Kursarbeit Informatik 12

Prototyp Platformer

Moritz Reiche, Marie Panzer

Abgabe: 19.02.2026

Kursarbeit Informatik Klasse 12

Platformer

Projektbeschreibung

Das Ziel dieses Informatikprojekts war die Entwicklung eines funktionalen 2D-Platformers im Metroidvania-Stil. Zu Beginn erarbeiteten wir ein gemeinsames Konzept, das die Spielmechaniken und das visuelle Design im Pixel-Art-Stil festlegte. Dabei orientierten wir uns für den größten Teil der technischen Umsetzung an der YouTube-Tutorialserie „Metroidvania Forge“ von Michael Games (siehe Quellenverzeichnis), brachten jedoch maßgeblich eigene Ideen und individuelle Einflüsse in das Spieldesign sowie die Ausgestaltung ein. Als technische Basis dienten die Godot Engine für die Programmierung sowie Aseprite für die grafische Gestaltung. Um eine effiziente Zusammenarbeit zu gewährleisten, nutzten wir GitHub als zentrale Plattform für die Versionsverwaltung, wobei wir zur Vereinfachung der Arbeitsabläufe die grafische Benutzeroberfläche von GitHub verwendeten.

Aufgabenverteilung und Umsetzung

Moritz übernahm die softwareseitige Architektur des Spiels. Er setzte das Grundprojekt in Godot auf und implementierte eine Finite State Machine zur Steuerung des Spieler-Charakters. Diese verwaltet die logischen Zustände *Idle*, *Run*, *Jump* und *Fall* und sorgt für eine präzise Trennung der Bewegungsabläufe. Zudem erstellte er funktionale Platzhalter für die erste Testphase und implementierte Game-Design-Techniken wie Coyote-Time sowie einen Jump Buffer. Ebenfalls fügte er Debug-Funktionen wie ein Player-State-Label und Jump-Debug-Indikatoren hinzu.

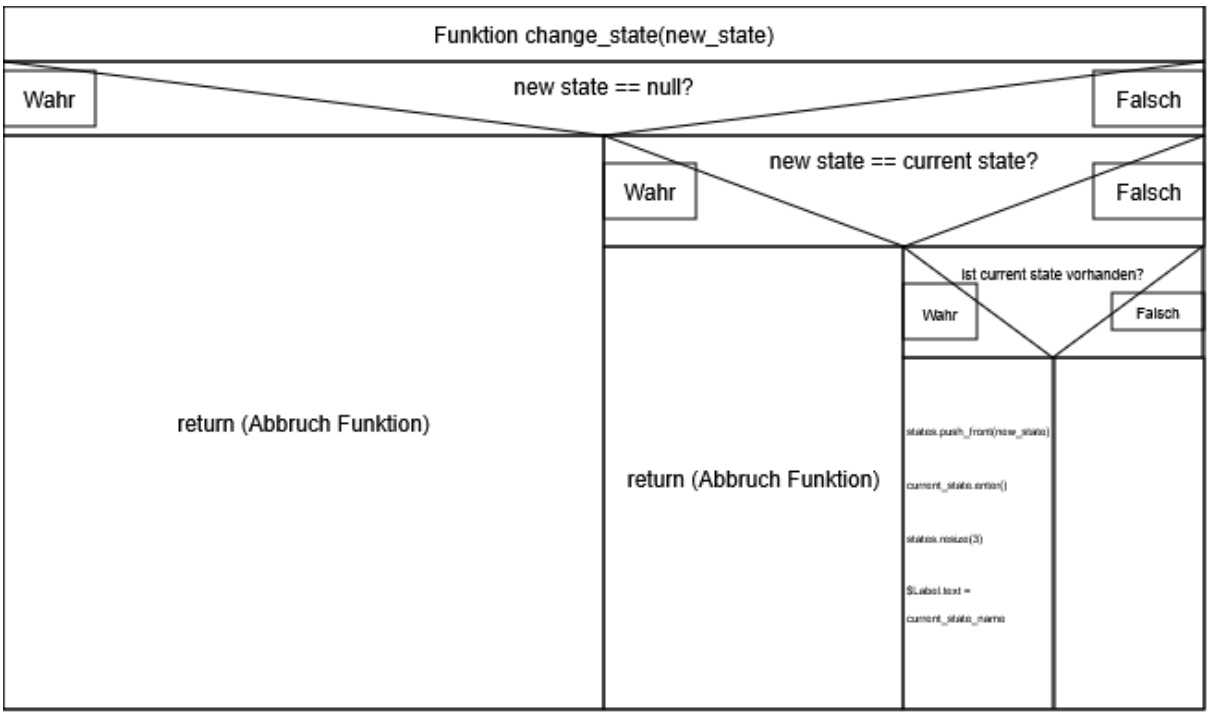
Marie war für die gesamte grafische Identität des Spiels verantwortlich. Sie entwarf das Charakter-Design und erstellte die dazugehörigen Animationen in Aseprite, wobei sie die Frames exakt auf die programmierten Zustände der State Machine abstimmte. Darüber hinaus entwickelte sie modulare Tile Sets für die Spielumgebung, um eine konsistente und erweiterbare Welt zu schaffen.

Kollaboration und Abschluss

In der finalen Phase arbeiteten wir kollaborativ an der Gestaltung der Spielwelt. Unter Verwendung der Tilesets bauten wir gemeinsam die Tilemap in Godot auf. Durch die Nutzung der grafischen Oberfläche von GitHub konnten wir unsere jeweiligen Fortschritte – sowohl Code-Änderungen als auch neue Grafik-Assets – synchronisieren, Konflikte vermeiden und den aktuellen Projektstand jederzeit für beide Teammitglieder verfügbar halten.

Theoretische Grundlagen der Programmorganisation

Das folgende Struktogramm stellt die Funktion change_state aus dem nachfolgenden Quelltext der Datei **player.gd** dar. Die .xml-Datei (Drawio) ist im GitHub-Repository enthalten.



Quelltext

Das Skript **player.gd** ist im GitHub-Repository enthalten und sollte der Lesbarkeit wegen in einer Entwicklungsumgebung (bereits im Godot-Projekt unter player->scripts enthalten) bzw. einem Codeeditor geöffnet werden.

```
class_name Player extends CharacterBody2D # erweitert/baut auf bereits existierende klasse

const DEBUG_JUMP = preload("uid://d3fxe3317dcqp")

#region export-variablen
@export var move_speed: float = 150
@onready var animation_player: AnimationPlayer = $AnimationPlayer
@onready var sprite: Sprite2D = $Sprite2D
#endregion
```

```
#region state machine variablen
var states : Array[PlayerState]
var current_state : PlayerState :
    get : return states[0]
var previous_state : PlayerState :
    get : return states[1]
#endregion

#region standart-variablen
var direction : Vector2 = Vector2(0, 0)
var gravity : float = 981
var gravity_mult : float = 1.0
#endregion


func _ready() -> void:
    initialize_states() # funktion: alle states initiieren
    pass

func _unhandled_input(event: InputEvent) -> void:
    change_state(current_state.handle_input(event))

func _process(_delta: float) -> void: #erstellen "dauerschleife"
    update_direction()
    change_state(current_state.process(_delta))
    pass

func _physics_process(delta: float) -> void: #erstellen physik-"dauerschleife"
    velocity.y += gravity * delta * gravity_mult
    move_and_slide()
    change_state(current_state.physics_process(delta))
    pass

func initialize_states() -> void:
    states = []
    for i in $States.get_children(): #alle states sammeln
        if i is PlayerState:
            states.append(i)
            i.player = self
        pass
    if states.size() == 0: # falls state versagt,  funktion beenden
        return
    for state in states:# alle states initiieren
        state.init()
    change_state(current_state) # ersten state setzen
    current_state.enter()
    $Label.text = current_state.name
```

```
    pass

func change_state(new_state : PlayerState) -> void: # state ändern
    if new_state == null: # .. nicht wenn neuer state nicht vorhanden
        return
    elif new_state == current_state: # .. nicht wenn neuer gleich aktueller state
        return
    if current_state: # aktuellen state beenden
        current_state.exit()

    states.push_front(new_state) # state zu neuem ändern, alte nach hinten verschieben
    current_state.enter()
    states.resize(3) # nur letzten 3 vergangenen states in liste speichern
    $Label.text = current_state.name # debug-label-text auf aktuellen state setzen
    pass
func update_direction() ->  void: # bewegungsvektor richtung ändern
    var prev_direction : Vector2 = direction
    # fix deadzones controller
    var x_axis = Input.get_axis("left", "right")
    var y_axis = Input.get_axis("up", "down")
    direction = Vector2(x_axis, y_axis)

    if prev_direction.x != direction.x:
        if direction.x < 0:
            sprite.flip_h = true # wenn nach links bewegen, sprite horizontal flippen
        elif direction.x > 0: # wenn nach rechts bewegen, zurück flippen
            sprite.flip_h = false
    pass



func add_debug_ind(color : Color = Color.RED) -> void:
    var d : Node2D = DEBUG_JUMP.instantiate() # debug-jump-szene kopieren und in
playground auf player platzieren
    get_tree().root.add_child(d)
    d.global_position = global_position
    d.modulate = color
    await get_tree().create_timer(3).timeout
    d.queue_free()
    pass
```

Ergänzungen und Erweiterungen

Aufgrund des zeitlichen Rahmens konnten einige fortgeschrittene Mechaniken des ursprünglichen Konzepts nicht mehr vollständig implementiert werden. Diese waren jedoch bereits detailliert geplant und bilden die Basis für eine potenzielle Weiterentwicklung des Spiels:

- Erweiterung der State Machine: Die Steuerung sollte um zusätzliche Zustände wie "Attack" (Nahkampf mit dem Degen), "Shoot" (Fernkampf mit dem Revolver), sowie Bewegungs-Upgrades wie Double Jump und Dash erweitert werden
- User Interface (UI): Für das Ressourcenmanagement sollte ein HUD (Heads-Up Display) integriert werden, das dem Spieler aktuelle Werte zu Lebenspunkten, Ausdauer und der verbleibenden Revolver-Munition anzeigt
- Loot- & Upgrade-System: In der Spielwelt sollten Kisten platziert werden, die dem Spieler neues Gear oder Waffenupgrades gewähren. Zudem waren Pick-up-Items vorgesehen, die temporäre oder permanente Statusveränderungen bewirken (z. B. Erhöhung von Stärke, Schaden, Schnelligkeit oder Sprungkraft)
- Level-Progression: Das Spiel sollte über mehrere miteinander verbundene Level verfügen
- Kampfsystem & Gegner-KI: Geplant war die Implementierung verschiedener Gegnertypen mit eigenen Verhaltensmustern sowie eines finalen Endbosses, um den Schwierigkeitsgrad des Metroidvanias abzurunden. Bei der Planung der Gegner und eines möglichen Endbosses stießen wir auf technische Herausforderungen, die im Rahmen des aktuellen Zeitplans und unseres damaligen Kenntnisstandes über unserem Umsetzungsniveau lagen. Insbesondere die Gestaltung abwechslungsreicher Gegnertypen sowie die Implementierung einer funktionalen KI, die dynamisch auf die Bewegungen des Spielers reagiert, stellte sich als hochkomplex dar, weshalb wir diese Idee schließlich verworfen haben
- Hauptmenü und Navigationsstruktur: Zur Verbesserung der User Experience (UX) war die Implementierung eines zentralen Hauptmenüs geplant. Dieses sollte als Einstiegspunkt dienen, um zwischen dem eigentlichen Spielgeschehen, einem Einstellungsmenü und dem Beenden der Applikation zu navigieren. Technisch wäre dies durch eine übergeordnete Szenenverwaltung realisiert worden, die den Wechsel zwischen verschiedenen Spielumgebungen (Scenes) steuert.
- Persistenz-System (Save/Load-Funktion): Um den Spielfortschritt über eine einzelne Sitzung hinaus zu sichern, sollte ein System zur Datenspeicherung integriert werden. Dabei würden relevante Variablen wie die aktuelle Spielerposition, erreichte Checkpoints oder gesammelte Ressourcen in einer externen Datei (z. B. im JSON- oder ConfigFile-Format) serialisiert. Beim Laden des Spiels könnten diese Daten ausgelesen werden, um den exakten Spielzustand wiederherzustellen.

Bedienungsanleitung

Diese Anleitung führt durch den Download, die Installation und die Steuerung des Prototypen.

5.1 Bezug des Projekts über GitHub

Um auf den Quellcode und die ausführbaren Dateien zuzugreifen, folgen Sie diesen Schritten:

1. Rufen Sie das GitHub-Repository über den bereitgestellten Link (https://github.com/ZombieMR/info-kursarbeit-metroidvania) auf.
2. Klicken Sie auf die grüne Schaltfläche **„Code“**.
3. Wählen Sie **„Download ZIP“** aus und entpacken Sie das Archiv nach dem Download auf Ihrem lokalen Rechner.
    - *Hinweis:* Alternativ kann das Repository über den Befehl git clone https://github.com/ZombieMR/info-kursarbeit-metroidvania geklont werden.

5.2 Ausführen des Prototypen (.exe)

Der Prototyp wurde für Windows vorkompiliert, damit er ohne Installation der Engine getestet werden kann.

1. Navigieren Sie im entpackten Ordner zum Unterverzeichnis des Exports abgabe/ Stellen Sie sicher, dass die Datei Informatik Kursarbeit Prototyp PanzerReiche.exe und die dazugehörige Ressourcen-Datei Informatik Kursarbeit Prototyp PanzerReiche.pck im selben Ordner liegen.
2. Starten Sie das Spiel durch einen Doppelklick auf die **.exe-Datei**.

5.3 Projekt im Godot-Editor öffnen

Um den Quellcode oder die Szenenstruktur live einzusehen:

1. **Godot Engine herunterladen:** Besuchen Sie **godotengine.org** und laden Sie die Version von **Godot 4.5.1** herunter (keine Installation notwendig, einfach die .exe ausführen).
2. **Projekt importieren:**
    - Starten Sie Godot und klicken Sie im Projekt-Manager auf **„Importieren“**.
    - Navigieren Sie zum entpackten GitHub-Ordner und wählen Sie die Datei **project.godot** aus.
    - Bestätigen Sie mit **„Importieren & Editieren“**.
3. **Starten:** Drücken Sie innerhalb des Editors die Taste **F5**, um das Projekt zu starten. *Um die Testumgebung, die während des Programmierens verwendet wurde, zu starten, navigieren Sie zu der Szene **playground.tscn** und drücken Sie die Taste **F5**.*

5.4 Steuerung des Spielers

Der Prototyp unterstützt sowohl die Tastatureingabe als auch die Steuerung per Gamepad (Plug-and-Play).

Aktion	**Tastatur**	**Gamepad (Xbox/Generic)**
Laufen (Links/Rechts)	Pfeiltasten (Links/Rechts)	Linker Analog-Stick / Steuerkreuz
Springen	Leertaste (Space) oder „Y“	Taste „A“ (Gamepad-Index 0)

Quellenverzeichnis

- https://youtube.com/playlist?list=PLfcCiyd_V9GFL_xF8ID9vIt5bs0NJI4EK&si=CSM_UYZISaka8vWD
- https://youtu.be/MDEWKaAhyRk?si=ikGyi4E_ud5SlvPq