team schwieriger umsetzbar waren, da für jede Veränderung eine unnötige Auseinandersetzung mit der Logik von Nöten war. Zudem ermöglicht es eine bessere Wiederverwendung des Codes.

Der ***PuppetCharacter***-Node bildet die Logik ab die durch simplen Input gesteuert werden kann. Das darüberliegende ***Puppet***-Node dient als Controller und liefert lediglich den passenden Input. Möglich wäre hier verschiedene Controller für verschiedene Verhaltensweisen einzubauen. Für den PuppetCharacter wurde ebenfalls wie bei dem Spieler eine Statemachine verwendet. Da in dem Spiel noch mehrere Gegner implementiert werden, bietet sich diese Lösung am besten an. Denn durch die wiederverwendbaren States können schnell Grundfunktionen implementiert werden (Laufen, Attackieren, Springen etc.). Der darüber liegende Puppet-Node löst die Implementierung der Verhaltensweise ebenfalls durch eine Statemachine mit jeweiligen ***Sub State Machines***. So kann der Puppet in den Basis-States: Combat, Patrol oder Alarmed sein und führt dann die dementsprechenden Sub State Machines aus. Auch hier lassen sich Grundfunktionen wiederverwenden (MoveForward, LookAround etc.)

Statemachine und Animationplayer

Animationen von komplexen Gameobjekten (Player, Enemies) werden durch die Statemachine gesteuert. Das Grundprinzip lautet: Jeder State hat eine gleichnamige Animation, welche beim Eintritt in den State abgespielt wird. Dies wird von den Basisklassen *EnemyState* und *PlayerState* implementiert.
Das ist allerdings nicht ganz ausreichend, da es sehr viele extra Fälle gibt, welche entweder im übergeordneten *AnimationPlayerController* oder in den einzelnen State Klassen bearbeitet werden.

Definition im AnimationPlayerController

Einige Animationen haben Animationen, welche davor oder danach abgespielt werden müssen, aber nicht in der gleichen Animation sind, da sie nicht mitgeloopt werden sollen. Beispielsweise gibt es eine extra Animation, wenn ein Character losrennt, bevor die eigentliche, geloopte Rennanimation gespielt wird. Solches Verhalten wird im *AnimationPlayerController* implementiert.

```
 9 func _ready():
10     animation_set_next("Run", "Run_loop")
11 
```

Definition in den einzelnen States

Oft ist eine Animation auch davon abhängig, aus welchem State der Charakter kommt. Dann ist die enter Methode der Basisklasse überschrieben und spielt abhängig des letzten States erst einmal eine Übergangsanimation ab. Im *AnimationPlayerController* ist dann wiederum der Übergang in die eigentliche Animation definiert.

```
 7 func enter(_msg := {}):
 8     if state_machine.last_state.name == "Jump" or state_machine.last_state.name == "Fall":
 9         animationPlayer.play("Jump_landing")
10     elif state_machine.last_state.name == "Dash" and player.is_on_floor():
11         animationPlayer.play("Decelerate")
12     elif state_machine.last_state.name == "Crouch":
13         animationPlayer.play("Crouch_End")
14     else:
15         .enter(_msg)
16         animationPlayer.set_speed_scale(animationPlayer.idle_speed)
```

Dauer eines States

Manche States sind abhängig von dem, was im Spiel passiert, beziehungsweise was der Spieler macht. Beispielsweise geht der Player aus dem Idle State, sobald er sich bewegt. Andere States halten für eine bestimmte zeit an, bzw für die Dauer einer Animation. Dafür kann aus der Länge einer Animation ein Timer generiert werden. Beim Timeout wird dann in den nächsten State übergegangen.

Qualitätsanforderungen

Qualitätsszenarien

Gameplay

Das Spiel “Celeste” ist ausgezeichnet durch Jump and Run Level mit angenehmer Steuerung. Es gilt, durch Mechaniken, wie einen Dash und Klettern Hindernisse zu überwinden.
Ziel, war es, dieses Spielgefühl nachzubilden. Es sollte ein Produkt entstehen, dessen Steuerung sich flüssig und gut anfühlt.
Zu diesem Zweck und dem Prinzip der kleinsten Überraschung folgend, verwendeten wir eine Button-Belegung am Controller, welche für viele Spiele üblich ist.
Wir implementierten eine Input-Queue, sodass sich die Bewegungsrichtung des Charakters intuitiv anfühlt, je nach Reihenfolge, der Tastenanschläge. Dies ist allerdings nur in Verwendung, bei Tastatursteuerung.
Des Weiteren sollte sich die Bewegung des Spielers flüssig anfühlen. Um dieses Ziel zu erreichen, wurde beispielsweise Beschleunigung bzw. bremsende Reibung in der Luft und am Boden implementiert. Dies vermittelt das Gefühl, dass der Charakter tatsächlich ein Objekt mit Masse ist.
Eine weitere Mechanik, die wir implementierten und die zu einem guten Spielerlebnis beiträgt ist die variable Sprunghöhe, je nachdem, wie lange der Knopf zum Springen gedrückt wurde.

Kampf

Bei den Kampfmechaniken wurde sich an dem Spiels “Whiskers and Wags” orientiert. Hier ist das Kampfsystem Kombo basiert, mit aufeinander aufbauenden Angriffen, sowie spezialangriffen.
Ziel war es, ein flüssiges Kampfsystem zu implementieren, mit dem es sich, ebenso, wie mit der Steuerung gut und flüssig spielen lässt.
Dazu implementierten wir, wie bei der Steuerung eine Input-Queue für Angriffe, sodass diese zuverlässig ausgeführt werden.
Es wurden, recovery Phasen nach Angriffen implementiert, um diese etwas zu schwächen und während Spezialangriffen die Steuerung gesperrt, sodass diese mit den Animationen eine gewisse Wucht bekamen.

Risiken und technische Schulden

Playtests

Playtests wurden fast ausschließlich von unserem Productowner durchgeführt. Das führt insgesamt zu einer sehr einseitigen Bewertung des Spielgefühls. Mit mehr Tests in verschiedenen Personengruppen, könnte ein objektiv besseres Spielgefühl erreicht werden. Das fertige Spiel im Gesamtlook mit allen Animationen und Mechaniken ausgiebig zu testen war durch die knappe Fertigstellung kaum möglich.

Gameplay

Um das Spielgefühl vollends abzurunden fehlen noch einige Dinge:
So wurde die Fallgeschwindigkeit nicht begrenzt, sodass der Charakter bei längeren Fällen sehr schnell wird. Dies erschwert manche Jump and Run Passagen.

Zusätzlich hätte noch implementiert werden können, dass es dem Spieler noch möglich ist zu springen, kurz nachdem er/sie von einer Plattform runtergelaufen ist. Dies ist zwar physikalisch nicht korrekt, verbessert allerdings das Spielgefühl.

Des Weiteren sind die Hitboxen noch nicht optimal an Objekte angepasst. Beispielsweise sind die Hitboxen der Stacheln viereckig, obwohl eine Dreiecksform durch die Textur vorgegeben ist. An anderen Stellen sind wie fast zu gut angepasst, so fällt es z.B. unnötig schwer den Dashresetter zu treffen.

Bugs

Seit dem Fertigstellen des Spiels für die Medianight sind zwei Bugs aufgefallen. Zum Einen ist es möglich unter Spikes durchzugelangen, für die eine Kombination aus Dash und Crouchslide angedacht war. Möglich ist es durch kurzes loslassen der Crouchtaste mit gleichzeitigem vorwärts laufen. Ursache für den Bug ist ein zu spätes wechseln der Form der Hitbox des Spielers von der Crouch-Form zu der Standardform.

Der zweite Bug hat zur Folge, dass der Spieler durch den Boden fällt, indem er/sie croucht und dann schnell hintereinander springt und angreift. Ursache für den Bug ist vermutlich die Mechanik, dass es möglich ist, durch dünne Plattformen zu fallen. Hier wird die Hitbox des Spielers kurz deaktiviert. Ein erneutes aktivieren der Hitbox des Spielers wird vermutlich durch einen Angriff unterbunden.

Tutorial

Noch fehlt ein richtiges Tutorial. Neuen Spielern fehlt die Idee, an welchen Stellen mit welchen Mechaniken gearbeitet werden muss. Manche Mechaniken werden dadurch vielleicht überhaupt nie verwendet.

Lessons learned

Gemeinsames Arbeiten

Dass einzelne Feature Branches möglichst schnell und ohne umfangreiche Merge Konflikte auf den Main Branch gepusht werden können, ist es wichtig die feature Branches möglichst aktuell zu halten. Das funktioniert am besten, wenn kleinere Arbeitspakete schnell abgearbeitet und auf den Main gepusht werden können. Um alles aktueller zu halten, hat uns auch der Development Branch sehr geholfen.

Zusammenspiel von Assets und Code

Am Anfang dachten wir, dass wir Programmierer recht unabhängig von den Artists sind. Mit der Zeit fiel uns auf, dass eine enge Absprache mit den Artists doch sehr wichtig ist. Wo sollen die Assets hin? Wann müssen sie da sein? Welches Format wird gebraucht? Was für Übergangsanimationen werden noch gebraucht?

Durch schlechte Absprachen mussten Assets einige Male verbessert werden und generell hat sich alles etwas verzögert.

“Erst alles Programmieren, dann die Assets einfügen und fertig”, hat leider auch nicht ganz geklappt. Da wir alle noch nicht so viel Erfahrung haben, sind uns viele Dinge auch erst später aufgefallen und mussten dann nochmal geändert werden.

Scrum

Zur Projektorganisation verwendeten wir Scrum mit recht strikten Regeln, wie beispielsweise, wer zum Meeting zu spät kommt, sei es eine Minute, darf nicht aktiv teilnehmen. Das hatte zur Folge, dass andere nicht auf dem neuesten Stand waren, mit den eigenen Fortschritten.
Im Laufe des Projekts kristallisierte sich langsam heraus, dass es Sinn ergibt, die Projektorganisation an das Projekt anzupassen, und nicht umgekehrt. Wir lockerterten einige Regeln und so wurde die Zusammenarbeit wesentlich angenehmer.

Game Entwicklung

Für die meisten von uns, war dieses Projekt das erste richtige Game-Projekt, bei dem die volle Funktionalität einer Gameengine ausgenutzt wurde. So lernten wir beispielsweise, wie eine Statemachine, wie Charakterbewegung und wie der Aufbau von Leveln funktioniert.

Auch der Weg der Implementierung von einem Prototypen auf dem Papier zu einem Computerspiel gingen viele das erste Mal.

Repository

Das zu bewertende Projekt zum Stand des 17.02.2022 befindet sich auf dem Branch **MI-Abgabe**.
https://gitlab.mi.hdm-stuttgart.de/epiales-stupro-ws21-22/Epiales-Code/-/tree/MI-Abgabe

Fertiges Builds befinden sich im Ordner **Exe** für macOS und Windows.
https://gitlab.mi.hdm-stuttgart.de/epiales-stupro-ws21-22/Epiales-Code/-/tree/MI-Abgabe/Exe