„LEAFY HOLLOWS“

Schriftliche Ausarbeitung
zur Klausurersatzleistung

Oskar Meyenburg
Robert-Havemann-Gymnasium
29.11.2023

Oskar Meyenburg
Robert-Havemann-Gymnasium
29.11.2023

Inhaltsverzeichnis

Oskar Meyenburg
Robert-Havemann-Gymnasium
29.11.2023

1. Einleitung und Ideenfindung

Eine Idee für unser Projekt haben wir schnell gefunden. Wir wollten ein Spiel programmieren, da wir beide bereits zuvor Spiele programmiert hatten. Inspiriert wurde wir durch Spiele, wie Rain World und Noita. Rain World simuliert ein sehr organischem Ökosystem, bestehend aus unterschiedlichsten Kreaturen und gesteuert von künstlicher Intelligenz. Noita hingegen simuliert das Verhalten von Stoffen, wie Sand und Wasser in einem zufällig generiertem Höhlensystem. Unser Plan war es ein 2D-Spiel zu programmieren, das in einem ebenfalls zufällig generiertem Höhlensystem in der Seitenansicht spielt und später vereinfachte Kreaturen und Stoffe hinzuzufügen. Letzteres haben wir jedoch später vernachlässigt. Als Programmiersprache haben wir hauptsächlich Python genutzt, da wir beide damit einiges bereits gemacht haben. Zwischenzeitlich haben wir auch C verwendet, hatten damit aber Probleme mit der Installation und Ausführung auf verschieden Geräten und haben nun auf C verzichtet. Neben Python nutzen wir die OpenGL Shading Language (GLSL) für Shader. Obwohl wir beide es vorher noch nicht genutzt hatten, haben wir von Anfang an GitHub verwendet.

2. Projektaufbau

Der Hauptordner des Projekts heißt "source". Darin befindet sich die "main.py"-Datei, mit der das Programm ausgeführt werden kann. Um diese auszuführen, muss Python installiert sein. Getestet haben wir es mit Python 3.10 und Python 3.11. Das Gerät muss OpenGL 3.3 core oder eine neuere Version unterstützen. Zusätzlich müssen die Python-Libraries mit den vorgegebenen Versionen in requirements.txt installiert sein. Dazu kann entweder install_requirements.py ausgeführt werden oder "pip install -r requirements.txt" in der Konsole ausgeführt werden. Bei der letzteren Variante sollte auf die genaue Version von bereits installierten Packages, wie "numpy", geachtet werden. Falls die "Microsoft Visual C++ Build Tools" nicht bereits installiert sind, kann "noise" nicht installiert werden. Sofern "opensimplex" aber problemlos installiert wurde, nutzt unser Programm dynamisch dieses, statt der standardmäßig verwendeten noise-Library.
In dem Hauptordner befinden sich auch noch die Unterordner "scripts" und "data". "scripts" enthält alle von der "main.py"-Datei verwendeten Python-Dateien. Diese werden unterteilt in die Unterordner "game", "graphics", "menu" und "utility". Im ersten Ordner sind alle Skripte, die für das Gameplay, die Physik, die Welt, den Spieler und die Weltgeneration zuständig sind. Der Ordner "graphics" enthält die Skripte, die für die Grafik, aber auch für das Laden von Bildern, sowie für das Laden und Abspielen von Audio-Dateien, genutzt werden. Zusätzlich befinden sich hier Skripte für Partikel und Schatten. Im Ordner "menu" sind Skripte, die den Aufbau des Menüsystems mit Knöpfen u.ä. bestimmen. In dem "utility"-Ordner befinden sich Skripte, die von verschiedenen anderen Teilen des Programms genutzt werden. Darunter sind Konstanten und Funktionen zum Lesen und Schreiben von Textdateien, zum Laden und Speichern von Einstellungen und zum Starten von Threads.
Im Ordner "data" kann man alle anderen Dateien des Projektes finden. Solche sind Schriftart-, Audio- und Bilddateien, sowie JSON-Dateien, die beispielsweise definieren, welches Bild zu welchem Block gehört, welche Eigenschaften bestimmte Blöcke haben und wie Bilder animiert werden sollen. Hier befinden sich auch die Skripte der Shader. Weiterhin sind Dateien von vorgebauten Strukturen zu finden, vorgefertigte Übersetzungen von Texten und Wörtern und Dateien vom Nutzer. Die Dateien des Nutzers werden von dem Programm beim Start gelesen und später überschrieben und enthalten zum einen Daten zu den Einstellungen und zum anderen alle Daten der generierten Welt, sowie das Inventar des Spielers.

3. Entwicklung

3.1 Erster Versuch des Grafiksystems

Ende Mai habe ich angefangen, ein simples Grafiksystem mit Pygame zu gestalten. Um Dateien, wie Bilder und Audio, zu importieren habe ich Funktionen geschrieben, die Dateipfade formatieren, sodass diese immer von dem "main"-Ordner starten. Das war wichtig, um später die Nutzung von PyInstaller zu ermöglichen.
Wir wollten zwar Pygame nutzen, gleichzeitig aber auch die Möglichkeit haben Shader zu verwenden. Um das gewünschte Ergebnis zu erreichen, musste ich einige Ansätze ausprobieren. Für Shader wollte ich OpenGL verwenden, auch wenn ich OpenGL zuvor noch nicht genutzt hatte. Das Ziel war, eine Möglichkeit zu haben, die die Geschwindigkeit nicht zu sehr beeinträchtigt und zwei "Leinwände" zum Abbilden von Bildern zu haben. Die eine Leinwand sollte von einem Fragment-Shader beeinflusst werden und die Welt repräsentieren. Die andere Leinwand sollte darüber liegen, teilweise transparent sein und eine nicht vom Shader beeinflussten Benutzeroberflächen abbilden.
Der erste Ansatz war, zwei Pygame-Surfaces zu haben und mittels Pygame auf diese zu malen. Dann sollten beide Surfaces in OpenGL-Texturen umgewandelt werden und ein GLSL-Fragment-Shader fügt diese dann zusammen. Dieser arbeitet auf der GPU für jeden Pixel einzeln und kann so das gewünschte Ergebnis erzeugen. Aufgrund der Umwandlung war der Ansatz jedoch zu langsam.
Zur Umsetzung setzte ich mich an die Implementation von Shadern und erstellte eine Shader Klasse. Diese habe ich verwendet, um auf einfachem Wege Buffer, Texturen und Variablen zu den Shadern zu schicken. Als erster Test der Shader bestand das Spiel zu diesem Zeitpunkt aus einem Bild eines Baumes, das der Maus folgt und dessen Pixel vertikal entlang einer Sinusfunktion durch einen Fragment-Shader verschoben wurden. Zu dem Zeitpunkt wurden Leinwände von Pygame umständlich zu OpenGL Texturen konvertiert.
Anfang Juni habe eine Möglichkeit eingebaut, dass das Programm, falls OpenGL nicht funktioniert, nur Pygame nutzt. Ich habe diese Funktionalität auch eingebaut, weil die Konvertierung von Surfaces zu OpenGL Texturen langsam ist. Dennoch habe ich diese Funktion später entfernt.
Die neu entwickelte Klasse "Window" war zuständig, um Eingaben mit Maus und Tastatur zu verwalten. Des Weiteren baute ich ein System, das Schriftarten laden kann, die ich zuvor als PNG-Datei gespeichert hatte. Zusätzlich habe ich eine Kameraklasse erstellt. Bei dieser Kamera kann ein Punkt festgelegt werden und die Kamera bewegt sich langsam dort hin, bzw. die Welt wird in entgegengesetzter Richtung verschoben.

3.2 Menü

Am 5. Juni begann ich mit dem Menü.
Das Menü wird generiert von den Dateien "menu.py" und "widgets.py". In der Datei "widgets.py" sind Klassen für alle Widgets definiert. In "menu.py" wird der Aufbau des mehrseitigen Menüs bestimmt. Dafür kann eine neue Seite erstellt werden und Widgets können dann auf diese Seite gesetzt werden. Die Widgets werden angeordnet, ähnlich wie bei Tkinter unter der Verwendung von Widget.grid(). Die Nutzung von `row=...` und `column=...` ist in dem System jedoch optional. Wenn diese Parameter weggelassen werden, werden die Widgets in der Reihenfolge der Erstellung dynamisch angeordnet, unter Berücksichtigung von der Anzahl der Spalten einer Seite und der Anzahl an Spalten, die ein Widget einnimmt. Im Gegensatz zu Tkinter wird nicht jedes einzelne Widget mit Widget.grid() gesetzt, sondern zum Schluss page.layout() ausgeführt. Werden danach noch Widgets hinzugefügt, können diese mit .layout_prepend() oder .layout_append() bestätigt werden. Mit page.open() kann eine Seite geöffnet werden. Die vorherige wird dabei

automatisch geschlossen. Die verfügbaren Widgets sind Label, Button, Slider, Entry, ScrollBox, LoadingBar und HoverBox.

Beispiel (vereinfachter Aufbau der Hauptseite):

```
# Erstellung einer Seite
main_page = Page(columns=2, spacing=MENU_SPACING)

# Erstellung eines großen Labels mit dem Text "Title", das zwei Spalten einnimmt. (zum diesem Zeitpunkt hatten wir noch keinen Namen für unser Spiel)
Label(main_page, MENU_TITLE_SIZE, columnspan=2, text="Title",
fontsize=TEXT_SIZE_HEADING)

# Erstellung eines Knopfes zum Spielen, der zwei Spalten einnimmt. Wenn er gedrückt wird, wird die Funktion 'lambda: self.set_state("load_world")' ausgeführt
button_main_play = Button(main_page, MENU_BUTTON_SIZE, columnspan=2,
text="Play", fontsize=TEXT_SIZE_BUTTON, callback=lambda:
self.set_state("load_world"))

# Erstellung eines Knopfes, der die Einstellungsseite öffnet; "callback" wird hier erst später definiert, wenn die Einstellungsseite auch erstellt wurde
button_main_settings = Button(main_page, MENU_BUTTON_SMALL_SIZE,
text="Settings", fontsize=TEXT_SIZE_BUTTON)

# Erstellung eines Knopfes, der das Spiel schließt, geregelt in der Funktion "window.quit()".
Button(main_page, MENU_BUTTON_SMALL_SIZE, callback=window.quit, text="Quit",
fontsize=TEXT_SIZE_BUTTON)

# Widgets anordnen
main_page.layout()

# Hauptseite öffnen
main_page.open()
```

3.3 Zweiter und Dritter Versuch des Grafiksystems

Nach der Arbeit an dem Menü, habe ich versucht Funktionen, die in C geschrieben sind, mit Ctypes in Python zu importieren und auszuführen. Dabei hoffte ich, besonders intensive Funktionen mit C ausführen zu können. Als dies funktionierte habe ich eine Klasse erstellt zum Speichern der Welt. Diese Welt wird unterteilt in 32x32 Chunks. Jeder dieser Chunks ist ein numpy array und wird indiziert in einem Dictionary mit der Koordinate des Chunks in der Welt.
Um aus einer globalen Koordiante die Koordinate des Chunks und die Koordinate eines Blocks im Chunk zu generieren ließ sich hier die divmod() Funktion nutzen.

```
chunk_x, block_in_chunk_x = divmod(x, 32) # entspricht x // 32, x % 32
```

Division- und Modulusoperationen waren vermutlich effizient genug. Dennoch entfernte ich das System relativ schnell wieder und stieg auf ein System um, in dem ein großes Dictionary alle Blöcke ohne Chunks enthält. Ein großes numpy array wollte ich nicht verwenden, da es nicht erweiterbar ist und viele Blöcke unnötig gespeichert werden würden, die der Spieler niemals sehen könnte.

Erst im November führte ich das Chunk-System wieder ein. 32 entspricht 2^5 bzw. den letzten 5 Bits einer Zahl. Alle Bits davor bestimmen die Koordiante des Chunks. Dadurch kann man hier auf Division- und Modulusoperationen verzichten.

```
chunk_x = x >> 5 # entspricht x // 32
block_in_chunk_x = x & 31 # entspricht x % 32; 31 im Binär System ist 00011111. Durch die bitwise-and-Operation besteht das Ergebnis aus den letzten 5 Bits.
```

Bis zum 11. Juni habe ich dann an einem verbesserten Rendering gearbeitet, da ich unzufrieden mit der Geschwindigkeit des vorherigen war, größtenteils aufgrund der zuvor genannten ineffizienten Konvertierung. Dabei nutzte ich, statt Pygame in Python, SDL und OpenGL in C und importierte einige Funktionen mittels Ctypes aus einer C-Datei in Python. Die Grundstruktur war dieselbe, aber die Geschwindigkeit wurde wesentlich besser. Obwohl Pygame auf SDL basiert, ist die Konvertierung von SDL-Surfaces zu OpenGL Texturen wesentlich schneller, da SDL Bilder direkt als Array bereitstellt. Das neue System funktionierte tatsächlich gut. Jedoch hatte ich in der folgenden Zeit Probleme, das Programm auf den Schulcomputern auszuführen und musste das System wieder entfernen.
Aufgrund der komplexen und zeitraubenden Installation von SDL, fing ich jedoch noch mal neu an. Dazu las ich mich zunächst tiefer in OpenGL hinein mit https://learnopengl.com und entwickelte ein neues System, das Pygame und OpenGL nutzt. Pygame stellt dabei allein das Fenster. OpenGL ist zuständig alles zu malen. Dabei befinden sich alle Bilder in einem Texture-Atlas. In zunächst vier und später drei Buffern werden dann die Daten "dest_rect", "source_or_color" und "shape_transform" gespeichert. Alle Bilder, Formen und Texte können so mittels Instanced-Rendering gemalt werden.
Alles, was gemalt bzw. gerendert werden soll kommt erst in einen Vertex-Shader und dann in einen Fragment-Shader. Zum Rendern eines Objektes bekommen diese beiden Shader Werte für "dest_rect", "source_or_color" und "shape_transform". Diese Buffer haben jeweils vier Werte, die in GLSL mit rgba bzw. xyzw indiziert werden. "dest_rect" bestimmt wird im Vertex-Shader verwendet und bestimmt den Ort, wohin das Objekt kommt und wie groß es ist. In "shape_transform" gibt der erste Wert (Red-Channel) den Typ an. Dieser kann einer der folgenden sein: Image (0), Rectangle (1), Circle (2), Text (3), Background (4) oder Foreground (5). Der zweite und dritte wert (Green- und Blue-Channel) werden verwendet um das Objekt, hier meist ein Bild, horizontal oder vertikal zu spiegeln. Der letzte Wert im Alpha-Channel gibt die Rotation des Objektes an. Der "source_or_color"-Buffer enthält ebenfalls vier Werte, die abhängig von dem Typ (Shape) des Objektes interpretiert werden. Bei einem Bild werden die Werte als Koordinaten im Textur-Atlas verwendet. Bei Rechtecken, Kreisen und Text-Objekten werden die Werte für die Farbe genutzt.
Mit diesem System können beliebig viele Objekte gemalt werden. Abhängig von der Reihenfolge befindet sich das zuvor gemalte Bild immer hinter den neueren. Es wird halbe und komplette Transparenz unterstützt. Das neue Grafiksystem konnte ich bis zum 23. Juni fertigstellen.

3.4 Weltgeneration

Zu der Zeit habe ich mir auch Gedanken über ein Weltgenerationssystem gemacht. Diese sollte auf zweidimensionalen Noisemaps basieren. Damit kann man sehr einfach und schnell organische Höhlen generieren. Später haben wir ein solches System zwar entwickelt, doch es war besser stattdessen einen einzelnen Pfad zu generieren, um einen vordefinierten Weg für den Spieler zu haben. Dabei wird die Welt in aneinandergereihten Segmenten generiert. Das erste Segment ist eine lange vertikale Höhle für das Intro. Danach werden zufällig ausgewählt horizontale Höhlen, große runde Höhlen, kürzere vertikale Höhlen und Strukturen.

Die Generation im Intro wiederholt sich alle 32 Blöcke. Dadurch kann das Intro übersprungen werden, indem der Spieler um ein Vielfaches von 32 nach unten versetzt wird, ohne dass man den Sprung bemerkt.
Basierend auf Jakobs Physik-System habe ich versucht eine Liane zu programmieren, um vertikale Höhlensegmente zu ermöglichen. An diesen sollte der Spieler hoch klettern können. Eine Liane bestand aus mehreren kleinen Segmenten, die aneinanderhängen sollten. Diese wechselseitige Interaktion funktionierte jedoch nicht, weshalb ich mich dazu entschied, sie durch statische Stangen zu ersetzen.
Relativ spät habe ich einen Strukturedитor hinzugefügt und am 18. November auch die Möglichkeit, dass Strukturen in der Welt generiert werden. Dabei wird in Zwischensegmenten der Radius der Höhle auf den Eingang und den Ausgang der Struktur mit linearer Interpolation abgestimmt.
Für eine bessere Generation von Pflanzen habe ich in den JSON-Dateien die Möglichkeit hinzugefügt, wie ein Dekorationsblock generiert. Dabei kann bestimmt werden, an welcher Seite eines Blocks, an welchem Block bzw. an welcher Block-Familie, zu welcher Wahrscheinlichkeit und ob der Block unter Wasser generiert.

3.5 Rendering der Welt

Über die Sommerferien habe ich einen Blick auf Pixel Art geworfen und die Bilder des Spielers und einigen Blöcken gemalt.
Am 11. Juli fügte ich dann erste Blöcke ein und überarbeitete das Laden von Blöcken.
Später habe ich auch Blockgruppen hinzugefügt. Diese werden automatisch erstellt, wenn das Bild eines Blocks größer als 16x16 Pixel ist. Diese Blockgruppen werden als in 16x16 Blöcke zerteilt und in der Weltgeneration passend angeordnet, da beim Rendern der Welt nur 16x16 Blöcke unterstützt werden.
Anfang August baute ich animierte Blöcke und Bilder ein. Das Rendering von der Welt ist dabei anders als das des Spielers. Um die Welt zu malen, wird ein sichtbarer Teil der Welt genommen und in eine Textur geschrieben, damit die Blöcke im Fragment-Shader gleichzeitig gemalt werden können. Eine weitere Textur enthält dafür alle Blöcke. Um alles andere zu malen, werden zwei weitere Texturen verwendet, die alle Bilder und eine Auswahl an Buchstaben und Symbolen enthalten. Diese drei Texturen werden auch für testzwecke im Ordner "data" gespeichert.
Nach einigen Änderungen habe ich ein System entwickelt, in dem die Welt aus 4 Lagen besteht. Für die ersten drei Lagen wird für jeden generierten Block eine Ganzzahl gespeichert, die der Index des Blocks in der Textur der Blöcke ist. Die erste Lage enthält alle Blöcke, mit denen der Spieler kollidieren kann. In der zweiten Lage sind Blöcke, die in der Rendering-Reihenfolge nach dem Spieler und damit über dem Spieler gemalt werden. Die dritte Lage enthält Blöcke, die hinter dem Spieler gemalt werden. In der vierten Lage wird der Wasserstand des Blocks gespeichert als Ganzzahl zwischen -1000 und 1000, wobei das Vorzeichen angibt, auf welcher Seite das Wasser sich im Block tendenziell befindet. Damit habe ich über mehrere Wochen verteilt auch an dem Rendering von Wasser gearbeitet. Obwohl die Mechaniken des Wassers und das Rendering sehr gut waren, ist fließendes Wasser kaum im Spiel vorzufinden, da ich es nicht gut in die Weltgeneration einbinden konnte.
Später habe ich Blöcke hinzugefügt, die als kleinere Blöcke die Ecken der normalen Blöcke füllen.

3.6 Schatten

Bis zum 10. September programmierte ich ein System für Schatten. Damit werden von der Spielfigur aus gesehen Schatten hinter Blöcken geworfen. Es verlangsamt das Spiel jedoch und ist daher standardmäßig ausgeschaltet, kann aber in den Optionen eingeschaltet werden. Es läuft in Echtzeit und unterstützt beliebig viele Lichtquellen, wobei der Spieler als einzige "Lichtquelle" schon intensiv genug ist. Zwischenzeitig habe ich allerdings überlegt, die Schatten nur einmalig zu rendern mit statischen Fackeln als Lichtquellen, was ich jedoch nicht mehr umgesetzt habe.
Um Schatten zu malen, müssen diese zunächst gefunden werden. Dazu muss eine Liste mit allen Kanten generiert werden. Dabei werden gerade Kanten mehrerer Blöcke zusammengefasst. Zunächst werden Eckpunkte gesucht. Dafür werden im sichtbaren Bereich der Welt alle Punkte in der Mitte von vier Blöcken gesucht, bei denen 1 oder 3 benachbarte Blöcke Luft (0) sind und die anderen 1 bzw. 3 Blöcke nicht Luft sind.

```
# Daten im Array verschieben
view_shifted_right = numpy.roll(view, shift=-1, axis=0)
view_shifted_down = numpy.roll(view, shift=-1, axis=1)

# Array mit der Anzahl der benachbarten Luftblöcke erstellen
count_air_blocks = (view == 0).astype(int) + (view_shifted_right == 0).astype(int) +
(view_shifted_down == 0).astype(int) + (view_shifted_diagonal == 0).astype(int)

# Liste mit Eckpunkten erstellen
corner_indices = numpy.where((count_air_blocks == 1) | (count_air_blocks == 3))
corners = list(zip(corner_indices[0], corner_indices[1]))
```

Diese Liste kann entlang der x-Achse sortiert werden. Wenn die Eckpunkte in Zweierpaaren zusammengefasst werden, erhält man eine vollständige Liste aller horizontalen Kanten. Gleiches wird für die y-Achse ausgeführt.

```
# Vertikale Kanten
corners.sort(key=lambda corner: (corner[0], corner[1]))
flattened_corners = [coord for corner in corners for coord in corner]
edges = flattened_corners

# Horizontale Kanten
corners.sort(key=lambda corner: (corner[1], corner[0]))
flattened_corners = [coord for corner in corners for coord in corner]
edges.extend(flattened_corners)
```

Danach werden die Eckpunkte sortiert, basierend auf ihrem Winkel zur Lichtquelle. Dabei werden alle Eckpunkte ausgeschlossen, bei denen ein imaginäres Liniensegment zur Lichtquelle mit einer anderen Kante kollidiert. Dabei wird zusätzlich zu beiden Seiten des Winkels je ein Strahl geworfen und der Punkt, wo der Strahl aufkommt, wird ebenfalls in die Liste aufgenommen.
Um Schatten zu malen, wird statt den Schatten zunächst das Licht in Dreiecken gemalt. Dazu formen immer ein Punkt der sortierten Eckpunktliste, der vorherige Punkt der Eckpunktliste und die Lichtquelle ein Dreieck. Dieses wird auf einer Pygame-Surface gemalt, die zu einer Textur konvertiert wird. Dabei kann man auch nur den Red-Channel kopieren und alle anderen ignorieren.

```
numpy_data = pygame.surfarray.pixels_red(pygame_surface).transpose()
```

Diese Konvertierung ist dennoch der Hauptgrund, warum das System langsam läuft. Hätten wir auf OpenGL verzichtet, für das wir uns hauptsächlich wegen der Schatten und Shader entschieden haben, liefe das Programm mit Schatten vermutlich schneller.

3.7 Steuerung

Verteilt über die Zeit der Entwicklung habe ich immer wieder an der Bewegung des Spielers gearbeitet, die nun recht komplex ist. Bei Standardeinstellungen kann der Spieler mit A und D laufen und mit der Leertaste springen. Wenn der Spieler läuft, kann er mit der Shift-Taste sprinten oder mit C kriechen. Wenn der Spieler in der Kriechposition ist, kann mit der Leertaste ein besonders weiter Sprung aufgeladen werden. Wenn der Spieler gegen eine Wand springt, die zwei oder drei Blöcke hoch ist, kann er sich an der Kante festhalten mit der Lauftaste zu Wand hin (A oder D) und sich mit der Leertaste hochziehen. Befindet sich der Spieler an einer Wand und beide Lauftasten werden gedrückt, springt der Spieler von der Wand ab.

3.8 Kreaturen

Erst im November habe ich erste Gegner hinzugefügt. Der erste Gegner war eine grüne Kugel, "Slime" genannt. Dazu habe ich eine Gesundheitsanzeige für den Spieler hinzugefügt und die Möglichkeit Pfeile zu schießen.
Am 19. November habe ich "Goblins" als weiteren Gegner hinzugefügt und bis zum 21. November auch Fledermäuse. Bis zum 22. November habe ich alle 5 Waffen gemalt und ein Inventar hinzugefügt. Dieses kann mit E geöffnet werden. Darin kann man Waffen auswählen und zerstören, um Leben zurückzugewinnen.
In der Zeit bis zum 25. November habe ich 14 Attribute für Waffen hinzugefügt. Diese verbessern Waffen auf verschiedenen Wegen. Jede Waffe hat genau zwei Attribute. Zwei Waffen können im Inventar kombiniert, wenn bei ihnen mindestens ein Attribut übereinstimmt, um das Attribut auf einer Waffe um ein Level zu erhöhen.
Danach habe ich einen kleinen Editor gebaut, um die Position und Rotation von Waffen in der Hand des Spielers auf alle Animationen abzustimmen.

4. Fazit

Das Ziel des Spiels ist, aus den Höhlen lebendig zu entkommen, was mit dem ersten Versuch vermutlich nicht gelingt. Dabei muss der Spieler mit Monstern kämpfen, kann Waffen finden und verbessern und die Welt erkunden.
Für mich ist erstaunlich, dass ein Spiel in diesem Umfang immer noch schnell läuft. Ich denke, die Entwicklung unseres Spiels „Leafy Hollows“ war erfolgreich. Zwar sind wir von unseren ursprünglichen Ideen abgekommen, doch konnten wir mit der gesammelten Erfahrung ein gutes Ergebnis erzielen. Denn beim Arbeiten kamen uns immer neue Ideen, die wir einbauen wollten. Viele dieser Ideen haben wir verwerfen müssen, andere haben wir aufgenommen und umgesetzt. Die Arbeit im Team war ungewohnt und wir hätten öfter über unsere Ziele kommunizieren können. Dennoch konnten wir gut zusammenarbeiten und uns gegenseitig unterstützen und motivieren.
Während der Entwicklung konnte ich einiges lernen, vor allem auf dem Gebiet von OpenGL, Shadern und Schatten. Außerhalb von der Programmierung konnte ich auch vieles über Pixel Art und Sound Design lernen, da ich alle 287 Bilder selbst gemalt habe und viele der Sounds selbst aufgenommen oder bearbeitet habe. Auch wenn man natürlich immer wieder an einen Punkt kommt, an dem man verzweifelt, hat die Programmierung mir Spaß gemacht. Meine Motivation war dabei immer, ein Spiel zu entwickeln, das mir selbst gefällt und das ich selbst auch länger spielen könnte.