Projektabschlussbericht MMS

Inhaltsverzeichnis

Einleitung

Hintergrund des Projekts

Am Anfang konnte sich ein Thema ausgesucht werden, welches mit der Übung oder der Vorlesung zu tun hat. Die wesentlichen Themen der Vorlesung wären zum Beispiel Text, Bild, Audio und Video. Bei den Beispielen in der Übung wurde ein Spiel gezeigt, wodurch die Entscheidung, ein Spiel zu programmieren, getroffen wurde.

Zweck und Ziel des Projekts

Mit dem Projekt soll das Team in den jeweils favorisierten Bereichen einen Leistungsaufwand von rund 20 Stunden umsetzen und sich dabei neue Fähigkeiten und Techniken aneignen. Im Fall dieses Videospiels sind dies das Design, die Kartengeneration, der Spieler und die Gegner. Um ein lauffähiges Spiel entwickeln zu können, musste das Team gut zusammenarbeiten, da jede dieser 4 Kategorien ineinandergreifen und zum Teil voneinander abhängig sind. Wenn dies nicht der Fall sein sollte, könnten ungewollte Bugs auftreten, das Thema des Spiels nicht zu der eigentlichen Spielerfahrung passen, die Spielerfahrung im Allgemeinen nicht gut sein oder das Spiel erst gar nicht funktionieren.

Team und Thema

Team

Unser Projektteam besteht aus vier engagierten Mitgliedern, die jeweils spezifische Bereiche des Projekts übernommen haben, um ein erfolgreiches Ergebnis zu erzielen. Je nach Teammitglied wird verschiedene Programmiererfahrung mitgebracht, jedoch wurde am Anfang des Projektes die Aufgabenteilung so gewählt, dass jedes Teammitglied die Bereiche des Projektes bekommt, die favorisiert wurden.

1. Jan Feldmayer – Versionierung (GIT), Player, Health-System & Collectables
2. Jonas Koplinger – Character- & Kartendesign
3. Mathias Kröpfl – Kartengeneration & Menu
4. Wasgen Tschobanjan – Gegner & Boss

Thema und Projektbeschreibung

“Pixel Purge” ist ein spannendes Top-Down 2D-Roguelike-Shooter-Spiel, das in Unity entwickelt wurde. Das Spiel wurde mit einem Singleplayermodus implementiert, in dem der Spieler gegen eine Vielzahl von Feinden kämpfen und versucht sich bis zum Boss durchzuschlagen. Wenn der Boss besiegt wurde, ist das Spiel beendet und man wird zurück ins Hauptmenu geladen. Durch die Tötung der Gegner in den verschiedenen Räumen, können Gegner kleine Health-Items fallen lassen, welche dem Spieler helfen dessen Leben wieder herzustellen.

Das Grundprinzip des Spiels ist, dass der Spieler mittels Maus und Tastatur einen Charakter durch ein Level steuert, indem ein Krieg zwischen Kreisen und Polygonen herrscht. Inmitten dieses Krieges ist der Hauptcharakter „Pixel-Bro“, der Anführer der Kreise. Dieser versucht sich durch die Territorien des Feindes zu Kämpfen und die Allianz der Polygone zu zerstören. Um dies zu schaffen, muss der Anführer

der Allianz, „Pixel-Dominator“, bezwungen werden. Dieser ist jedoch in seinem Geheimversteck, welches sich nur mithilfe aller Schlüssel aufsperren lässt.

Realisierung der Teilbereiche

Player

Der Spieler ist die Hauptfigur und steht im Zentrum des Spiels steht. Der Player besteht aus einem Körper (Kreis) und ein kleineres Rechteck, welches vor dem Körper platziert wird und eine Waffe simulieren soll. Dieser verfügt über einige Fähigkeiten, welche mittels folgender Funktionen und Skripts implementiert wurden.

Abbildung 1: Player

PlayerController: Der PlayerController gibt die Möglichkeit den Spieler mittels Tastatur und Maus zu steuern. Dabei wird das integrierte Input-System von Unity (legt fest mittels welcher Tasten welche Aktionen gefeuert werden) mit einem Skript, welches die physikalischen Eigenschaften (Beschleunigung, Geschwindigkeit, Rotationsgeschwindigkeit) setzt, verbunden. Dabei gibt es die Möglichkeit den Spieler in Richtung der Tastatur oder in Richtung der Maus zielen zu lassen.

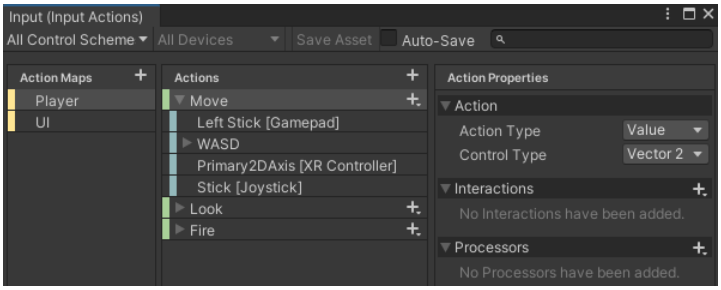

Abbildung 3: Unity-Input System

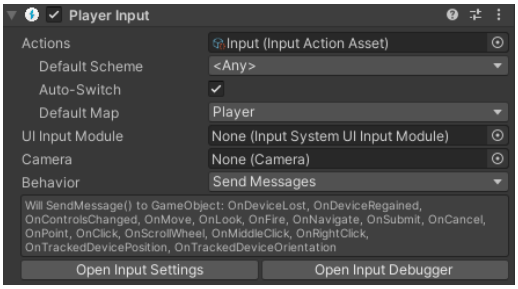

Abbildung 2: Input-System am Player setzen

PlayerShoot: Dieses Skript erlaubt den Spieler in die gezielte Richtung ein Bullet-Objekt mit einer angegeben Geschwindigkeit zu feuern. Zusätzlich kann man regulieren, wie schnell in Folge man Bullets abfeuern kann.

Abbildung 4: Player shooting

Bullet: Bullet ist ein Objekt, welches einem Projektil einer Waffe entsprechen soll. Dieses Objekt wird mittels dem PlayerShoot (eine Funktion oberhalb beschrieben) gefeuert. Im Bullet Skript selbst wird geregelt was passieren soll, wenn die Bullet auf ein anderes Objekt stoßt. Ist das kolierende Objekt der Spieler selber oder ein Gegner wird mittels der HealtContorller-Funktion TakeDamage() diesem Leben abgezogen. Ist das kolierende Objekt eine z.B. Wand wird die Bullet beim Aufprall zerstört.

HealthContolller: Der Spieler verfügt über eine Lebenspool und verliert, wenn er von einem Gegner getroffen wird, eine gewisse Menge. Hat der Spieler keine Leben mehr übrig, wird jegliche

Spieleraktionen gesperrt und es ist Game Over. Diese Funktion wird über einen universellen HealthController gesteuert und ist im Punkt Health System genauer erklärt.

Health System

HealthController:
Die Funktionen, welche es ermöglichen das der Spieler/ein Gegner Leben verliert und erhaltet, wird mittels einen universellen HealthController Skript gesteuert. Dieses Skript befindet sich auf jedem Objekt, welches über Leben verfügen soll (Spieler, Gegner) und besteht grundlegend aus zwei Funktionen TakeDamge() und AddHealth() sowie drei UnityEvents OnDied(), OnDamaged() und OnHealthChanged(). In den zwei Funktionen werden mit Logik die Events gefeuert um das gewünscht Verhalten zu erreichen.

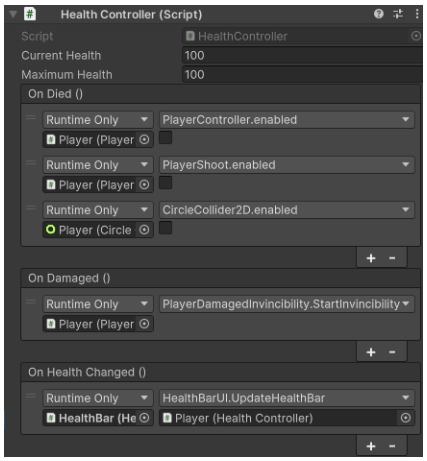

Abbildung 5: HealthController Player

HealthBar:
Das Leben des Spielers wird im UI im linken oberen Eck mittels länglichen horizontalen Rechtecks dargestellt. Diese sogenannte HealthBar besteht aus einem schwarzen Hintergrund und einem roten Vordergrund (stellt das Leben dar). Das Vordergrundbild verfügt über die Eigenschaft Fill Amount, welche von 0 – 1 die Breite des Vordergrundbilds bestimmt. Durch Zusammenspiel aus den Funktionen des Skripts HealthController, HealtBarUI und dem HealtController Event OnHealthChanged zeigt die HealthBar ständig die Menge des aktelles Lebenspools des Spielers an.


Abbildung 6: HealthBar voll


Abbildung 7: HealthBar halb voll

Invincibility Controller:
Der InvincibilityController ist wie der HealthController ein universelles Skript, welches für Spieler und Gegner benutzt wird. Wenn der Spieler/Gegner Schaden erhält wird das Event OnDamaged() des HealthControllers gefeuert und die Funktion StartInvincibility() des InvincibilityController ausgeführt. Dadurch wird der Spieler/Gegner für eine angegebene Zeit unverwundbar. Dieses Verhalten ist für den Spielverlauf notwendig, da sonst der Spieler bei Berühren einen Gegner pro Frame Schaden erleidet und somit „sofort“ Tod ist. Dieser Effekt des kurzen „unsterblich seines“ wird visuell durch einen sogenannten Flash Sprit (wechseln zwischen voller Objektfarben und transparenterer Version der Farbe) erzielt.

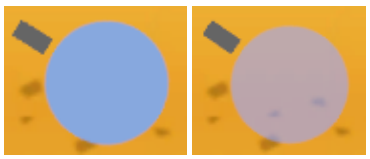
Abbildung 8: Player Flash Sprite

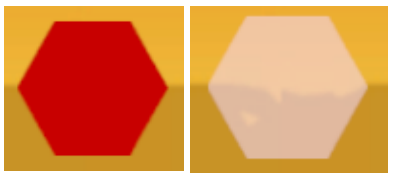
Abbildung 9: Gegner Flash Sprite

Collectable

Collectables ist ein Überbegriff für alle Objekte, welche von einem Spieler gesammelt werden können (z.B. HealthPack, Keys, etc.).
Für die Umsetzung wurde dafür ein Prefab Collectable mit dem gleichnamigen Skript erstellt und aus diesem die verschiedenen Prefab Variant wie z.B. Health_Collectable & Key_Collectable. Zusätzlich wurde ein Interface ICollectableBehaviour erstellt welches die Methode OnCollected (GameObject player) enthält, welche dann in den Klassen z.B. HealthCollectableBahavour konkret implementiert wird.
Der HealthCollectableSpawner enthält einen Liste an GameObjects(Objekte welche man spawnen will) und ist wie der Name schon sagt dafür zuständig, dass die Collectables an eine übergebene Position erscheinen. Alle Collectables werden nach dem „Aufheben“ zerstört, sodass man diese nicht unendlich oft wiederverwenden kann.

HealthCollectable(HealtPack):

HealthCollectables sind Collectabels, welche beim Aufheben mittels HealthController dem Spieler Leben zurückgeben. Das Skript EnemyCollectableDrop ist dann zuständig, dass besiegte Gegner mit einer angegebenen Wahrscheinlichkeit diese Collectables an die Stelle, wo sie besiegt wurden, fallen lassen.

Abbildung 6: Player with Health_Collectable

KeyCollectables:

KeyCollectables sind Collectables welche beim Aufheben die Anzahl der gesammelten Schlüssel (Keys) erhöht. Diese Schlüssel werden beim Spielstart in zufälligen Räumen auf der Spielfläche verteilt. Wird die benötigte Menge an Schlüssel gesammelt, wird der Boss-Raum geöffnet. Um die Anzahl der bereits gesammelten Schlüssel sowie die Anzahl der zu sammelnde Schlüssel verfolgen zu können, gibt es im UI im rechten oberen Eck einen Zähler (z.B. 1/3).

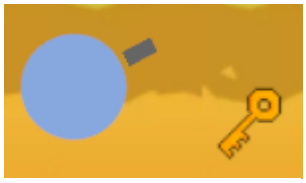

Abbildung 7: Player mit Key_Collectable

Character-Design

Beim Character Design ging es hauptsächlich um die Frage „Wie soll unser Spieler aussehen?“.
Man kennt es in nahezu jedem Spiel, dass es die Möglichkeit gibt, sich das Aussehen des Spielers selbst auszusuchen. Meist muss man Münzen oder ähnliches verdienen, um alle Charaktere freischalten zu können. Diese Option könnte man in weitere Folge in „Pixel Purge“ implementieren, derzeit ist es dem Spieler jedoch frei, zu entscheiden mit welchem Charakter er nun tatsächlich spielen möchte.

Aber wie implementiert man eine „Select Character“-Funktion nun tatsächlich?
Zuerst wurde eine neue GameScene, namens „Select Character“ erstellt und mit Buttons für das Navigieren durch die Scene erstellt. Danach wurde das Main-Menu um einen Button „Select Character“ erweitert. Dieser erhält im Inspector unter „OnClick()“ das MainMenu-Script, in dem programmiert ist, dass, wenn man auf den Button klickt, dass man zu dem SelectCharacterCanvas geleitet wird. Weiters muss man in OneClick() dem Button die Funktion zuweisen, dass auf diese Scene gegangen werden soll, sobald dieser Button gedrückt wird..

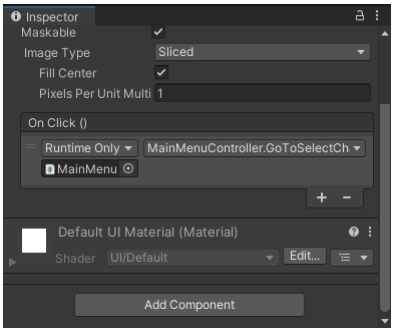

*Abbildung **9:** Hinzufügen Funktionen Button*

Danach erfolgte die Erstellung einer Character-Datenbank, die mit Sprites gefüllt werden kann. In dieser Datenbank kann man im Inspector funktionell eingeben, wie viele Sprites man hinzufügen möchte bzw. die gewünschten Sprites können dann einfach hineingezogen werden. Somit sind diese dann in dieser Datenbank, auf die dann zugegriffen werden kann. Bevor wir uns als Team auf ein einheitliches Design des Spiels festgelegt hatten, war diese Datenbank mit verschiedensten Emoji-Sprites gefüllt, um zu testen, ob die Datenbank auch wirklich funktioniert.

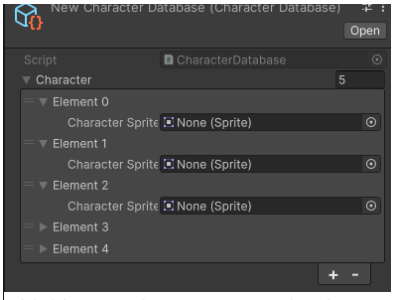

Abbildung 8 Character-Datenbank

In weiterer Folge wurde in einem CharagterManager-Script die Funktionalitäten des Next-Buttons, des Previous-Buttons bzw. das Speichern und Updaten des ausgewählten Designs des Characters ausprogrammiert, um dem Spieler zu ermöglichen, dass auch wirklich mit dem ausgewählten Design gespielt werden kann. Den Next- bzw. Previous-Buttons mussten, wie dem „Select Character“-Button im Main-Menu, die nötigen Funktionen zugewiesen werden.

Nachdem feststand, wie unser Spiel und unsere Charaktere nun tatsächlich aussehen sollen, wurde die Implementierung der „Select Character“-Funktion nochmals überarbeitet und in einem neuen Script „SelectCharacter“ implementiert. In diesem Script wurden zuerst die Farben des Players definiert und in einer Variable gespeichert. Durch Erhöhung/Verringern eines Indizes, das durch die Betätigung der Prev-/Next-Button geschieht, verändert sich in der GameScene ebenfalls die Farbe des Characters bzw. bei Erreichung der Grenzen (Erster/letzter Character) verschwinden oder erscheinen die Prev-/Next-Buttons.

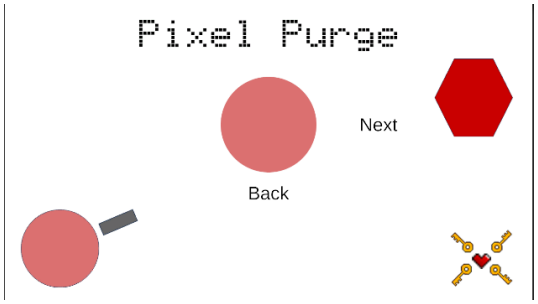

Abbildung 11 Beispiel "Select Character"-GameScene

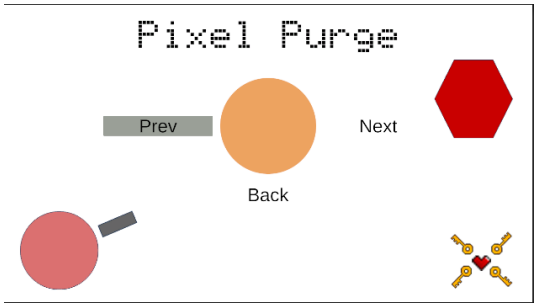

Abbildung 10 Beispiel "Select Character"-GameScene

Neben dem Player, der einen Kreis darstellt, entschieden wir uns als Team, den Gegnern eckige Formen zu geben. Somit wurden den verschiedenen Gegnern in SpriteRenderer eine Hexagon-Form (Melee-Enemy) oder Dreiecks-Form (Ranged/Sniper-Enemy) bzw. dem Boss (Elite-Enemy) ein Zehnzackige-Sternenform hinzugefügt.

Kartendesign

Hier ging es darum, dem Spiel in Form von passenden Hintergründen und Designanpassungen Leben einzuhauchen. Wir einigten uns als Team darauf, dass die Hintergründe des Spiels eher einfach gehalten werden sollten, um nicht vom eigentlichen Spielerlebnis abzulenken.

Da in unserem Spiel 3 „Basements“-Gamescenes (BasementStart: Spawnkarte des Players, BasementEmpty: Spawnpunkt der Gegner bzw. allgemeine Map und BasementEnd: Herrschaftsbereich des Endgegners „Pixel-Dominator“) vorhanden waren, die ein neues Design benötigten, mussten herfür 3 Hintergründe gestaltet werden.

Als erster Anhaltspunkt, wie die Hintergründe der 3 Basements aussehen könnten, diente die Internetseite „https://assetstore.unity.com“ (Siehe Quellen). Dort kann man sich etwas durchstöbern, bis man findet, nach was man sucht. In diesem Fall ging es um eventuell passende Map-Hintergründe. Eine Filterfunktion, Unterscheidung zwischen kostenlosen und kostenpflichtigen Assets und eine Suchfunktion erleichtern die Suche. Schlussendlich wurden passende Hintergründe gefunden, welche noch etwas angepasst und in das passende Format der Basement-Gamescenes (1920 x 1080 Pixel) formatiert werden mussten. Danach wurden diese zuerst unter Assets -> Models -> Map zum Projekt eingefügt und folgend

unter den jeweiligen „Basement“-GameScenes unter „Room“ -> „Background“ im SpriteRenderer als Sprite hinzugefügt.

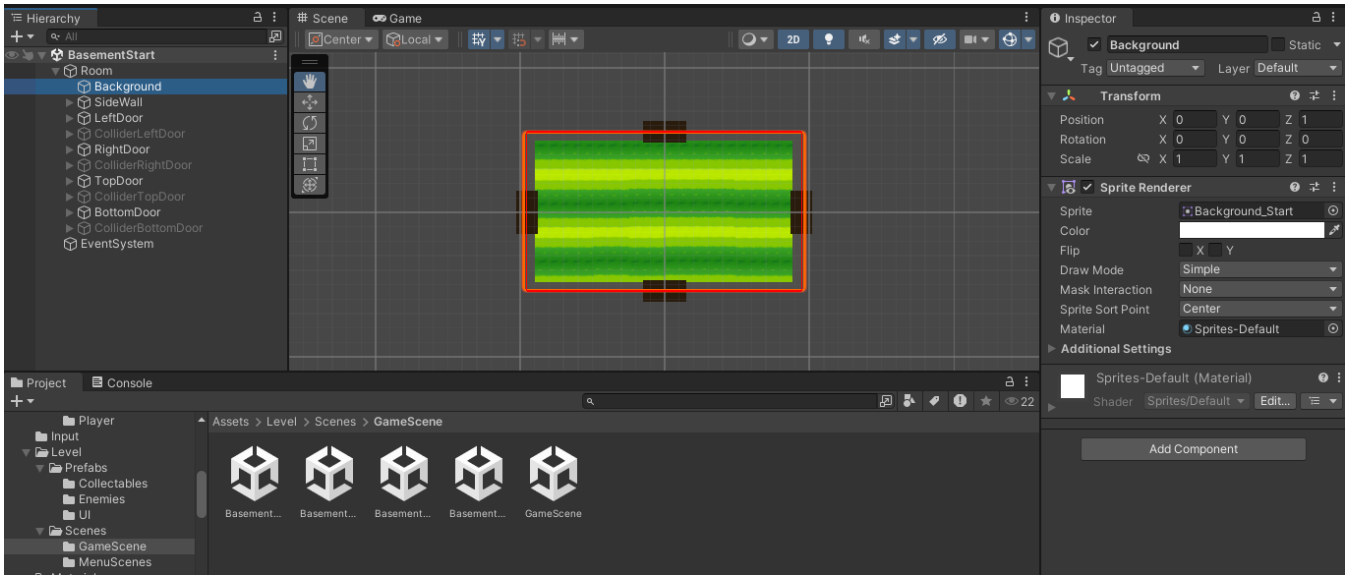

Abbildung 12 Einfügen Hintergund im SpriteRenderer

Für den BasementStart wurde sich für ein grüner Hintergrund entschieden, welcher eine grasähnliche Atmosphäre und Ruhe ausstrahlen soll, weil der Kampf der „Formen“ noch nicht begonnen hat, da das Spiel hier mit dem Spawnen des Players beginnt. Sobald man durch einen der anliegenden Türen steigt, wird man auf Gegner treffen. Die BasementEmpty beginnt ebenfalls mit Durchschreiten einer beliebigen Tür. Daher wurde ein bräunlicher Hintergrund gewählt, der Wüste und somit einen harten Kampf auf trockenem Boden imitieren soll. Der 3 Raum, der ein Design wünschte, war das „BasementEnd“ – Das Königreich des Allianzanführers der Polygone. Dieser Raum wurde mit einem eher dunkleren, mystischen Hintergrund mit rötlich/violettem Design versehen, um dem Spieler das Gefühl zu geben, in einem höhlen- bzw. höllenähnlichem Lager eingedrungen zu sein.

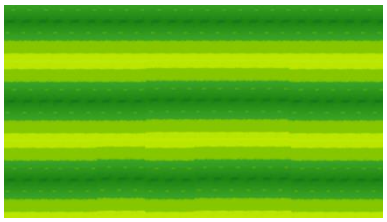

Abbildung 15 Background_Start

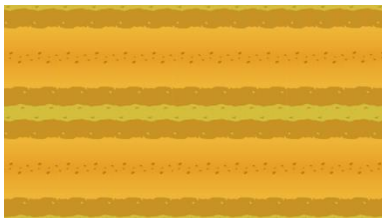

Abbildung 14 Background_Empty

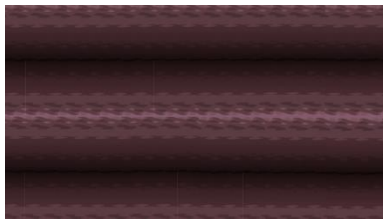

Abbildung 13 Background_End

Weiters benötigten auch die
Menu-Szenen eine Überarbeitung des Aussehens. Dazu wurde einfach ein weißer Hintergrund hergenommen und in Pixelschrift der Name des Spiels „Pixel-Purge“ und eine Abbildung des „Player-Characters“ der seine Waffe auf einen Gegner zielt, draufgesetzt. Im unteren rechten Eck wurden die Schlüssel, die der Spieler während des Spiels einsammeln muss, um den Bossraum freizuschalten und ein Herz, welches nach Ableben der Gegner spawnen kann, eingefügt.

Das Designen dieses Menu-Hintergrundes geschah ganz simpel in PowerPoint.

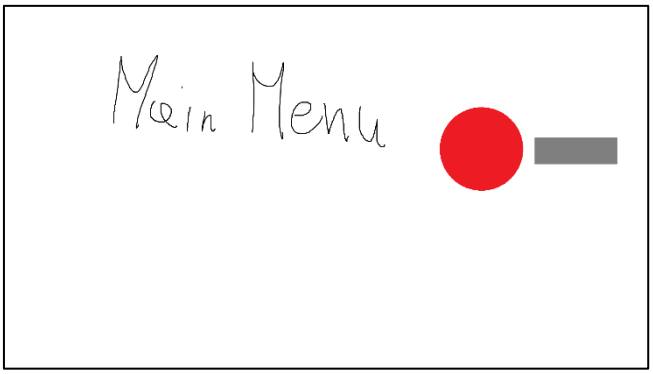

Abbildung 17 Menu-Design vorher

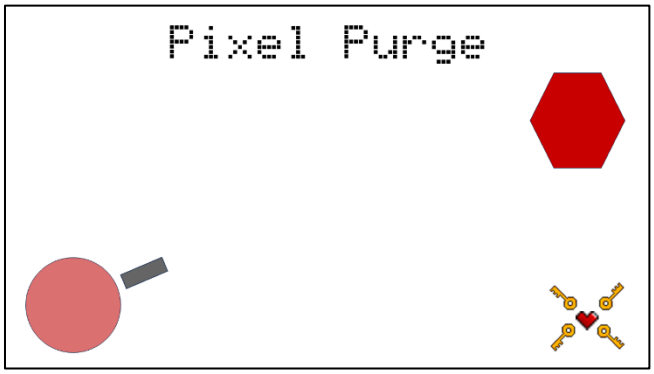

Abbildung 16 Menu-Design nachher

Kartengeneration

Für die Kartengeneration wird mit der Szene „BasementMain“ begonnen, die ein leeres Objekt namens „GameManager“ enthält. Dieser GameManager ruft bei seiner Initialisierung die Methoden zur Generierung der Karte auf. Im Spiel sind drei verschiedene Raumtypen erforderlich, die jeweils als separate Szenen umgesetzt wurden und beliebig oft in die „BasementMain“-Szene geladen werden können. Die drei Raumtypen sind:

1. Startraum: Dieser Raum dient als Ausgangspunkt für den Spieler und wird daher immer an den Koordinaten (0,0) platziert.
2. Leerer Raum (EmptyRoom): Dieser Raum ähnelt dem Startraum und bildet den Großteil der Spielfläche. Er ist flexibel und wird mehrfach im Dungeon eingesetzt.
3. Bossraum: Hier wird der Boss des Spiels platziert, und es markiert in der Regel das Ende eines Levels oder Abschnitts.

Zu Beginn der Entwicklung wurden provisorische Sprites als Hintergrund verwendet, um die Struktur der Räume darzustellen. Jeder dieser Räume ist mit vier Türen ausgestattet, die es dem Spieler ermöglichen, in angrenzende Räume zu wechseln.

In den beigefügten Abbildungen (von links nach rechts: Startraum, EmptyRoom, Bossraum) ist gut zu erkennen, dass jeder Raum über diese vier Türen verfügt, die den Übergang zu anderen Räumen ermöglichen. Dies schafft eine dynamische und vernetzte Spielwelt, die dem Spieler erlaubt, die Karte nach Belieben zu erkunden.

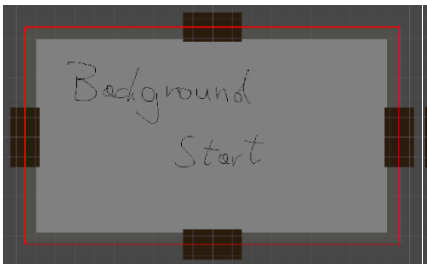

Abbildung 20 Startraum

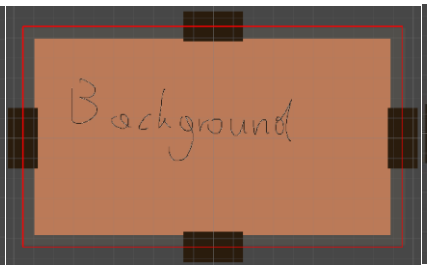

Abbildung 18 EmptyRoom

Abbildung 19 Bossraum

Um die Kartengeneration einfacher zu erklären, wird diese in 4 Schritte unterteilt: Crawler, RoomQueue, RemoveDoors, SpawnEntities.

Crawler

Durch die programmierten C#-Scripts ist es möglich eine zufällige Dungeon-Generierung durch die Zusammenarbeit von drei Hauptkomponenten zu erzeugen.

1. **DungeonGenerationData**: Dieses ScriptableObject speichert grundlegende Einstellungen für die Generierung. Es ermöglicht die Erstellung eines Konfigurationsdatensatzes im Unity-Editor, der die Anzahl der Erkundungseinheiten sowie die minimale und maximale Anzahl von Schritten, die sie ausführen sollen, festlegt.
2. **DungeonCrawlerController**: Dieses Skript steuert die Bewegung und Positionserfassung der Erkundungseinheiten im Dungeon. Es hält eine Liste aller besuchten Positionen und definiert, wie sich jede Richtung (oben, links, unten, rechts) auf die Position einer Einheit auswirkt.
   In der Hauptmethode dieses Skripts werden mehrere Einheiten initiiert und auf eine Startposition gesetzt. In einer Schleife, deren Länge zufällig innerhalb der festgelegten Grenzen variiert, bewegen sich die Einheiten durch den Dungeon. Jede neue Position, die erreicht wird, wird in die Liste der besuchten Orte aufgenommen, wodurch ein Netzwerk von Wegen entsteht.
3. **DungeonCrawler**: Diese Klasse modelliert eine einzelne Einheit, die den Dungeon erkundet. Sie speichert die aktuelle Position der Einheit. Beim Erstellen einer Einheit wird sie an eine Anfangsposition gesetzt. Die Methode zur Bewegung der Einheit wählt zufällig eine Richtung aus und aktualisiert die Position entsprechend.

In Kombination ermöglichen diese Skripte eine zufällig generierte Dungeon-Umgebung. Durch die Bewegung mehrerer Einheiten, die nach vordefinierten Regeln auf zufällige Weise neue Positionen erkunden, entsteht ein komplexes Labyrinth. Die generierten Wege und Räume basieren auf den Eingabedaten und erzeugen jedes Mal eine einzigartige Spielwelt.

RoomQueue

Die Skripte RoomController und Room arbeiten zusammen, um Räume zu laden, zu positionieren und die Spielwelt zu strukturieren. Der Fokus liegt hier auf dem Mechanismus des Raum-Spawns und der spezifischen Methode UpdateRoomQueue(), die die Reihenfolge des Ladens von Räumen steuert.

RoomController:

1. **Main**: Der RoomController ist das zentrale Steuerungsskript, das die gesamte Raumverwaltung und -erzeugung übernimmt. Er hält eine Liste aller geladenen Räume und verwaltet, wann und wie neue Räume hinzugefügt werden.
2. **Raum-Queue**: Eine Warteschlange (loadRoomQueue) speichert Informationen über die zu ladenden Räume. Diese Warteschlange wird in der Methode UpdateRoomQueue() verarbeitet.
3. **Raum-Registrierung**: Sobald ein Raum geladen ist, wird er mit seiner Position und Dimension in der Liste der geladenen Räume registriert.
4. **Bossraum-Management**: Wenn alle regulären Räume geladen sind, wird der Bossraum erzeugt und in die Liste der Räume eingefügt.

Room:

1. **Main**: Die Room-Klasse repräsentiert einen einzelnen Raum im Dungeon. Sie speichert Informationen über Position und Dimension und verwaltet die Türen des Raums.
2. **Tür-Management**: Da der Raum über alle nötigen Informationen verfügt, um die Türen zu entfernen und durch eine Wand zu ersetzen wurde eine öffentliche Methode geschrieben welche durch den RoomController aufgerufen wird.
3. **Spielerinteraktion**: Die Klasse verfolgt, ob sich der Spieler im Raum befindet, und steuert die Aktionen, die ausgelöst werden, wenn der Spieler den Raum betritt.

UpdateRoomQueue():

1. **Ladevorgang stoppen**: Wenn bereits ein Raum geladen wird, tut die Methode nichts weiter (die boolesche Variable isLoadingRoom verhindert die gleichzeitige Verarbeitung mehrerer Räume).
2. **Leere Queue überprüfen**: Wenn keine Räume mehr in der Warteschlange sind, wird geprüft, ob der Bossraum schon gespawnt wurde. Ist dies nicht der Fall, startet die Methode den Ladevorgang für den Bossraum. Ist der Bossraum bereits vorhanden, entfernt die Methode unverbundene Türen in allen geladenen Räumen.
3. **Raum aus der Queue laden**: Falls noch Räume in der Warteschlange sind, wird der nächste Raum aus der Queue genommen (currentLoadRoomData wird aktualisiert) und der Ladevorgang für diesen Raum beginnt (isLoadingRoom wird auf true gesetzt).
4. **Routine für Raumladen starten**: Die Methode startet die Coroutine: StartCoroutine(LoadRoomRoutine(currentLoadRoomData)), die den Raum asynchron lädt. Das bedeutet, dass sie das Laden eines Raums in die Spielszene handhabt, ohne den Hauptspielablauf zu unterbrechen oder zu blockieren. Dies ist besonders nützlich, um Ladezeiten im Hintergrund abzuwickeln, während das Spiel weiterläuft.

Nach dem Laden der Räume sieht die Karte wie folgt aus:

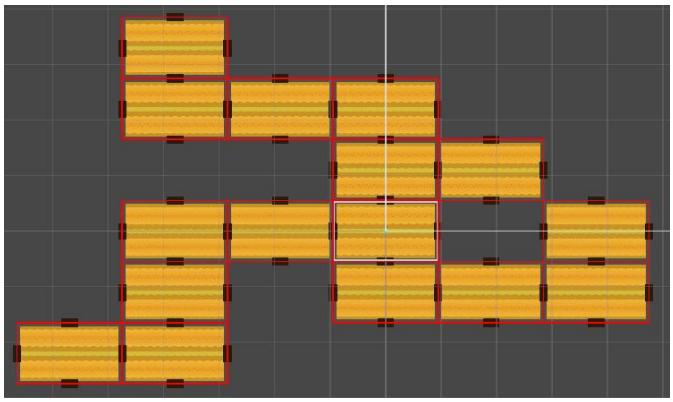

Abbildung 21 die Karte, nachdem alle Räume geladen wurden

Zum Schluss werden die Räume entfernt die doppelt vorkommen, zum Beispiel der Raum der auf dem aktuellen Startpunkt ist. Dies geschieht sehr simpel in einer Foreach-Schleife der loadedRooms, welche jeden Raum in ein Set speichert.

RemoveDoors

Die Methode RemoveUnconnectedDoors() in der Room-Klasse überprüft alle Türen eines Raums und entfernt diejenigen, die nicht zu einem benachbarten Raum führen. Sie durchsucht alle Türen des Raums und deaktiviert sie, falls der benachbarte Raum nicht existiert. Dies geschieht durch Aktivieren der entsprechenden Collidertüren, welche verhindern, dass der Spieler durch leere Türöffnungen geht. Damit wird sichergestellt, dass der Raum nur Verbindungen zu tatsächlich vorhandenen, benachbarten Räumen zulässt. Nachdem Entfernen der Türen, wird noch der Bossraum und die Gegner gespawned.

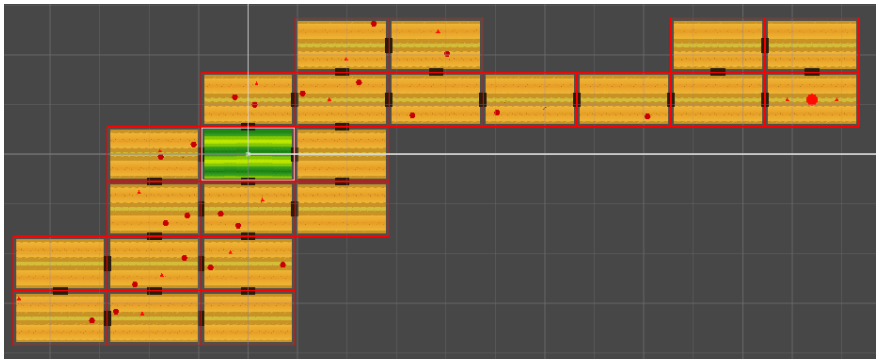

Abbildung 22 die Karte, nachdem alle Türen entfernt wurden

SpawnEntities

Für die Platzierung von Nahkampf- und Fernkampfgegner sowie von Schlüsseln in den Spielräumen wird der EntitySpawnController verwendet, welcher für das Spawnen von Gegner nur das vorgefertigte GameObject benötigt und eine minimale, sowie eine maximale Anzahl der zu generierende Gegner. Dieselben Voraussetzungen werden auch für die Generierung der Schlüssel vorausgesetzt, jedoch ist die Methode wie diese platziert werden eine andere.

1. **SpawnEntitiesInRoom**: Diese Methode durchläuft die geladenen Räume im Spiel und fügt eine variable Anzahl von Gegner hinzu. Beim Durchlaufen der Räume wird die Distanz des aktuellen Ruames zum Start raum berücksichtigt und die Gegneranzahl interpoliert. Räume wie der Start- und der Bossraum werden übersprungen, da sie besondere Anforderungen haben. Die Einheiten werden innerhalb der Raumgrenzen zufällig platziert, um ihre Erscheinungsorte unvorhersehbar zu machen.
2. **SpawnKeysForBossRoom**: Diese Methode verteilt eine zufällige Anzahl von Schlüsseln in den Spielräumen, die benötigt werden, um den Bossraum zu betreten. Die genaue Anzahl der Schlüssel wird innerhalb eines festgelegten Bereichs zufällig festgelegt. Die Schlüssel werden nicht im Start- oder Bossraum platziert, sondern in den regulären Räumen des Spiels. Die Positionen der Schlüssel innerhalb der Räume werden ebenfalls zufällig gewählt.

Wenn die Räume fertig geladen sind, und die Gegner sowie die Schlüssel platziert wurden, ist die Kartengeneration abgeschlossen und die Karte sieht wie folgt aus:

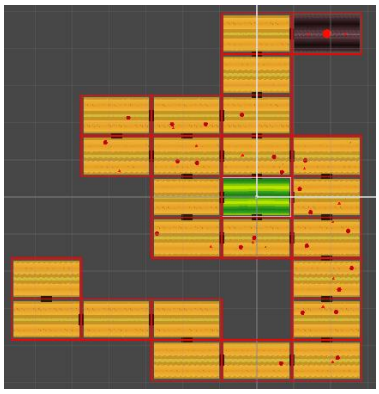

Abbildung 23 Karte nachdem der Bossraum hinzugefügt und alle Gegner gespawnt wurden

Hauptmenü

Das Hauptmenü ist die Anfangsscene, welche beim Starten des Spiels ausgeführt wird. In dieser Scene ist es möglich ein paar Spieleinstellungen zu ändern. Es kann die Farbe des Spielers verändert werden, außerdem ist es möglich Musik ein- bzw. auszuschalten.

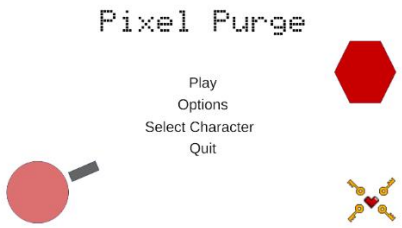

Abbildung 26 Hauptmenü

Abbildung 25 Optionsmenü

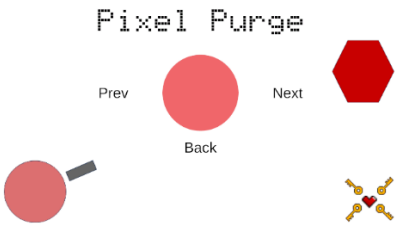

Abbildung 24 CharacterSelectmenü

Für das weitere erklären des Hauptmenüs hilft die Hierarchie der Scene. In dieser ist ein Canvas welcher das Hintergrundbild, das MainCanvas, den OptionsCanvas und den CharacterSelectCanvas enthält. Wie im Unterpunkt CharacterDesign bereits beschrieben, besteht jedes dieser Menüs aus Knöpfen und dem Hintergrund. Die Knöpfe können unter anderem verwendet werden, um GameObjekte zu de- oder aktivieren. Somit kann mit dem OptionsButton der MainCanvas deaktiviert und der OptionsCanvas aktiviert werden.

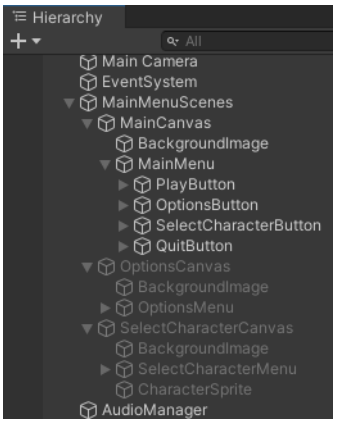

Abbildung 27 Hierarchie der Menüscene

Gegner

Es gibt eine Vielzahl von Gegnertypen, darunter Nahkampf-, Fernkampf- und Boss-Gegner, die jeweils einzigartige Fähigkeiten, zusätzliche Mechaniken und Verhaltensweisen aufweisen.

1. Nahkampf-Gegner: Die Nahkampf-Gegner nutzen
die EnemyMovement und EnemyAttack Klassen. Die EnemyMovement Klasse steuert die Bewegung des Gegners. Sie aktualisiert ständig die Richtung zum Spieler und passt die Geschwindigkeit des Gegners entsprechend an. Wenn der Spieler in Reichweite ist (_attackRange), bewegt sich der Gegner in Richtung des Spielers und greift an.
Die EnemyAttack Klasse ermöglicht es dem Gegner, Schaden zu verursachen, wenn er mit dem Spieler kollidiert. Dies stellt eine direkte und unmittelbare Bedrohung für den Spieler dar und erfordert schnelle Reaktionen und Bewegungen, um Schaden zu vermeiden.

2. Fernkampf-Gegner: Die Fernkampf-Gegner nutzen auch die EnemyMovement aber auch die RangedEnemyAttack Klasse. Die RangedEnemyAttack Klasse ermöglicht es dem Gegner, Projektile auf den Spieler zu schießen. Die Geschwindigkeit, der Schaden und die Zeit zwischen den Schüssen können individuell eingestellt werden. Die Projektile werden in Richtung des Spielers abgefeuert, wenn der Spieler in Sichtweite ist und der Gegner sich nicht bewegt. Dies stellt eine indirekte Bedrohung für den Spieler dar und erfordert taktisches Bewegen und Ausweichen.

3. Boss-Gegner: Die Boss-Gegner nutzen die EnemyMovement und EliteEnemyAttack Klassen. Die EliteEnemyAttack Klasse ähnelt der RangedEnemyAttack Klasse, ermöglicht es dem Boss jedoch, sich gleichzeitig zu bewegen und zu schießen. Dies erhöht die Schwierigkeit und macht den Boss zu einer größeren Herausforderung für den Spieler. Der Boss stellt die ultimative Prüfung der Fähigkeiten des Spielers dar und erfordert eine Kombination aus schnellen Reaktionen, taktischem Bewegen und effektivem Angriff.

4. Spielerbewusstsein: Die PlayerAwarenessController Klasse ermöglicht es den Gegnern, sich des Spielers bewusst zu sein, wenn er sich in einer bestimmten Entfernung befindet. Dies wird verwendet, um das Verhalten des Gegners zu steuern, z.B. ob er den Spieler angreift oder ihm folgt. Dies trägt zur Dynamik des Spiels bei und macht das Verhalten der Gegner weniger vorhersehbar und realistischer.

5. Unverwundbarkeit nach Schaden: Die EnemyDamagedInvincibility Klasse ermöglicht es den Gegnern, nach dem Erhalt von Schaden für eine bestimmte Dauer unverwundbar zu werden. Während dieser Zeit ändert sich die Farbe des Gegners, um den Spieler visuell darauf hinzuweisen, dass der Gegner vorübergehend unverwundbar ist. Dies fügt eine zusätzliche Ebene der Komplexität hinzu und erfordert, dass der Spieler seine Angriffe sorgfältig zeitlich abstimmt.

6. Zerstörung von Gegnern: Die EnemyDestroyController Klasse ermöglicht es, einen Gegner vollständig aus dem Spiel zu entfernen. Dies wird typischerweise verwendet, wenn der Gegner genug Schaden erlitten hat und besiegt wurde.

7. Sammelobjekte fallen lassen: Die EnemyCollectableDrop Klasse ermöglicht es den Gegnern, zufällig Sammelobjekte fallen zu lassen, wenn sie besiegt werden. Die Wahrscheinlichkeit, dass ein Sammelobjekt fällt, kann individuell eingestellt werden. Dies fördert das Besiegen von Gegnern und belohnt den Spieler für seine Bemühungen.

8. Boss-Gegner Tod: Die EliteEnemyOnDeath Klasse definiert das Verhalten des Spiels, wenn ein Boss-Gegner besiegt wird. In diesem Fall wird das Spiel zum Hauptmenü zurückgeführt.

Projektentwicklung

Am Anfang des Projektes wurde eine einfache Unity-Spielszene (Scene) erstellt, die als Basis für die weitere Entwicklung galt. Jedes Teammitglied hat sich eine eigene SampleScene erstellt und unabhängig von den anderen weiterentwickelt, um zu vermeiden das Fortschritt durch Überschreibung verloren geht. Für die Versionierung und Verwaltung des Entwicklungsfortschrittes wurde GIT (GitHub) verwendet. Durch GIT und Anpassung der .gitignore Datei (Datei, wo konfiguriert wird was bei GIT-Commits nicht in die Versionierung mit aufgenommen werden soll) konnte eine reibungslose Entwicklung von statten gehen, mit kaum vorhandenen Konflikten zwischen bearbeiteten Dateien. Nachdem die einzelnen Spielfunktionen in den separaten Scenes funktionierten, wurden diese einzeln zusammengefügt, um das Hauptspiel zu bilden. Zum Beispiel wurde in der SampleScene_Jan (Abb. 31) die Funktionen des Spielers, sowie der Gegner und in der BasementMain (Abb. 30) die Funktion der Kartengenerierung getestet. Durch die Nutzung von Unity Prefabs konnte man dann problemlos die beiden Scenes zusammenführen. Das zusammengeführte Spiel wurde dann getestet, um noch kleine Fixes, Verbesserungen sowie Balancing vorzunehmen.

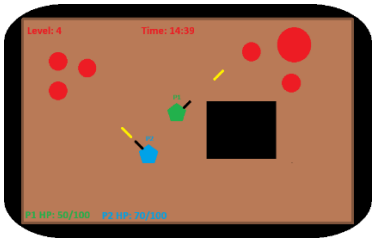

Abbildung 29 Ursprüngliche Skizze des Spiels

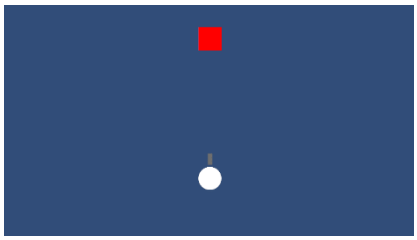

Abbildung 28 SampleScene

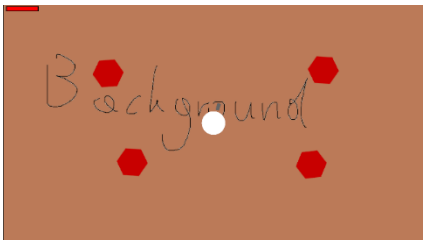

Abbildung 30 SampleScen_Jan

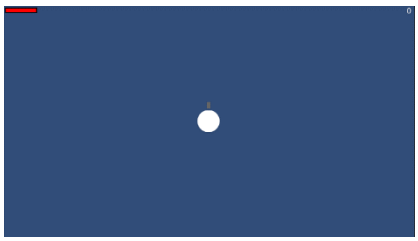

Abbildung 31 BasementMain

Quellen

Da mindestens drei Quellen für unsere Kollegen angegeben werden sollen, welche für zukünftige Projekte relevant werden könnten, haben wir uns für diese 3 entschieden:

1. https://unity.com/how-to/organizing-your-project
2. https://docs.github.com/en/get-started/start-your-journey/hello-world
3. https://learn.microsoft.com/en-us/dotnet/csharp/

Der erste Link führt zu einer Seite, welche eine gute Organisation in Unity erlaubt, wir finden es sehr wichtig bei größeren Projekten einen guten Überblick zu haben. Falls dies nicht der Fall ist und die Dateien irgendwie benannt und irgendwo gespeichert werden, wird das Projekt sehr schnell sehr unübersichtlich.

Im zweiten Link ist ein kurzer Guide zu GitHub, welcher die Grundprinzipien erklärt und die verschiedenen Möglichkeiten zeigt. Für uns war GitHub ein sehr nützliches Tool, da man einen Ordner teilt und jedes Teammitglied durch einfaches Drücken eines Knopfes die neuen Features hinzufügen oder auch die von anderen Teammitgliedern implementierten Scripts sofort abrufen kann. Ein weiteres nützliches Feature von GitHub ist die History welche erlaubt zu sehen wer wann welche Änderung vorgenommen hat.

Der dritte Link soll nur daran erinnern, bevor man zum Programmieren anfängt, sich zuerst mit der Sprache vertraut zu machen. Da C# für die meisten im Team neu war, wären ohne kurze Einblicke in die Sprache sowie Worst-/Best-Practices der Entwicklungsverlauf einiges holpriger verlaufen.