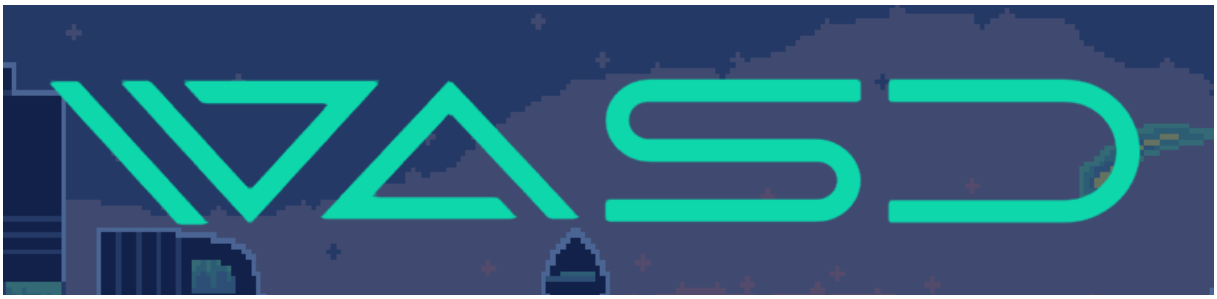


HOCHSCHULE
DER MEDIEN

Projektdokumentation

Software-Entwicklung 3 - 11330

M.Sc. Tobias Jordine

Gruppe 8 (WASD):

André Schwabauer (as439)
Manith Mam (mm334)
Kevin Cipric (kc028)
Kimberly Köhnke
Tim Breunig (tb
Alexander Schrag (as436)
Rometh Umic-Senol (ru006)

https://github.com/Andrushka130/WASD

Inhalt

Vorwort

Für unser Projekt in Softwarentwicklung 3 hatten wir die Idee ein simples Videospiel zu entwickeln. Anfangs wollten wir ein Browsergame mithilfe von Frameworks wie „Phaser “programmieren. Jedoch haben wir uns nach gemeinsamen Überlegungen und Empfehlung von Tutoren entschieden eine richtige Videospielengine zu nutzen. Entschieden haben wir uns für „Unity“. Die Idee für das Spiel hatten wir schon von Anfang an festgelegt. Wir wollten ein Spiel im Stil von „Vampire Survivors“ und „Brotato“ entwickeln, da sie uns recht simpel und unterhaltsam erschienen.

Ziel

Unser Hauptziel war es den Core Gameplay Loop dieser Spiele zu replizieren. Sprich, der Spieler besiegt eine unendliche Welle von Gegnern, welche immer stärker werden, während der Spieler gleichzeitig stärker wird. Der Spieler wird durch den Kauf von zufälligen Waffen und Items und dem Leveln von Attributen in einem Shop nach jeder Welle stärker. Geplant war es mehrere Charaktere zu entwickeln mit ihren eigenen einzigartigen Startwaffen. Die Anzahl an besiegten Wellen stellt hier den Highscore dar. Dieser soll mithilfe einer Datenbank gespeichert werden. Spieler können sich einloggen, um ihren Highscore zu speichern. Die Daten sollen im Hauptmenü für alle eingeloggten Spieler sichtbar sein.

Technologien

Unity

Bei Unity handelt es sich um eine Game Engine. Es ist die weltweit führende Plattform für interaktive Echtzeit-3D-Inhalte. Mit Unity hat man einen Werkzeugkasten um 2D, 3D und auch 2,5D Spiele erstellen zu können.

Neben Unity gibt es auch andere Game Engines. Unity gehört wie Unreal Engine 5 zu den größten Game Engines. Zudem ist Unity, wie auch Godot, sehr Einsteigerfreundlich. Die Kombination aus diesen beiden Aspekten hat dazu geführt, dass wir uns für Unity entschieden haben. Außerdem gibt es für Unity sehr viele Lernmöglichkeiten und Tutorials, sowohl auf der Webseite von Unity selbst als auch auf YouTube.

MonoBehaviour

MonoBehaviour ist die Basisklasse, von der jedes Unity-Skript abgeleitet ist.

Wenn man C# verwendet, muss man explizit von MonoBehaviour ableiten. Wenn ein UnityScript (eine Art von JavaScript) verwenden wird, muss nicht explizit von MonoBehaviour abgeleitet werden.

S.O.L.I.D. bei Unity

Single Responsibility Principle:

- Ein GameObject kann mehrere Skripte, die die einzelnen Klassen darstellen, enthalten. Dadurch kann man jede einzelne Funktion die das GameObject hat von einer einzelnen Klasse ausführen lassen. Die einzelnen Skripte können in anderen GameObjects wiederverwendet werden, die die gleiche Funktion verwenden sollen.

Open-Closed Principle:

- Einzelne Klassen oder Funktionen können erweitert werden durch Interfaces, Vererbung oder die Verwendung von Scriptable Objekct. Scriptable Objects sind eine Unity spezifische Funktion, mit der man ähnliche Game Objects nach Vorlage einer Überklasse erstellen kann.

Liskov Substitution Principle:

- Unity erlaubt es sehr einfach mit Vererbungs- und Interface-Hierarchien zu arbeiten.

Interface Segregation Principle:

- In einem GameObject werden alle Funktionen in Form von Skripten zusammengeführt. Die einzelnen Skripte haben nur eine Schnittstelle zu dem jeweiligen GameObject und haben dadurch keinerlei Abhängigkeit untereinander. Eine Veränderung in einer einzelnen Klasse führt direkt zu einer konkreten Veränderung der Funktion des GameObjects.

Dependency Inversion Principle:

- Die Objekte, die die einzelnen Funktionen ausführen, erhalten ihre Anweisungen von einzelnen Klassen. Diese Klassen werden vom Objekt ausgerufen. Die Klasse selbst ist vom Objekt unabhängig und Veränderungen am Objekt haben keine Auswirkungen auf die Klasse.

DockerDesktop

Docker Desktop ist eine für Mac-, Linux- oder Windows-Umgebungen nutzbare Anwendung, mit welcher sich containerisierte Anwendungen und Microservices erstellen und gemeinsam nutzen lassen. Es bietet eine unkomplizierte grafische Benutzeroberfläche, mit der Sie Ihre Container, Anwendungen und Images direkt von Ihrem Rechner aus verwalten können. Docker Desktop kann entweder eigenständig oder als ergänzendes Tool zum CLI verwendet werden. Docker Desktop reduziert den Zeitaufwand für komplexe Setups, sodass Sie sich auf das Schreiben von Code konzentrieren können. Es kümmert sich um Port-Zuordnungen, Dateisystembelange und andere Standardeinstellungen und wird regelmäßig mit Fehlerbehebungen und Sicherheitsupdates aktualisiert. Wir nutzen Docker Desktop, um MongoDB einfach und schnell installieren und ausführen zu können.

MongoDB

MongoDB ist ein dokumentenorientiertes **NoSQL** Datenbankmanagementsystem, welches Sammlungen von **JSON**-ähnlichen Dokumenten verwalten kann. So können viele Anwendungen Daten auf natürlichere Weise modellieren, da die Daten zwar in komplexen Hierarchien verschachtelt werden können, dabei aber immer abfragbar und indizierbar bleiben. Wir nutzten **MongoDB** in Form eines von Docker offiziell bereitgestellten **Images** und kommunizierten mit Hilfe des **MongoDB Drivers** für **Node.js** mit der Datenbank.

Node.js und Express.js

Node.js ist eine plattformübergreifende **Open-Source-JavaScript-Laufzeitumgebung**, welche es ermöglicht **JavaScript-Code** außerhalb eines Webbrowsers ausführen zu können. **Node.js** nutzt asynchrone Programmierung, um das Warten wegen eines File Request überspringen zu können und einfach mit der nächsten Anfrage fortfahren zu können, was ideal für Serveranwendungen ist. Express.js erweitert dabei Node.js um Werkzeuge, mit welchen das Entwickeln moderner Webanwendungen einfacher gestaltet wird. Wir nutzten **Node.js** um mit Hilfe des **Node.js Frameworks Express.js** für Webanwendungen einen simple Webserver aufsetzen zu können.

NUnit

Zum Schreiben unserer Tests haben wir **NUnit** verwendet. **NUnit** ist ein Framework, mit dem man Unit Test für alle Net.-Programmiersprachen schreiben kann. Es wurde ursprünglich aus **JUnit** portiert.

GitHub Desktop

Github Desktop ist eine Anwendung, die dafür sorgt, dass man eine GUI anstatt Command Lines benutzen kann, um mit Github zu arbeiten. Außerdem vereinfacht es das lokale arbeiten des Projekts.

GameCI

GameCI ist eine Community von Videospielentwickler, die sich mit dem Automatisieren von Prozessen für Videospiele sowie mit der Aufrechterhaltung von Videospielen beschäftigen. **GameCI** bietet OpenSource-Tools für **CI/CD** Prozesse. Diese Tools werden weltweit von Entwicklern verwendet.

Sie erleichtern den Zugang zu **CI/CD** für Video - spielentwickler. In unserem Projekt unter *.github/workflows/template.yml* verwenden wir Tools von **GameCI** für unsere Pipeline.

Hier finden Sie einen Job, der das Tool *game-ci/unity-test-runner@v2* verwendet. Dieser Job bewirkt, dass wir unsere geschriebenen Tests in **EditMode** und **PlayMode** (*EditMode*: bevor das Spiel startet, *PlayMode*: während des Spiels) durchlaufen lassen können.

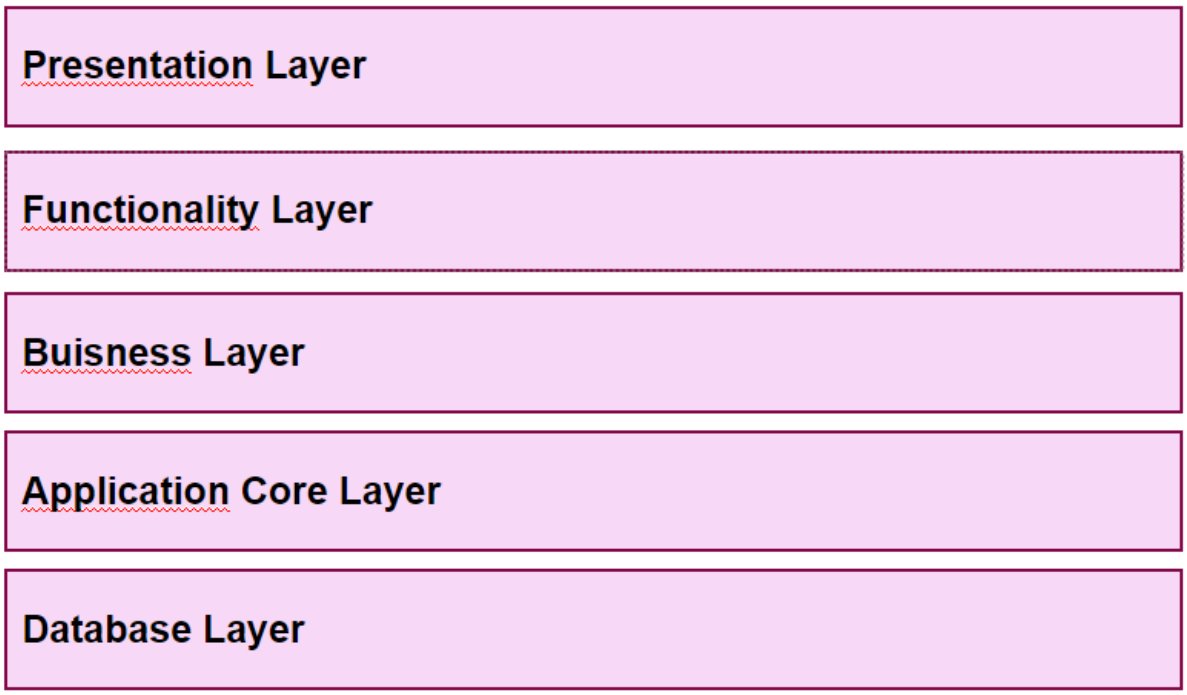
```
uses: game-ci/unity-test-runner@v2
id: testRunner
with:
  projectPath: ${{ matrix.projectPath }}
  testMode: all
  githubToken: ${{ secrets.GITHUB_TOKEN }}
  checkName: ${{ matrix.testMode }} Test Results
```

Hier finden Sie die Webseite von **GameCI**: https://game.ci

Architektur und Prinzipien

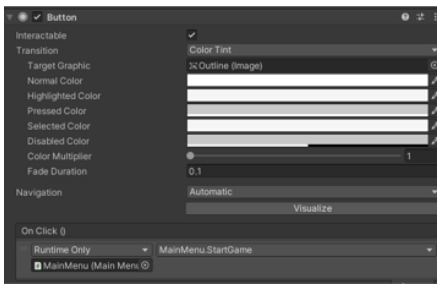
Architektur

Im Projekt wird die Layer-Architektur verwendet.



Presentation Layer: die Gui

Funktionality Layer: die bearbeitbaren Attribute



Buisness Layer: der Code und das Verhalten der Spielobjekte

Applicaton Core Layer/Persistenz Layer: die Schnittstelle zwsichen Backend und Spiel

Database Layer: die MongoDB

Prinzipien

SOLID Prinzipien in unserem Projekt

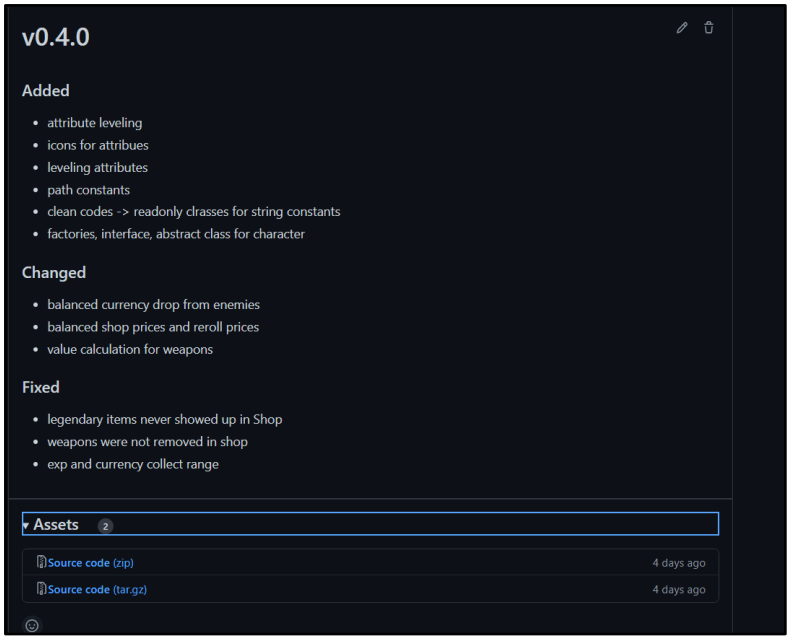
Wir haben in unserem Projekt darauf geachtet, dass einzelne Klassen nur spezifische Funktionen ausführen und nicht für mehrere Funktionen verantwortlich sind. Die Funktionen, die für Objekte benötigt werden, wurden dann in einem Manager oder in dem Objekt selbst zusammengeführt.

Funktionen, die wir in verschiedenen ähnlichen Klassen brauchen, haben wir in Interfaces oder mittels Vererbung zusammengefasst. Um unseren Code übersichtlich zu gestalten haben wir Vererbungs- und Interface-Hierarchien verwendet. Zudem haben wir die Funktion der Scriptable Objects von Unity verwendet.

Objekte, die das Spielgeschehen beeinflussen haben eine Schnittstelle, in Form einer Verknüpfung, zu den einzelnen Klassen. Die einzelnen Klassen werden durch die Eingabe des Spielers oder Vorkommnisse im Spiel aufgerufen und ausgeführt. Die Klasse selbst wird dabei nicht verändert.

Installation

Im GitHub Repository gelangt man über den Button “Releases” zur Übersicht aller Releases. Anschließend klickt man auf den neuesten Release. Unter “Assets”



unterhalb des erschienenen Changelogs kann der Source Code des Projekts als .zip Datei heruntergeladen werden. Die Datei wird anschließend entpackt.

Als nächstes benötigt man Unity. UnityHub wird über folgenden Link heruntergeladen und installiert: https://unity.com/de/download.

Nun kann UnityHub gestartet werden und nach Einrichten des Accounts mit der nötigen Lizenz wird über den “Open” Button unter dem Bereich Projects zum entpackten Projekt Ordner auf der Festplatte navigiert und die enthaltene WASD Datei geöffnet. Da noch keine UnityEditor Version installiert ist, schlägt UnityHub die für das Projekt nötige Version zum direkten Installieren vor. Nach dem Klick auf Installieren wird nach der Installation zusätzlicher Module gefragt, diese können ignoriert werden und die Installation kann fortgesetzt werden. Nach Abschluss der Installation kann das Projekt WASD unter Projects per Doppelklick geöffnet werden.

Anleitung

Das Grundkonzept von WASD dreht sich rund um eine möglichst einfache Steuerung. Im Folgenden sind alle wichtigen Tasten- und Mausbefehle mit Erklärung aufgelistet, die der Spieler benötigt.

Gameplay

W	nach oben bewegen
A	nach links bewegen
S	nach unten bewegen
D	nach rechts bewegen
Maus	auf Gegner zielen
Esc	Pause / Menü öffnen und schließen

Shop

Im Shop erfolgt die Steuerung ausschließlich über die Maustasten.

- Linksklick auf den **BUY-Button**, um ein Item zu kaufen
- Linksklick auf **PLUS-Buttons**, um Attribute zu erhöhen. Dafür werden die Attributpunkte verwendet, die durch Level-Ups verdient werden können.
- Linksklick auf den **REROLL-Button**, um den aktuellen Shop zu aktualisieren.

Nach jeder Welle öffnet sich der Shop und der Spieler bekommt die Möglichkeit, neue Waffen und Items zu kaufen. Items dienen vor allem dazu, die eigenen Attribute zu erhöhen. Doch nicht nur das! Manche von ihnen kommen auch mit einem kleinen Extra – zum Beispiel kann ermöglicht werden, Doppelschüsse abzufeuern! Neue Waffen dagegen bringen noch mehr Möglichkeiten für den Kampf gegen die Gegner. Falls Sie schon eine **Waffe auf Level 1 oder 2** besitzen, können Sie mit ein bisschen Glück die verbesserte Version erhalten und so noch mehr Schaden anrichten. Für den Fall, dass die Auswahl im Shop gar nicht nach Ihrem Geschmack ist, können Sie das Angebot im Shop aktualisieren. Doch Vorsicht, denn das hat seinen Preis…wortwörtlich.

Liste der Attribute

 Health – Die maximalen Lebenspunkte des Spielers

 Damage – Grundschaden, den der Spieler pro Treffer austeilt

 Attack-Speed – Angriffsgeschwindigkeit des Spielers

 Crit-Damage – Schadenserhöhung bei kritischem Treffer

 Crit-Chance – Chance, dass ein kritischer Treffer zustande kommt

 Psycho-Level – Limitierung für die Anzahl nutzbarer Passiv-Items

 Luck – Erhöht Wahrscheinlichkeit auf seltenere Items im Shop

 Speed – Geschwindigkeit, mit der sich der Charakter bewegt

 Attack Range – Angriffsreichweite der verschiedenen Waffen

Waffenkategorien

 Nahkampfwaffen – für Angriffe aus kurzer Distanz

 Fernkampfwaffen – für Angriffe aus größerer Distanz

 Spezialwaffen – Nah- oder Fernkampfwaffen mit speziellem Feature

Anforderungsanalyse

Wir haben folgenden in den Vorlesungen besprochenen Anforderungen verwendet:

- Funktionale Anforderungen
- Nicht-Funktionale Anforderungen
- Technische Anforderungen
- Fachliche Anforderungen
- Anforderungen an das MVP

Mehr Informationen dazu gibt es im Projekt-Wiki.

Git

Git-Commits

Bei jedem Commit in einen Branch soll definitiv eine Commit-Message angehangen werden. Unsere Commit-Struktur sieht so aus:

```
-git commit -m "[Commit-type] [Issue-Number]: [Message (What did you do in particualar? Sum it up in a few words)]"
```

Die *Issue-Number* durch auch nach der Message angehangen werden. Aber man sollte immer seine zugehörige Issue referieren.

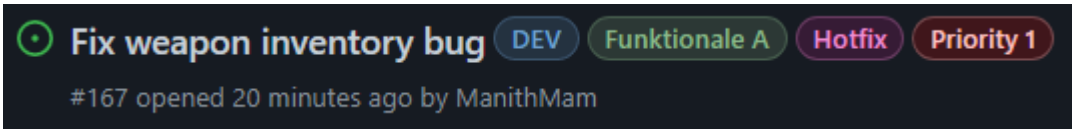
Ein gutes Beispiel sähe so aus (auch gerne kürzer):

Good example: -git commit -m "Feat #21: the player is now able to dodge bullets with SHIFT, gains 2 invincibility frames while dodging"

Hier sieht man einen Commit aus unserem GitHub:

Tags, ChangeLog, ReadMe

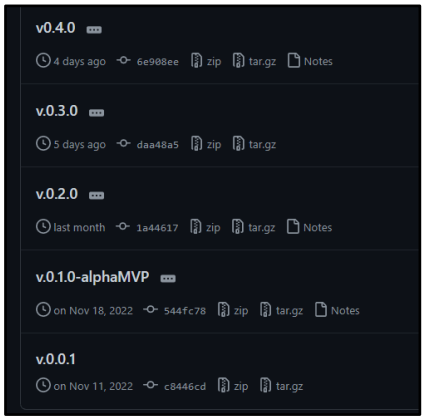
Für unser Projekt nutzen wir Issues. Wir füllten unser Backlog meist wöchentlich mit neuen Issues auf und haben sie mit unseren Labels markiert.



Unsere **ReadMe** findet man standardmäßig wie in vielen andere Git-Projekte direkt auf der Hauptseite. In der **ReadMe** gibt es eine kleine Beschreibung unseres Projektes. Außerdem findet man unter dem Punkt Features Eigenschaften unseres Spiel. Diese Eigenschaften beschreiben den Hauptinhalt unseres Spiels und die Besonderheiten. Danach existiert eine Installationsanleitung für den Docker und anschließend die Tastenbelegungen.

Die **ChangeLog** findet man direkt über der **ReadMe**. Die **ChangeLog** wird strukturiert in die jeweiligen Releases und Veränderungstypen. Hier wurde grob eine Erklärung der Änderungen eingefügt.

Tags wurden nur bei **Releases** verwendet. Wenn ein neuer Release ansteht, wird dieser mit dem gleichen Namen der Version getaggt.



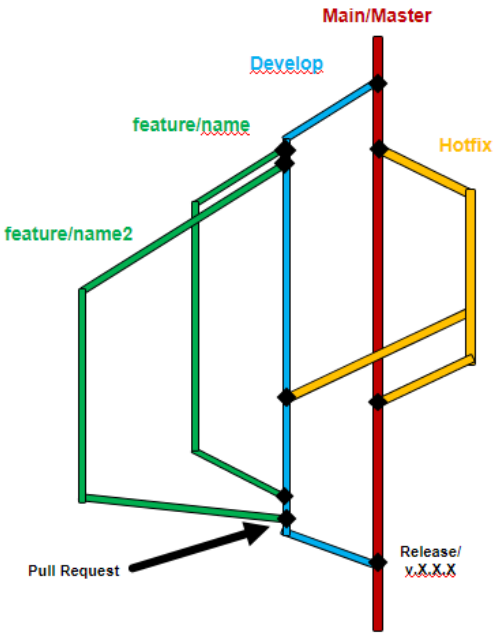
Vorgehen und Git-Branching

Unser Projekt besitzt einen **Main-Branch** und einen **Develop-Branch**. Der Main-Branch dient als Master bzw. Release-Branch. Wenn ein neuer Release ansteht, wird der Develop-Branch in den Main-Branch gemerged. Auf dem Develop-Branch wird entwickelt. Wenn man ein neues Feature entwickeln möchte, erstellt man zuerst einen neuen Branch aus Develop. Dieser wird als ***feature/featureName*** benannt. Auf diesem Feature-Branch wird nun das Gewünschte implementiert und danach wieder zurück in Develop gemerged. Doch bevor ein **Merge** durchgeführt wird, muss man eine Pull-Request starten. Hier müssen zwei andere Gruppenmitglieder die Pull-Request „*approven*“. Erst nach zwei „*Approves*“ hat man die Möglichkeit den Branch zu mergen.

Auf dem Develop-Branch und auf dem Main-Branch ist es nicht möglich Commits durchzuführen, durch **Branch-Protection-Rules**. Diese lauten:

- Linearer Verlauf
- Es benötigt eine Pull-Request vor einem Merge
- Es benötigt zwei Genehmigungen
- Status Check vor dem Merge

Unser Workflow-Graph:

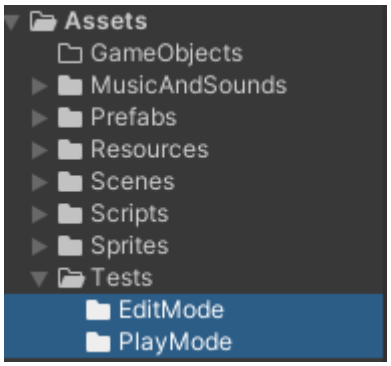


Tests

Die Tests werden mit dem Framework **NUnit** durchgeführt. In Unity existieren zwei Arten von Tests. Diese separiert man in **PlayMode** und **EditMode**.

Bei **PlayMode** werden Tests implementiert, die während dem Spiel laufen. Beispielsweise werden Gegner-Objekte zuerst im Spiel kreiert. Daher ist es nicht möglich vor dem Starten der Spielszene auf ein existierendes Gegner-Objekt zuzugreifen. Daher muss ein Test über Gegner-Objekte in **PlayMode** stattfinden. Generell kann man sagen, dass alle Klassen, die von **MonoBehaviour** erben in **PlayMode** getestet werden müssen.

In **EditMode** testet man Objekte bzw. Klassen, die schon vorher existieren. Diese können vor dem Starten des Spiels getestet werden.



Die Tests findet man jeweils in *Assets/Tests/EditMode oder PlayMode*.

In PlayMode existieren drei Tests. Zum Beispiel der Test *EnemySpawingAmount()* prüft, dass nicht zu viele Gegner auf einmal erscheinen können.

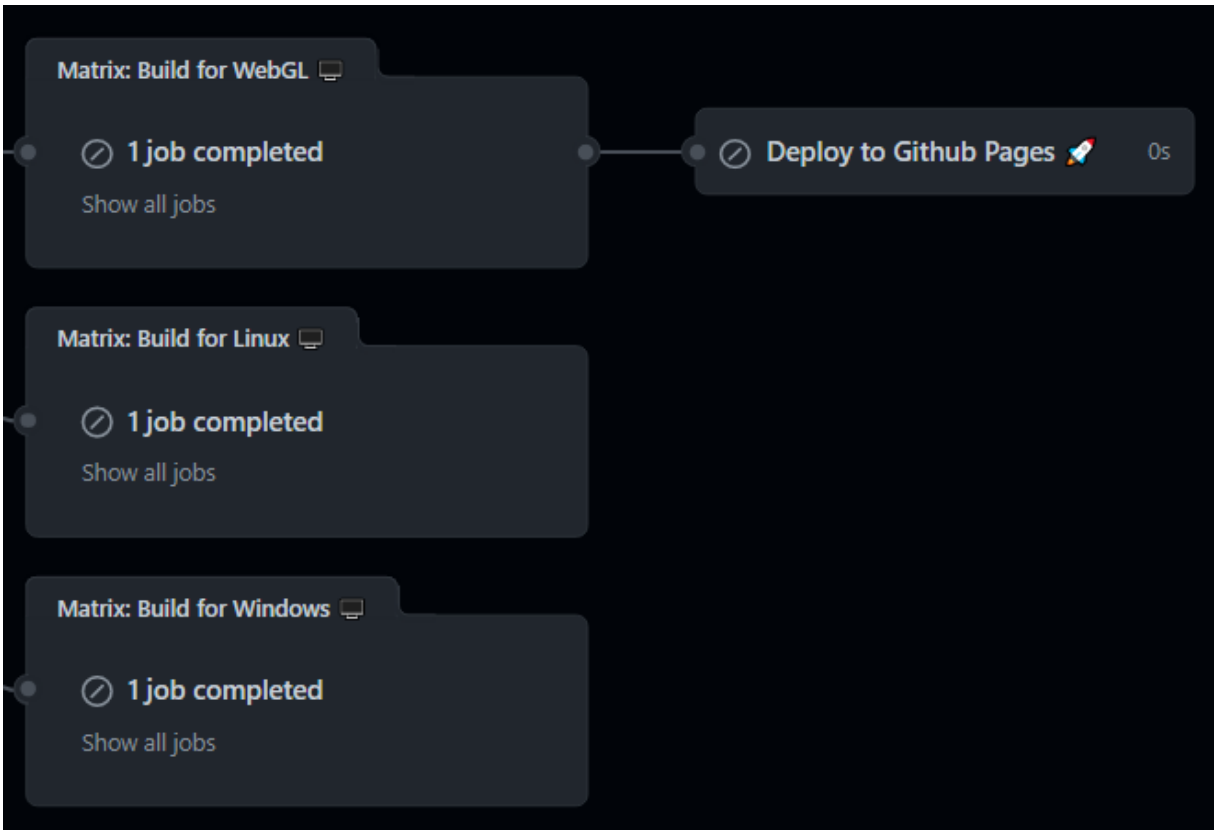
```
public class PlayModeTest
{
    [OneTimeSetUp]
    public void LoadScene()
    {
        SceneManager.LoadScene("SampleScene");
    }

    [UnityTest]
    public IEnumerator EnemySpawningAmount()
    {
        yield return new WaitForSeconds(5f);

        GameObject[] gameObject = GameObject.FindGameObjectsWithTag("Enemy");

        Assert.IsTrue(gameObject.Length <= 8);
    }
```

[OneTimeSetUp] wird einmal ausgeführt bevor vor den Testmethoden. Hier wir die im Spiel Szene *SampleScene* geladen. Im Test *EnemySpawingAmount()* werden zuerst 5 Sekunden gewartet, damit Gegner um einen herum erscheinen können. Danach wird überprüft, dass die Anzahl nicht über acht ist.

Das Testen im PlayMode in Unity ist leider sehr kompliziert, weil man keinen Zugriff auf die Methoden von MonoBehaviour hat. Das Verhalten der Spielobjekte muss über Umwege geschehen.

Building

Bei der Arbeit mit Unity gibt es keine klassischen Build-Manager wie Maven oder Gradle, wie wir sie aus Java kennen.
Stattdessen kann die Game-Engine selbst Builds mit individuellen Einstellungen für unterschiedliche Plattformen erstellen. Diese Builds beinhalten leider nur die spielbare .exe und verschiedene logDateien, aber nicht das ganze Projekt selbst.
In unserem Projekt werden beim Merge auf den Main-Branch drei Builds generiert: einer für Windows, einer für Linux und einer für WebGL. Letzterer wird dabei auch direkt auf github.io bereitgestellt, womit wir automatisch die ausführbare Anwendung erhalten.



CI/CD

Unter .github/workflows findet man die yml Dateien. Die activation.yml ist zum Aktivieren der Pipeline da und in der template.yml findet man die Definition der Jobs.

Bei einem Commit werden nur die ersten zwei Elemente der Pipeline verwendet. Zuerst wird die Unity Lizenz überprüft und danach laufen die Tests durch. Der Rest von CI/CD wird nicht verwendet.

Dieser Rest wird nur verwendet, wenn es zu einem Merge bzw. Release auf Main kommt. Dann werden die drei Builds kreiert und unter Artifacts gespeichert. WebGL wird auf sogar auf github.io bereitgestellt.

```yaml
buildWebGL:
  needs: testRunner
  if: ${{ github.ref == 'refs/heads/main' }}
```

UI

Unity besitzt ein UI Toolkit. Dieses beinhaltet auch UI Elemente, die man einer Scene hinzufügen kann und in einem "visual tree" angeordnet sind. Dabei sind schon Funktionen implementiert, wie die Größe des Elementes festzulegen oder auch die Position auf dem Canvas. Den UI Elementen kann man auch weitere Komponenten hinzufügen, sodass noch mehr Funktionen aus dem Toolkit benutzt werden können, z.B. welche die beim Layout helfen (vertikal layer group) oder um Bilder hinzuzufügen. Das bedeutet, dass man für statische Dinge, also für graphische Inhalte, die sich nicht ändern, kein Script braucht. Nur die OnClick-Methode der Buttons müssen programmiert werden (meistens ein Scenenwechsel). Dabei findet man alle Scripte zu UI und den bestimmten Szenen bei dem Ordner UI unter Scripte.
Für den Fall, dass Inhalte sich dauerhaft ändern können wie im Shop und Leaderboard, haben wir Templates verwendet. Diese sind Vorlage für die zu erstellenden UI Elemente und können im Script via Instantiate() Methode kopiert werden. Auf die Kindelemente des duplizierten Templates greifen wir mit der Find() Methode zu und können diesen so bestimmte Werte zuweisen. Beim Leaderboard waren es zum Beispiel die Highscore-Einträge aus der Datenbank.
Die HelperUI Klasse existiert, damit Methoden nicht zweimal geschrieben werden müssen. Nämlich werden in PauseMenu als auch im Shop die aktuellen Stats und ausgerüsteten Gegenstände angezeigt.

Schnittstellen

Backend:

async App.js:

- Initialisiert express Server
- Baut Verbindung zu Datenbank auf
- Definiert Router mit Endpoints und deren Middleware
- Definiert URL unter welcher der Router und deren Endpoints zu erreichen sind
- Definiert errorMiddleware

controllers:

account.js:

async getAccounts:

- sendet unter Status 200 JSON object mit Array aus allen Objects in Datenbank. Objects beinhalten felder playertag, password und email.
- Falls keine accounts gefunden sendet error message und status 400
- Erreichbar mit url: http://127.0.0.1:3000/account
- Methode: GET

async createAccount:

- Nimmt im request body json object entgegen
- Überprüft ob email oder playertag schon in Datenbank vorhanden
- Falls bereits vorhanden sendet status 400 und error message mit was von beiden bereits vorhanden

- Falls nicht wird request body in Datenbank eingefügt und status 201 mit erfolgs message gesendet
- Erreichbar mit url: http://127.0.0.1:3000/account
- Methode: POST

async Login:

- Nimmt parameter playerTag in url und request body mit password in json object entgegen
- Prüft ob playerTag vorhanden
- Wenn nicht sendet status 400 und error message
- Wenn doch wird geprüft ob password zu playertag passt
- Wenn nicht sendet status 400 und error message
- Ansonsten sendet status 200 und string true
- Erreichbar mit url: http://127.0.0.1:3000/account/:playerTag
- Methode: POST

async changeAccount:

- Nimmt in request parameter aktuellen playerTag entgegen
- Nimmt in request body json object mit zu ändernden feldern entgegen
- Prüft ob neuer playerTag oder email schon vorhanden
- Wenn ja sendet status 400 und error message
  Sonst ändert felder in Datenbank, wenn in body nicht leerer string oder nicht gleich wie felder in datenbank
- Wenn geändert sendet status 200 und erfolgs message
- Erreichbar mit url: http://127.0.0.1:3000/account/:playerTag
- Methode: PUT/ PATCH

async deleteAccount:

- Nimmt request parameter playerTag entgegen
- Wenn playertag nicht vorhanden sendet status 400 und error message
- Sonst sendet 200 und erfolgs message
- Erreichbar mit url: http://127.0.0.1:3000/account/:playerTag
- Methode: DELETE

async insertTestData:

- fügt in array DATA definierte Testdaten in account in Datenbank ein
- sendet status 201 und erfolgs message
- Erreichbar mit url: http://127.0.0.1:3000/insertTestData
- Methode: POST

playerData.js:

async getAllPlayerData:

- sendet unter status 200 json object mit nach highscore absteigend sortiertem array aus allen objects in datenbank. Objects beinhalten felder playertag, highscore.
- Falls keine accounts gefunden sendet error message und status 400
- Erreichbar mit url: http://127.0.0.1:3000/playerData
- Methode: GET

async getPlayerData:

- Nimmt request parameter playerTag entgegen
- Sendet bei erfolgreicher Datenbankabfrage unter status 200 json object. Object beinhaltet felder playertag, highscore.
- Falls keine accounts gefunden sendet error message und status 400
- Erreichbar mit url: http://127.0.0.1:3000/ playerData/:playerTag
- Methode: GET

async updateHighscore:

- Nimmt request parameter playerTag entgegen
- Nimmt in request body json object mit neuem highscore entgegen
- Updatet highscore und sendet status 200 und erfolgs message
- Erreichbar mit url: http://127.0.0.1:3000/ playerData/:playerTag
  Methode: PUT / PATCH

Middleware:

Error.middleware.js:

sync handleServerErrors:

- Sendet status 500 und error message bei fehler in server code sync

handleNotFoundErrors:

- Sendet bei nicht vorhandenen endpoints status 404 und error message

Log.middleware.js:

sync logRequests:

- Logt in serverconsole requests auf server

db:

db.js:

async connect:

- Baut verbindung zu MongoDb auf
- Setzt Datenbankverbindung auf locale variable connection

sync getConnection:

- Returned locale variable connection, welche Datenbankverbindung beinhaltet

Da unser Leaderboard Daten von Usern aus der ganzen Welt beinhalten soll mussten wir eine
Netzwerkschnittstelle nutzen. Dadurch das der Server mehrere Anfragen in kurzer Zeit erhalten kann
und Datenbank Operationen je nach Auslastung dauern können müssen alle Endpoints Funktionen
die Datenbank Operationen durchführen oder eine Verbindung zur Datenbank aufbauen asynchron
sein. Andere Funktionen besitzen solche Abhängigkeiten nicht und können/müssen daher synchron
ablaufen

Persistenz

CRUD-Operationen werden durch Methoden wie *GetHighscore, CreateAccount, UpdateHighscore* und *DeleteAccount* abgedeckt und finden auch anwendung. Demodaten stehen bereit aber Tests der Datenbankoperationen wurden keine erstellt.

Known Issues

- Die Sprites bei dem Waffen für das Schießen fehlen.
- Die Icons für die Items fehlen.
- Der Tooltip für die Attribute und ausgerüsteten Waffen fehlt, sodass man als Benutzer nicht genau weiß, was das Icon bedeutet.
- Ursprünglich wollten wir noch ermöglichen, Änderungen an dem Account vorzunehmen. Wegen Zeitmangels ist dies aber nicht mehr möglich.
- Leider konnten wir nur einen Charakter implementieren, obwohl wir mehrere geplant hatten.
- Es fehlen weitere Unit Tests, um zu prüfen, ob wirklich alles funktioniert.
- MasterVolume Slider funktioniert nicht

Reflexion

Unsere Reflexion setzt sich zum Großteil aus Dingen zusammen, die wir im Bereich Workflow gelernt haben. Hierzu zählt bereits das Auswählen der richtigen Programme für das Projekt. Anfangs griffen wir auf “Sourcetree” zurück um unseren Git-Workflow zu managen. Dies brachte eine Vielzahl an Problemen mit sich, die wir später mit GitHub Desktop nicht hatten. Auch wenn wir uns recht schnell auf Unity als Engine geeinigt haben, ohne viel Fortschritt gemacht zu haben, der den Wechsel erschweren könnte, hätten wir dies beim Workflow vermeiden können, indem wir direkt auf GitHub

Desktop gesetzt hätten. Als Folge dessen, hatten wir Anfangs Startschwierigkeiten in dieser etwas größeren Gruppe zu arbeiten. Dies hat sich jedoch im Laufe des Projekts schließlich routiniert eingearbeitet.

Unser Arbeitsmodell mit etwa einer Woche langen Sprints hat sich als weiterhin sinnvoll erwiesen, da wir so Probleme schnell ansprechen und ggf. In der Gruppe lösen konnten. Die Aufgabenverteilung war somit auch sehr strukturiert und klar. Die Teamgröße brachte jedoch das Problem mit sich, dass einige, auch wenn sie einen Sachverhalt im Meeting verstanden haben, aufgrund der Anzahl an Leuten, die sich bereits mit dem Problem beschäftigten, nicht sonderlich viel beitragen konnten. Hier wären einzelne Meetings in kleineren Gruppen womöglich sinnvoller und effizienter gewesen mit anschließendem Zusammentragen der Ergebnisse der kleinen Meetings in der gesamten Gruppe.

Nach den Meetings, die wir zu Beginn des Semesters zum Thema Vision für unser Projekt gehalten haben, waren wir der Ansicht, dass wir alle die gleiche, oder zumindest eine sehr ähnliche Vorstellung von dem Projekt hatten. Als wir jedoch später auf Details in der Umsetzung eingegangen sind, wurde klar, dass unsere Vorstellungen weiter variierten als wir erwarteten und somit noch der Bedarf dafür bestand, weiter auf die besprochenen Pläne einzugehen.

Grundsätzlich können wir noch festhalten, dass das konsequentere Setzen von Milestones bereits zu Beginn des Projekts sinnvoller gewesen wäre. So lässt sich das endgültige Ausmaß des Projekts besser abschätzen und zu implementierende Features besser planen und danach ordnen, wie realistisch und sinnvoll diese im Endeffekt sind.

Dennoch lässt sich festhalten, dass wir es geschafft haben, die Liste an übrigen Problemen oder fehlender Features überschaubar zu halten. Wir haben sehr viel zum Thema Arbeiten im Team gelernt und sind zufrieden mit unserem Projekt, trotz anfänglicher Unsicherheiten dazu ein Unity Projekt in C# zu wählen, ohne jegliche Erfahrung mit Unity zu haben.

Anmerkungen

- In Unity existiert kein Package-Private
- Reines OOP in Unity ist nur bedingt möglich (Mischung mit ECS)
- Es werden oft Komponenten zu Objekten hinzugefügt
- Klassenstrukturen darstellen der Packages ist daher nur bedingt möglich
- Beim mergen haben wir *Squash and Merge* verwendet, daher kann die Anzahl der Commit im Branch Main falsch sein
- Die Musik ist selbst kreiert und eingespielt 😊