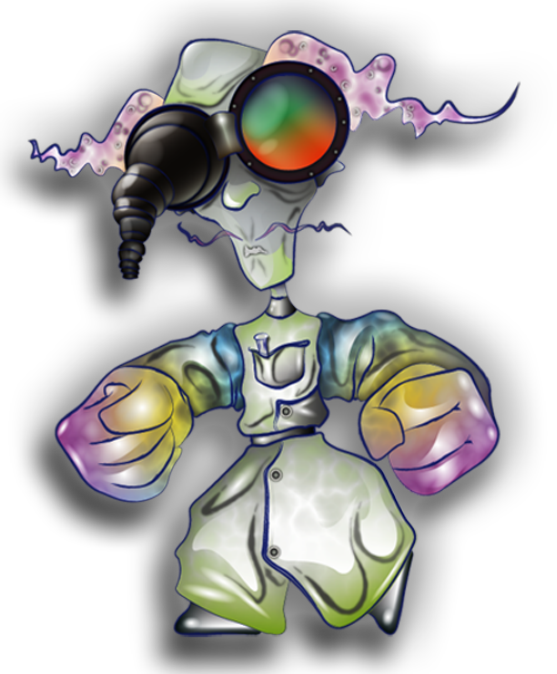
Hochschule Hamm-Lippstadt

Computervisualistik und Design



Crazy Doc's Evolution Run

Entwicklung eines Computerspiels in der Unity 3D Engine

Dokumentation

(Zeitraum vom 17.03.2014-16.07.2014)

Verfasser:
Ahmed Arous
Christoph Duda
Hans Ferchland
Daniel Geblinski
Stefan Hartwig
Marcel Müller

Betreuer:
Prof. Dr. Christian Sturm

Modul:
Projektarbeit SS 14
(CVD-B-2-6.01.SS 14.001)

Inhaltsverzeichnis

Projektbeschreibung

Ahmed Arous

Einleitung

Der Studienverlauf des Bachelorstudiengangs Computervisualistik und Design der Hochschule Hamm-Lippstadt sieht die Durchführung eines Softwareprojekts im 6. Semester vor. Ziel dieser Projektarbeit ist das eigenständige Erarbeiten einer ergebnisorientieren und praxisbezogenen Problemlösung.
Das Thema der Projektarbeit ergab sich einerseits durch das persönliche Interesse der beteiligten Mitglieder sowie der Komplexität der Anforderung und umfasst die Entwicklung eines Videospiels, angefangen bei der Erstellung des Spielkonzepts bis hin zur exportierten spielbaren Version.
Bei der Ausführung des Projekts wurde großen Wert darauf gelegt realistische Szenarien zu erstellen um möglichst Industrieüblich zu agieren, dies zeigt sich im Ansatz bei der Durchführung und Organisation. Dabei konnten vor Allem die erlangten Erfahrungen aus dem Praxissemester einfließen.
Im Rahmen dieses Moduls ist die Spielesoftware "Crazy Doc's Evolution Run" als Prototyp für einen Action-Platformer entstanden.
Im weiteren Verlauf dieser Dokumentation wird im Einzelnen auf alle beteiligten Personen und allen Aspekten der Entwicklung eingegangen und das erlangte Ergebnis beurteilt.

Team und Verantwortlichkeiten

In der ersten Phase des Projekts bestand das Team aus Hans Ferchland, Christoph Duda und Ahmed Arous. In dieser Phase wurden, bewusst in kleiner Runde, erste Ideen zu möglichen Spielkonzepten ausgetauscht.
Nachdem man sich grob auf ein Konzept geeinigt hat und ein Betreuer für das Projekt gefunden wurde, war es möglich die anfallende Arbeit einzuschätzen und weitere Kommilitonen für die Projektarbeit zu gewinnen.
Dabei war es außerordentlich wichtig das jeder im Team gefallen am vorgeschlagenen Konzept gefunden hat und über das Projekt hinaus Interesse an der Entwicklung von Spielen und Erfahrungen mit diesen aufweist, um somit frühzeitig möglichen Motivationsproblemen auszuweichen.
Aus diesem Grund wurde auch von Beginn an auf die Interessen und Fähigkeiten jedes Beteiligten Rücksicht genommen.
Mit jedem weiteren Mitglied sind die selbstgestellten Anforderungen im Spielkonzept ausgebaut worden um sicher zu stellen, dass alle ausgelastet sind und sich die Möglichkeiten im Spiel steigern. Dabei konnte sich jeder einbringen.

Darüber hinaus haben sich die Verantwortungen wie folgt ergeben:

Programmierung

- Ahmed Arous
  Character Controlling, Input Handling, Dynamische Levelelemente, Sound

- Hans Ferchland
  Levelgenerierung, Sessions, GUI, Kamera, Dynamische Levelelemente, Gamestates, Spielerlogik

Art

- Christoph Duda
  Character Assets (3D-Modellierung, Texturierung, Animation), VFX (Shader, Partikelsysteme), SFX (Titelmusik, Feedbacksounds)

- Marcel Müller
  Level Assets (3D-Modellierung, Texturierung, Animation), Leveldesign

- Daniel Glebinski
  Character Assets (3D-Modellierung, Texturierung, Animation), VFX (Shader,Partikelsysteme)

- Stefan Hartwig
  Concept Art, 2D Assets (GUI Elemente, Logo,...)

Jeder war in einem gewissen Maße am Game Design und Projektmanagement beteiligt.

Organisation

Ahmed Arous

Technik

- Game Engine
  Aufgrund der angestrebten Ziele und dem engen Zeitplan sollte eine Game Engine als Entwicklungsbasis dienen. Hierzu wurden mehrere bekannte Vertreter betrachtet wie z.B. die Unreal Engine, CryEngine und Unity. Ausschlaggebend für die Entscheidung Unity zu nutzen war einerseits die beworbene Einsteigerfreundlichkeit, die simple Lizensierung, sowie das Bedürfnis in einer klassischen objektorientierten Sprache wie C# arbeiten zu wollen. Die Unreal Engine befand sich zu diesem Zeitpunkt in einer Umbruchphase mit der Betaversion der Unreal Engine 4 und die CryEngine hätte mit seinen grafischen Möglichkeiten einen anderen Fokus gehabt als wir ihn benötigt hätten. Mit ins Gewicht fiel natürlich auch der seit Jahren anhaltende Trend der Unity 3D Engine auf dem deutschen Markt und die Verfügbarkeit an den Hochschulrechnern.

Tools

- GitLab
  Eine Open Source Software um ein Git Repository zu managen.

- Skype
  Telefonie und Messaging-Software, die sowohl stationär wie auch mobil zur Kommunikation diente

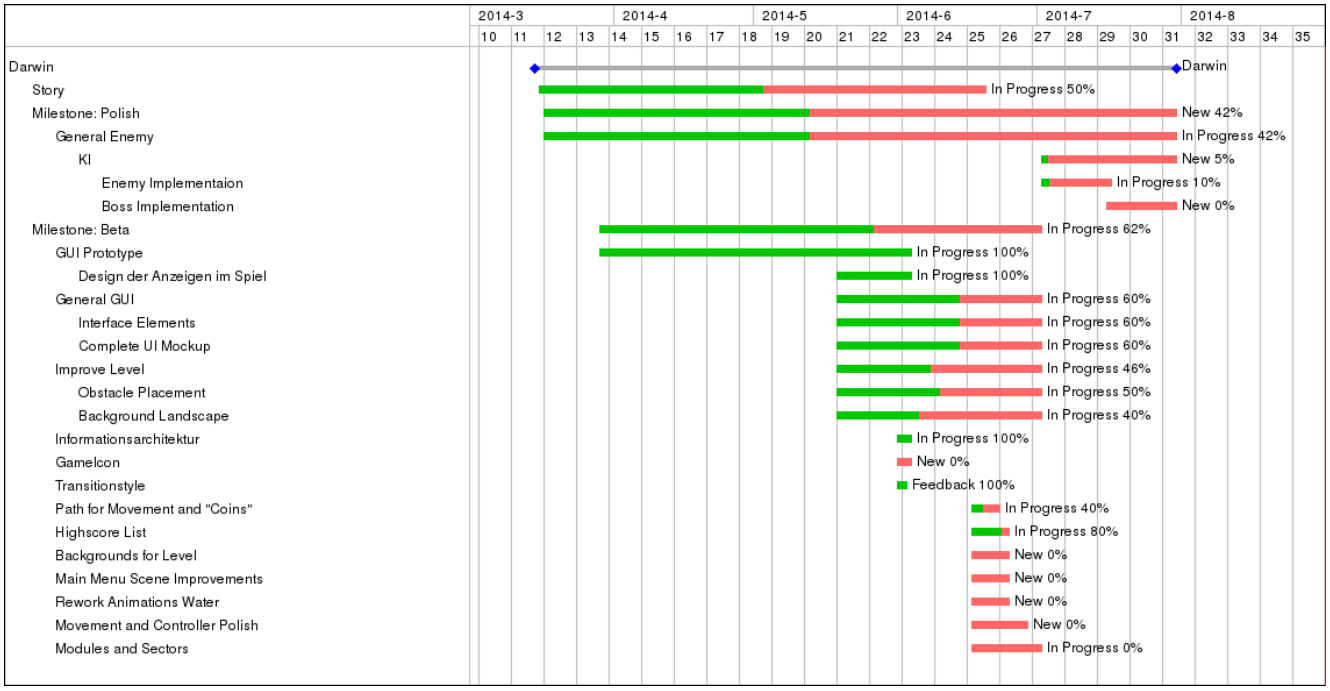
- Google Drive
  Die von Google angebotenen Webanwendungen, die genutzt wurden um Dateien untereinander auszutauschen und um gemeinsam in Echtzeit an Dokumenten zu arbeiten.

- Blender, Cinema 4D
  Alle 3D Assets die nicht in Unity selbst erstellt wurden, entstanden in eines dieser 3D-Programme.

- FL Studio
  FL Studio ist ein patternbasierter Software-Sequenzer, der für die Titelmusik und Soundeffekte zum Einsatz kam.

Projektplanung

Nachdem alle Rahmenbedingungen festgelegt worden sind begann die Entwicklung. Im Zeitraum vom 17.03.2014 - 16.07.2014 sollte eine möglichst vollständige Version des aufgestellten Spielkonzepts realisiert werden.
Zur Planung wurde die Webanwendung "Redmine" genutzt. Auf Redmine wurden Arbeitspakete und Meilensteine definiert und zugewiesen. Jeder war angehalten seine Tasks selbständig zu aktualisieren, sodass für alle der Arbeitsstand und der Projektplan stets einsehbar waren.
In der Anfangsphase war dies sehr hilfreich um einen gemeinsamen Arbeitrhythmus zu finden, allerdings wurden nach dem Auftritt eines Serverausfalls weitere Arbeitspakete auf agilerem und direkterem Wege verteilt, denn der gefühlte Mehrwert in unserer kleinen Gruppe entsprach nicht dem anfallenden Aufwand für das Erstellen und Verwalten der Tickets.



Gannt-Diagramm auf Redmine

Darwin - Tickets

#	Priorität	Aktualisiert	Projekt	Autor	% erledigt	Zugewiesen an	Thema	Tracker	Status	Beginn	Abgabedatum
86	High	17.06.2014 23:09	Darwin	Hans Ferchland	0	Marcel Müller	Modules and Sectors	Feature	In Progress	17.06.2014	01.07.2014
85	High	17.06.2014 23:04	Darwin	Hans Ferchland	0	Ahmed Arous	Movement and Controller Polish	Feature	New	17.06.2014	28.06.2014
84	High	17.06.2014 23:03	Darwin	Hans Ferchland	0	Daniel Glebinski	Rework Animations Water	Feature	New	17.06.2014	24.06.2014
83	High	17.06.2014 23:01	Darwin	Hans Ferchland	42	Hans Ferchland	Milestone: Polish	Feature	New	17.03.2014	30.07.2014
82	High	17.06.2014 22:53	Darwin	Hans Ferchland	0	Chris Duda	Main Menu Scene Improvements	Feature	New	17.06.2014	24.06.2014
81	High	17.06.2014 22:54	Darwin	Hans Ferchland	0	Chris Duda	Backgrounds for Level	Feature	New	17.06.2014	24.06.2014
80	High	22.06.2014 15:42	Darwin	Hans Ferchland	80	Hans Ferchland	Highscore List	Feature	In Progress	17.06.2014	24.06.2014
79	High	22.06.2014 15:42	Darwin	Hans Ferchland	40	Hans Ferchland	Path for Movement and "Coins"	Feature	In Progress	17.06.2014	22.06.2014
78	Normal	17.06.2014 23:10	Darwin	Stefan Hartwig	100	Stefan Hartwig	Transitionstyle	Feature	Feedback	01.06.2014	02.06.2014
77	Normal	17.06.2014 22:57	Darwin	Stefan Hartwig	0	Stefan Hartwig	GameIcon	Feature	New	01.06.2014	03.06.2014
76	Normal	17.06.2014 22:57	Darwin	Stefan Hartwig	100	Stefan Hartwig	Informationsarchitektur	Feature	In Progress	01.06.2014	03.06.2014
75	High	17.06.2014 22:57	Darwin	Hans Ferchland	46	Hans Ferchland	Improve Level	Feature	In Progress	19.05.2014	01.07.2014
74	High	22.06.2014 15:42	Darwin	Hans Ferchland	62	Hans Ferchland	Milestone: Beta	Feature	In Progress	29.03.2014	01.07.2014
73	High	01.06.2014 14:55	Darwin	Stefan Hartwig	100	Stefan Hartwig	Design der Anzeigen im Spiel	Feature	In Progress	19.05.2014	03.06.2014
72	Normal	13.05.2014 13:47	Darwin	Stefan Hartwig	100	Stefan Hartwig	Entwürfe des GManipulator	Feature	Closed	04.05.2014	07.05.2014
71	High	13.05.2014 13:47	Darwin	Stefan Hartwig	100	Stefan Hartwig	Elementenplatzierung	Feature	Closed	03.05.2014	07.05.2014
70	High	13.05.2014 13:43	Darwin	Hans Ferchland	100	Hans Ferchland	Character Spawning	Feature	Closed	22.04.2014	01.05.2014
68	High	13.05.2014 13:48	Darwin	Stefan Hartwig	100		Entwurf/Elementenplatzierung	Feature	Closed	02.04.2014	
67	Normal	13.05.2014 13:45	Darwin	Hans Ferchland	100	Ahmed Arous	Controller Implementation	Feature	Closed	30.03.2014	01.05.2014
66	High	13.05.2014 13:42	Darwin	Hans Ferchland	100	Hans Ferchland	Player Implementation	Feature	Closed	30.03.2014	01.05.2014
65	Normal	20.05.2014 13:19	Darwin	Daniel Glebinski	100	Chris Duda	3D Modelling	Feature	Closed	30.03.2014	19.05.2014
64	Normal	20.05.2014 13:19	Darwin	Daniel Glebinski	100	Chris Duda	Locatergontula als Character zulassen	Feature	Closed	30.03.2014	19.05.2014
63	Normal	13.05.2014 13:47	Darwin	Daniel Glebinski	100		Concept Art	Feature	Closed	30.03.2014	
62	Normal	20.05.2014 13:29	Darwin	Daniel Glebinski	100		Character - Luft - Locatergontula	Feature	Closed	30.03.2014	19.05.2014
61	High	20.05.2014 13:29	Darwin	Hans Ferchland	100		Level gameplay, enemies, fighting, evolution	Feature	Closed	20.03.2014	19.05.2014
60	High	01.06.2014 14:55	Darwin	Hans Ferchland	100		GUI Prototype	Feature	In Progress	29.03.2014	03.06.2014
59	High	17.06.2014 22:56	Darwin	Hans Ferchland	100	Hans Ferchland	Multiplayer gameplay	Feature	Closed	19.05.2014	01.06.2014
58	High	20.05.2014 13:29	Darwin	Hans Ferchland	100		Create three characters, one for each level	Feature	Closed	17.03.2014	19.05.2014
57	High	20.05.2014 13:28	Darwin	Hans Ferchland	100	Hans Ferchland	Level Gerenerating	Feature	Closed	16.03.2014	01.05.2014
56	High	20.05.2014 13:30	Darwin	Hans Ferchland	100	Hans Ferchland	Milestone: Alpha	Feature	Closed	15.03.2014	19.05.2014
55	Normal	20.04.2014 14:08	Darwin	Stefan Hartwig	100	Stefan Hartwig	GegnerEndEntwurf/Wasser	Feature	Closed	28.03.2014	11.04.2014
54	Normal	13.05.2014 13:46	Darwin	Stefan Hartwig	100	Stefan Hartwig	GegnerEndEntwurf/Luft	Feature	Closed	28.03.2014	04.04.2014
53	Normal	13.05.2014 13:55	Darwin	Daniel Glebinski	100	Daniel Glebinski	Animation	Feature	Closed	20.03.2014	15.04.2014
52	Normal	13.05.2014 13:47	Darwin	Daniel Glebinski	100	Daniel Glebinski	Rigging	Feature	Closed	07.04.2014	19.04.2014
51	Normal	13.05.2014 13:46	Darwin	Daniel Glebinski	100	Daniel Glebinski	Texturing	Feature	Closed	31.03.2014	14.04.2014

Eine Auswahl von Tickets auf Redmine

Meetings

Das Team hat sich ein mal die Woche persönlich in der Hochschule getroffen und gemeinsam im PC-Pool gearbeitet. Zusätzlich gab es am Ende jeder Woche eine Telefonkonferenz bei der jeder seinen Fortschritt mitteilen und man offene Fragen klären konnte.

Feedback

Es wurde eine Projekt Webseite aufgesetzt mit einem Blog. Dort sollen Meinungen zu dem Spiel eingeholt werden. Außerdem wurden Kommilitonen in der Hochschule regelmäßig befragt.

Die Webbasierten tools wie GitLab, Redmine sowie die Webseite wurden auf einem privaten Server gehostet und von Hans Ferchland aufgesetzt und gewartet.
Alle relevanten Links sind aufrufbar auf https://darwin.void-ptr.org/ [Stand 31.08.14].

Game Design

Um was für ein Spiel handelt es sich?

Bei "Crazy Doc's Evolution Run" handelt es sich um kompetitiven 2.5D Action Platformer. Zwei Spieler treten in einem Rennen gegeneinander an und müssen dabei tödlichen Hindernissen ausweichen.

Warum gerade dieses Spiel?

Das Konzept zu "Crazy Doc's Evolution Run" entstand im gemeinsamen Austausch von Spielideen und ist genau genommen ein Kompromiss der sich aus diesen ergab. Es ist jedoch keine unüberlegte Kombination verschiedener Elemente sondern eine Anlehnung an erfolgreichen Indie Games der letzten Jahre wie z.B. Flappy Bird, Guacamelee, Castle Crashers und bringt in seinem Kern frischen Wind in das klassische Platformer Genre. Bei der Erstellung des Konzepts war es entscheidend das zumindest ein spielbarer Prototyp von unserem noch unerfahrenem Team entwickelt werden konnte.

Was steuert man im Spiel?

Der Spieler schlüpft in die Rolle einer von einem Fanatischen Professor künstlich mutierten Kreatur und muss sich nun im Kampf um seine Existenz beweisen.

Wo findet das Spiel statt?

Das Spiel findet in sogenannten Testsektoren statt, das sind abgeschottete Gebiete in denen der Professor seine Kreaturen entlässt um diese auf ihre Anpassungsfähigkeiten zu testen. Testsektoren sind eine Inszenierung einer Lebensfeindlichen Umgebung und vom Professor eigens entwickelt. Überall droht Gefahr.

Was ist unser Hauptfokus?

Crazy Doc’s Evolution Run soll vorallem Spaß bringen. Die Besonderheit hier ist das Steuerungkonzept welches eine zentrale Rolle einnimmt. Als Local CO-OP Game also, als ein Spiel, dass gemeinsam am selben Ort gespielt wird, lebt das Konzept durch die Interaktion zwischen den Spielenden. Der regelmäßige Wechsel der Steuerung, sowie Elemente die Spielerübergreifend reagieren, sollen die Spieler belustigen. Im Spielverlauf werden zusätzlich durch Rankings kompetitive Spieler angesprochen.

Unity Engine

Hans Ferchland

Überblick

Die Unity Engine ist eine 2D/3D Spiele-Engine mit vielen Tools, Erweiterungen, weiter Verbreitung und guter Dokumentation. Sie ist in zwei Versionen verfügbar, die sich im Umfang und der Lizenz unterscheiden. Neben diesen beiden Hauptversionen gibt es aber auch noch Lizenzen für die Lehre, das Studium, etc. Von uns genutzt wurde die freie Version “Unity Basic”, sowie eine Lehrversion “Unity Pro” die uns an der Hochschule zur Verfügung stand.
Hier soll nun kurz der Editor und die generelle Arbeitsweise mit der Engine dargestellt werden.

Editor

Der Editor ist das Herz der Engine und verbindet alle Bereiche die in der Erstellung eines Spiels beteiligt sind. Er ist in seinem Layout (verschiedene Views) konfigurierbar und über Scripts vielseitig erweiterbar.

http://docs.unity3d.com/Manual/LearningtheInterface.html

Die Views beinhalten Tools die spezielle Funktionen für verschiedene Aufgaben (Level-Erstellung, Animation, Assetverwaltung, Speichern, Vorschau, etc.) enthalten.

Die entsprechenden Bereiche wurden im Bild markiert(“Scene”, “Inspector”, “Project”, “Hierachie”, “Toolbar”).
Funktionen wie das und Suchen von Assets, das Anzeigen an welcher Stelle welche Assets benutzt werden, Shortcuts und mehr sind meist gut zu sehen und benutzen, manches ist aber auch gut versteckt. Gut sichtbar ist aber die Toolbar mit dem “Play” Button, der das Projekt in der aktuellen Szene startet.
MonoDevelop ist die bevorzugte IDE für Unity. Visual Studio und andere Compiler werden aber auch unterstützt, notwendig sind sie nicht. Unity unterstützt das Scripting in C# (.NET ähnliche Syntax wie Java/C++), Unity Script (Javascript Derivat) und Boo (.NET und Python-ähnlicher Syntax).
Spezielle Script-Tools sind etwa die Console, der Profiler (nur in Unity Pro, siehe Register) und Logging plus Log-Files.

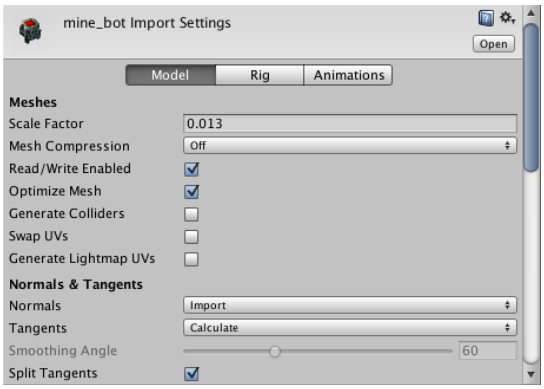
Architektur

Ein Spieleprojekt besteht aus den Assets (eigene Scripts, Modelle, Audiodateien, Texturen, Text, etc.) und mindestens einer Szene. Jedes Projekt hat individuelle Einstellungen für Audio, Input, Netzwerk, Grafik, Qualität, Script-Order, uvm.
Das Gerüst der Engine baut sich um die Klasse GameObject, ScriptableObject, Transform und die verschiedenen Asset Components (Audio, Texturen, Modelle, etc.). Dazu kommen spezielle Komponenten wie Kamera, Lichter, Trigger- und Kollisions-Volumen oder GUI Layer.
Alle Komponenten finden sich in der “Scene” bzw. “Hierachy” wieder die als persistentes Objekt (oder auch Level) gespeichert werden.
GameObjects können in der Szene in der “Hierachy” hierarchisch angeordnet werden und können so abhängig in ihrer Transformation und Sichtbarkeit sein.
ScriptableObject bietet die Möglichkeit eigene Komponenten mit speziellem Verhalten zu Erstellen, die im Editor dann individuell genutzt werden können. So kann schnell Gameplay-Verhalten erstellt und iteriert werden.

Konzepte

- Audio Clips: Sind importierte Audiodateien (.aif, .wav, .mp3 plus .xm, .mod, .it und .s3m) die von Audio-Quellen genutzt werden können.
- Fonts: Importierte Schriftarten (.ttf oder .otf) die für GUI oder Text-Meshes (3D Text) genutzt werden können.
- Meshes: Importierte 3D Polygon-Modelle entweder als exportierte Datei (.fbx, .dae, .3DS, .dxf und .obj) oder direkt als Anwendungs-Datei (.blend, .max, .mb, .ma)

- Animationen: Werden automatisch beim Modellimport mit-importiert.
- Materials: Wird bei Bedarf auch bei Meshes mit-importiert. Können aber auch für Partikel-Systeme genutzt werden. Materialien haben immer einen Shader assoziiert.



- Render Textures: Spezielle Texturen die erst zur Laufzeit mit einem beliebigen Kamerabild beschrieben werden können und sonst wie 2D Texturen funktionieren.
- Lighting: Beleuchtung kann mit verschiedene Licht-Typen (Spot, Point, Directional Light) erstellt werden. Optional kann man Lightmapping nutzen um Performance und/oder Qualität zu erhöhen.
- Shaders: Vertex-, Fragment- und Surface-Shader bestimmen das Verhalten beim Rendern. Unterstützt werden Cg und HLSL und ShaderLab. Shader werden mit Hilfe von MonoDevelop editiert.
- Cameras: Rendern den sichtbaren Ausschnitt der Spiel-Szene. Können durch viele Effekte (Antialiasing, DoF, Bloom, HDR, etc.) beeinflusst werden. Es können mehrere Kameras in einer Szene vorhanden sein, je nach dem wie groß der Viewport der Kameras ist. Auch für Render-Textures werden Kameras benötigt.
- Physics: wird durch Rigidbodies und Collider abgebildet. Rigidbodies halten die Eigenschaften wie die Masse und Widerstand, die Collider die Größe und Form des Objekts.

- Prefabs: Ist ein beliebiger Teil der Szenen-Hierachie, beliebig viele Game-Objekte, exportiert und als Asset wiederverwendbar, und auch zur Laufzeit instanziierbar.
- Szenen: Sind die Dateien die alle vorher genannten Konzepte und deren Komponenten beinhalten können. Es gibt zum Spielstart eine Start-Szene, alle weiteren Szenen werden nachgeladen oder gewechselt.
- Plugins: sind wie auch Modelle im Asset Store erhältlich und erweitern die Funktionen von Untiy. Wichtige Plugins sind etwa Shader-Forge oder der Unity Serializer.

Workflow

Die Engine bietet für die verschiedenen Aufgaben die in der Spieleentwicklung anfallen Tools zur Bewältigung der Aufgabe. Hier sollen kurz die verschiedenen Workflows die dazu nötig sind erklärt werden.

Modelle und Animation

Da Modelle und Animationen in vielen Modellierungsprogrammen erstellt werden können ist es wichtig die Kompatibilität des Genutzten zu Unity sicherzustellen. Wenn das Programm die entsprechenden Formate (.fbx, .dae, .3DS, .dxf und .obj) exportieren kann oder die Dateiformate der Anwendung (Max, Maya, Blender, Cinema4D, Modo, Lightwave & Cheetah3D) unterstützt werden können die Modelle von Unity genutzt werden.
Ist das der Fall, kann die Datei einfach via Drag'n'Drop in das gewünschte Verzeichnis, in der Unity-Asset Struktur im View "Project", gezogen werden. Danach müssen Materialien und Texturen des Modells und bestehende Animationen für die weitere Verwendung in Unity angepasst werden.

Audio

Unterstützte Audio-Dateien können einfach via Drag'n'Drop in das gewünschte Verzeichnis, in der Unity-Asset Struktur im View "Project", gezogen werden.
Audio-Sources können dann beliebige importierte Audio-Assets abspielen.

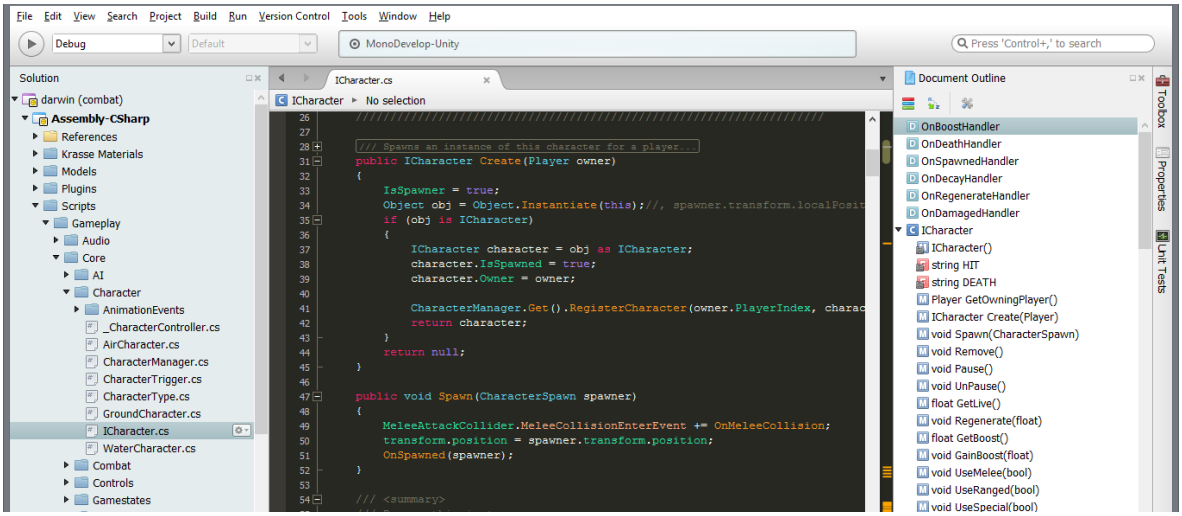
Texturen

Wie auch Audio-Dateien werden Texturen via Drag'n'Drop in das gewünschte Verzeichnis, in der Unity-Asset Struktur im View "Project", gezogen.
Render Texturen werden direkt in Unity erstellt. Alle importierten Texturen können für Materialien, Partikeleffekte oder in der GUI genutzt werden.

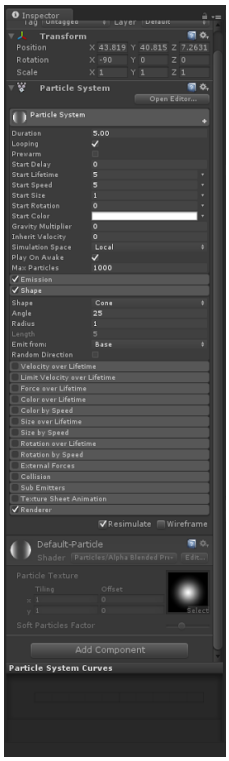
GUI

Die GUI kann in Unity relativ simpel durch vorhandene Assets wie GUI-Text und GUI-Texture im Editor erstellt werden. Events wie Klicks oder Tasten müssen über das Scripting implementiert werden. Einfache GUI-Animationen kann man über den integrierten Animations-Editor hinzufügen.
Auch eigene GUI-Elemente können mit Hilfe von Scripts erstellt werden, genutzte Texturen müssen vorher importiert worden sein.

Scripting



Hier soll nur das Scripting in MonoDevelop (oder auch Mono) erläutert werden. Scripts können im View "Project" oder direkt an GameObjects mit "Add Component" erstellt werden. Die Datei öffnet sich durch Doppelklick automatisch in MonoDevelop, dort sieht man alle Scriptdateien des Unity Projekts und die just geöffnete.
Die IDE bietet mit Code-Vervollständigung, Outliner, Debugger, Refactoring und mehr alles was man braucht in einer modernen Entwicklungsumgebung.
Der Debugger kann dynamisch mit dem Editor verknüpft werden um zur Laufzeit oder auch im Editor Fehler zu finden. Je nach Komplexität und Aufbau des Projekts kann man auch Scripts während der Laufzeit editieren, sie werden dann automatisch neu geladen, solange sie korrekt sind. Übersetzungsfehler werden sowohl in Mono als auch in Unity im "Output" angezeigt, durch Klick springt man dann zur Fehlerquelle. Verschiedene Scriptsprachen werden in Assemblies zusammengefasst, genau wie Gamelogik und Editorlogik in jeweils einer eigenen Assembly landen. So kann der Programmcode besser strukturiert werden, ist besser wartbar und erweiterbar.



Levelerstellung

In Unity beschreibt jede einzelne Szene ein Level. Der Inhalt der Szene besteht aus Game-Objekten, je nach Level können das einfach nur Game-Logik Objekte oder auch Modelle, GUI, Kameras, Lichter und Partikeleffekte sein. Unity Prefabs bieten darüber hinaus auch die Möglichkeit wiederkehrende Objekte zunächst einzeln zu erstellen um es dann mehrmals in der Szene zu platzieren. Der Vorteil daran ist, wenn sich ein Objekt ändert, kann man die

Änderungen auf die anderen Objekte übertragen. In der Hierarchie kann man die Abhängigkeit der Objekte voneinander modellieren, die Szene kann baumartig Strukturiert werden.

Effekte

Effekte gibt es in viele Arten, hier sollen Partikeleffekte und Post-Processing Effekte beleuchtet werden. Unity hat zwei verschiedene Partikelsyteme. Zum einen das "Partikelsystem" und zum Anderen verschiedene "Emitter" in Verbindung mit einem "Partikel Renderer". Für Partikelsysteme bietet Unity einen umfangreichen Editor, wo man zahlreiche Details wie Kollision, Events, Rotation, Farbe und mehr konfigurieren kann.
Emitter und Partikel Renderer müssen manuell als Komponente zu einem GameObject hinzugefügt werden. Sie haben keinen so komfortablen Editor wie die Partikelsysteme.
Post-Processing Effekte werden in Unity einfach zu einer Kamera als Komponente hinzugefügt. Ist keine Kamera vorhanden, wird sie automatisch hinzugefügt. "Unity Pro" bietet dabei die wesentlich größere Auswahl an Screen-Effekten.

Einschätzung & Zusammenfassung

Unity macht den Einstieg sehr einfach, bietet eine gute Dokumentation und hat eine große Community - gleichzeitig büßt es wenig in Konfigurationsmöglichkeiten ein und ist gut erweiterbar. Der "Asset Store" bietet viele kostenpflichtige und kostenlose Plugins, Modelle oder Effekte an. Durch mehrere Scriptsprachen, Versionen und Plattformen ist man für fast jedes Gerät, Budget und persönliche Präferenz mit Unity gut bedient.

Programmierung

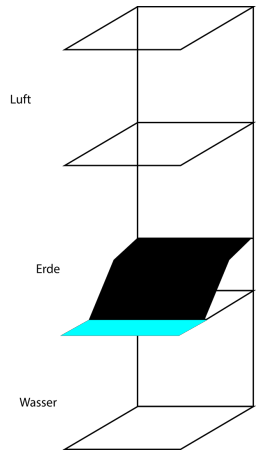
Hans Ferchland & Ahmed Arous

Überblick

Das Scripting des Spiels entstand in kompletter Eigenarbeit auf Basis der Unity Engine. Als Plugins wurden “Unity Serializer” und “Shader Forge” installiert, deren Funktionsumfang haben wir aber relativ wenig erschöpft.
In der Projektzeit entstanden eine komplexe Input-Behandlung, dynamische Spiel- und Menüstrukturen, seed-basierte Levelgenerierung, viele verschiedene Levelelemente, Effekte, drei spielbare individuelle Charaktere, Highscores und Satistiken, Sound und Sound-Events die iterativ und kontinuierlich integriert und getestet wurden und den Modellen und Grafiken Leben einhauchten.

Grobkonzept

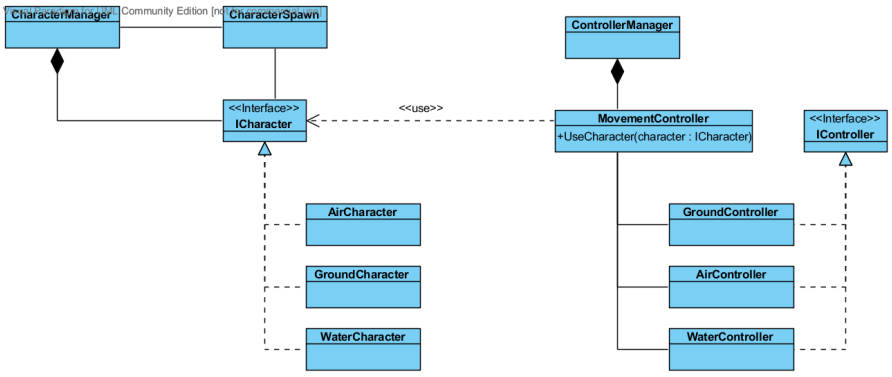
Durch das Gamedesign Dokument (kurz: GDD, siehe Register) was wir zusammen erstellt haben (→ siehe Anhang), hatten wir zu Anfang bereits eine sehr genaue Vorstellung was wir an Kernkomponenten für unser Spiel brauchten.
Wesentliche Komponenten werden nachfolgend Manager genannt. Für die verschiedenen Aufgaben die bei der Umsetzung implementiert werden gibt es einen solchen Manager, der die Übersicht über beteiligte Subkomponenten und die Koordination derer übernimmt. Manager sind zudem statisch und singuläre Instanzen, dadurch fast zur gesamten Laufzeit von überall aus verfügbar.



Da wir eine komplexere Steuerung anstreben, brauchen wir eine Komponente die sich speziell darum kümmert und auch robust mit Wechseln zwischen unseren verschiedenen Layouts umgehen kann. Da die Charaktere mit der Steuerung wechseln, sind die Steuerung und der aktuelle Spieler-Charakter voneinander abhängig. Ein Spieler hat dazu eine Komponente, die dies so abbildet und Verantwortung für Steuerung und Charakter übernimmt.

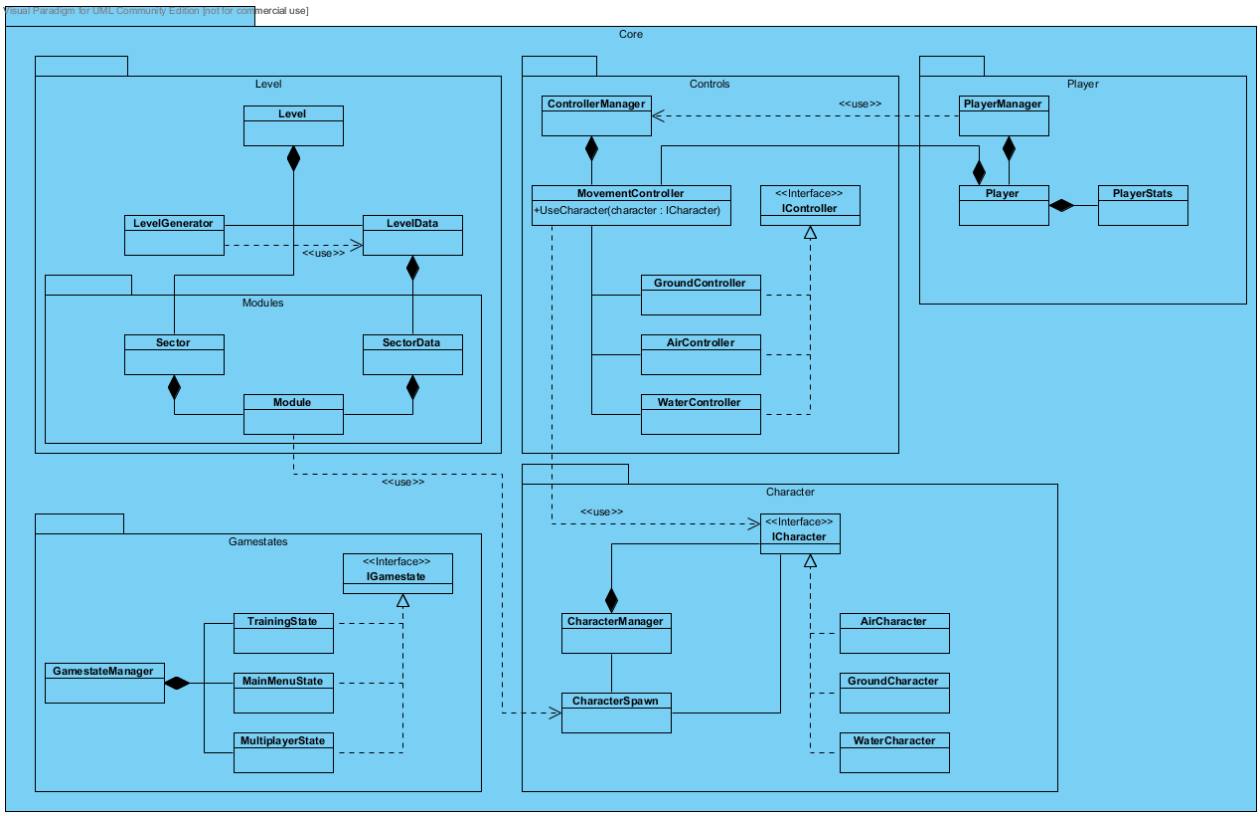
Die Level sollen sich immer anders aus den entstandenden Modulen zusammen setzen, das Konzept dazu war schon weiter fortgeschritten und wurde fast komplett wie im GDD beschrieben implementiert. Hinzu kamen noch die Hindernisse im Level, die in verschiedene Typen aufgeteilt sind. Um die Hindernisse zu beleben wurde ein Trigger+Action System entworfen, das die Levelelemente vielfältig beeinflusst. Der Kern des Spiels, der die Spieler und Spiel-Zustände verwaltet ist relativ simpel als State-Machine für die Zustände und eine Liste aller Spieler entworfen.
Für die GUI entwerfen wir ein einfaches System, das ein einfaches Ingame-HUD für die Spieler und das Menü darstellt. Das Persistente Objekte wie Highscore oder

Einstellungen gespeichert werden müssen ist geplant, aber implementierungs-spezifisch.



Fachkonzept

Auf Basis des GDD und des Grobkonzepts entstanden die Komponenten nacheinander und wurden zum großteil bis zum Ende des Projekts immer weiter entwickelt. Dabei wurden viele Teilkonzepte überarbeitet um den Unity-eigenen Konzepten entgegen zu kommen, Design-Änderungen umzusetzen, Fehlerquellen zu eliminieren oder die Arbeit des Leveldesigners zu erleichtern.
Spezifische Manager und Komponenten sind während der Projektlaufzeit entstanden, dazu gehören Settings-, Highscore-, Camera- und GUI-Manager.



Input Handling

Die Bedienung des Spiels erfolgt mittels Gamepad (siehe Anhang/GDD). Die Unity-Engine bringt einen eigenen InputManager mit, der die Definition von Input-Actions erlaubt und auch später Konfiguration zulässt. Leider werden nicht alle Gamepads sofort und einheitlich unterstützt, weswegen wir uns während des Projekts auf Microsofts Xbox 360 Pad geeinigt haben.
Die Behandlung des Inputs wird für GUI und Charaktere getrennt, so können wir flexibel die Charakter-Steuerung iterieren.
Die GUI-Steuerung war anfangs komplexer designt, wurde dann wesentlich vereinfacht ohne groß an Funktionalität einzubüßen. Grundlage ist ein eigener

InputManager der speziell für das Spiel die Menü-Navigation und Auswahl für beide Spieler an den GUIManager weiterleitet. Grundidee war es auch die Charaktersteuerung mit Hilfe es eigenen InputManagers zu regeln. Leider waren kleine Verzögerungen, die die Steuerung ungenau werden ließen, der Grund für Optimierung.

Für unser Steuerungskonzept mussten dem Input Handling zusätzliche Funktionalitäten bereitgestellt werden.

Triggertasten werden nativ als Achsen behandelt, da diese ursprünglich für Rennspiele konzipiert wurden und sich Achsen für das Beschleunigen und Bremsen ideal eignen. Für unsere Anwendung mussten wir jedoch diese Achsen wie übliche Tasten behandeln, sodass wir unsere eigenen Input Methoden hinzugefügt haben.

Um Simultane Tasteneingaben zu erfassen wurde auf die Geschwindigkeit des Unity Input Managers zurückgegriffen, jedoch mussten Zeitfenster definiert werden, in denen relevante Tastenkombinationen mit einer bestimmten Reaktionstoleranz erkannt werden.

Es war ebenfalls notwendig bestimmte Eingaben mittels eigens definierter Deadzones, also Zonen in denen keine Eingaben erfasst werden, zu ignorieren.

Das eindeutige Zuweisen von Eingaben zu Spielern, durch die Konkatenation einer eindeutigen ID in allen Methoden.

Gamestates & Spieler

Die Kernlogik des Spiels ist von diesen beiden elementaren Komponenten bedingt. Die Gamestates steuern alle anderen Komponenten und laden die Szenen für ihren Bereich. Es gibt MainMenu-, Multiplayer- und Training-State; dazugehörig werden die aktiven Spieler und Game-Controller bestimmt (Trainig → ein Spieler, Multiplayer → zwei Spieler) und die entsprechende GUI geladen oder entladen. Die abstrakte Gamestate-Klasse hat Methoden (`Enter` und `Leave`) die beim Wechsel zwischen verschiedenen States die vorgesehenen Bedingungen herstellen oder zurücksetzen. Die speziellen States erben vom abstrakten Gamestate und implementieren ihr individuelles Verhalten. Der Wechsel und die Erstellung werden vom GamestateManager übernommen. Da dieser auch ein Singleton ist (wie alle Manager, siehe Grobkonzept) kann der besagte Wechsel von außen durch einen entsprechenden Methodenaufruf forciert werden, etwa:

```
GamestateManager.Get().ChangeState(GamestateType.Multiplayer);
```

Um eine falsche Nutzung zu verhindern nimmt diese Funktion nur im GamestateType-enum definierte Werte an!
Spieler werden vom PlayerManger erstellt und verwaltet. Abhängig von der Anzahl der Gamepads, werden ein oder zwei Spieler erstellt. Die Spieler-Klasse hält den Namen und einen eindeutigen Player-Index und die seine Spiel-Statistik. Jeder Spieler hat eine Referenz auf seinen aktuellen Controller und Charakter, feuert Events für Gewinnen, Verlieren und das Beenden eines Levels.

Character Controlling

Ein Character Controller ist eine Komponente die es einem Charakter erlaubt sich im Spiel zu bewegen. Unity bietet hierfür von Haus aus zwei Ansätze, jedoch eigneten sich beide nicht für unsere Zwecke.

Character Controlling mit einem Rigidbody

Ein Character Controller der auf einen Rigidbody aufbaut, basiert auf Unitys Physik Engine. Dies bedeutet das man keine präzise Kontrolle über den Charakter besitzt, weil dieser sich allein durch von außen einwirkende Kräfte bewegt. Man müsste mit Physikalischen Materialien arbeiten, um einiges der verlorenen Kontrolle wieder zu erlangen, doch für uns war es diesen Aufwand nicht wert. Denn auch wenn es das realistischere Ergebnis wäre, so würde man die Steuerung, wie man sie beispielsweise aus Klassikern wie Super Mario Bros. oder Sonic kennt, vermissen. In diesen Spielen war es beispielsweise möglich aus einer hohen Geschwindigkeit sofort in Stillstand zu geraten wenn man dies wollte. Bei diesem Ansatz würden wir also eher gegen die Physik Engine arbeiten als mit ihr.

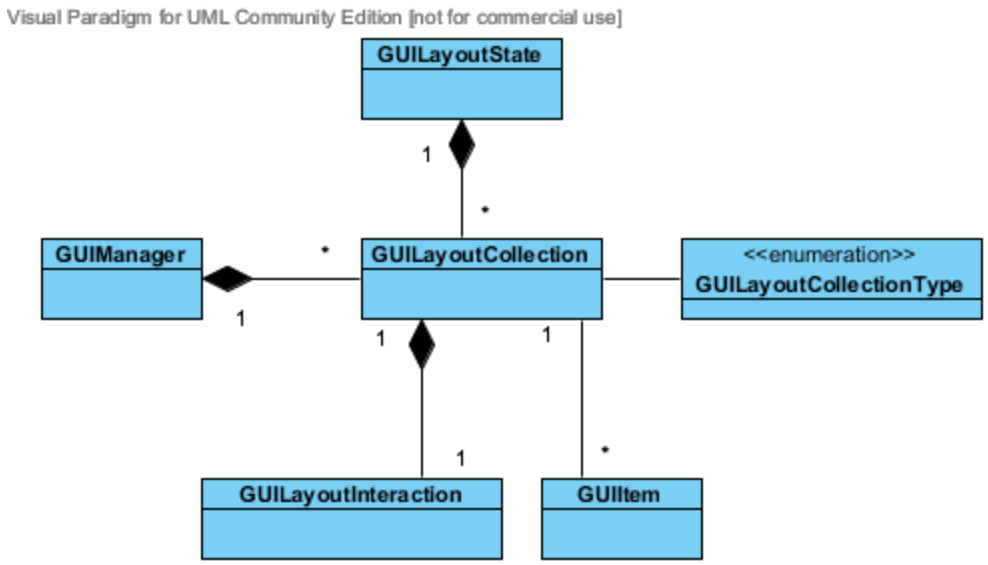
Unitys Character Controller

Unity bietet auch einen eigenen Character Controller, der nicht auf einen Rigidbody aufbaut, allerdings hat dieser einige Einschränkungen wie z.B. dass der eigene Character Controller nur Capsule Collider unterstützt und diese in ihrer Rotation starr sind.
Es war für unsere Spielcharaktere nicht die bestegeeignetste Form, denn selten ließen sich diese sinnvoll in Kapseln unterbringen. Denn in Zukunft könnte es notwendig sein mehr Kontrolle über den Collider zu benötigen, um ihn beispielsweise bei einer Spezialattacke in seiner Größe und Rotation zu manipulieren. Dazu kommt noch ein Kanten Problem, denn Capsule Collider bleiben gerne an rechtwinkligen Kanten stecken.

Nachdem beide Ansätze ausprobiert wurden, entschlossen wir uns das Problem auf klassische Weise anzugehen und einen eigenen Charakter Controller zu erstellen. Dabei war es also nötig Methoden zu finden, um den Spielerinput in tatsächliche Bewegung umzuwandeln. Es war außerdem wichtig Methoden für das Kollisionsverhalten zu bestimmen und Methoden einzubinden die einen Äußeren Kräfteeinfluss auf die Charaktere erlaubt.
Hier hätte uns eine längere Planungsphase oder mehr Ehrfahrung viel Zeit einsparen können, denn man hätte direkt mit einem eingenen Controller starten können und somit auch die benötigten Feature für den Character Controller frühzeitig definiert. Jedoch wurde unser Controller anfangs simpel konzipiert und musste gezwungener Maßen immer wieder angepasst und erweitert werden.
So war es Anfangs nicht möglich Schrägen zu bewältigen oder sich dem Boden entsprechend zu rotieren.
Im Verlauf dieser Anpassungen wurden auch alle Krafteinwirkungen wie z.B. Gravitation, oder die Windboxen, als auch alles weitere was ansatzweise mit der Bewegung zu tun hat im Character Controller vereint, um einen sauberen Updatezyklus einzuhalten und Jittering möglichst zu vermeiden.

GUI und HUD

Der GUIManager verwaltet die durch Gamestates und ihre geladenen Level erzeugten GUI-Komponenten.



Dabei sammelt der GUILayoutState alle GUILayoutCollections eines Gamestates. Die GUILayoutCollection selber muss im Unity-Editor mit den zugehörigen GUIItems befüllt werden und zum GUILayoutState der Szene hinzugefügt werden.
Der GUILayoutState trennt zwischen der GUI für beide Spieler, jede GUILayoutCollection hat zudem einen PlayerIndex, der die GUI einem Spieler zuordnet. Die Funktionalität wird in den GUIItems definiert, dazu existieren viele Elemente wie Checkboxen, Buttons und Listen, welche von Oben nach unten geleitet wird - vom GUIManager über die GUILayoutCollection zu dem momentan selektierten GUIItem der GUILayoutInteraction.
Wechselt der Gamestate, wird die aktuelle GUI für beide Spieler entladen und die GUI aus den folgendem Gamestate geladen. Während eines Gamestates wechselt das Spiel auch zwischen verschiedenen GUILayoutCollections, etwa im Hauptmenü zwischen Optionen- und Multiplayer-Menüpunkt, auch unabhängig vom Spieler.
Das Ingame HUD visualisiert Daten von Charakteren und Spielern, die GUIItems für Lebensbalken, Punkte, Zeit und Fortschritt holen sich ihre Daten über den PlayerIndex.

Alle Elemente der MainMenu GUILayoutCollection

Kamera & Effekte

Die Kameras der Spieler sind durch den CameraManager überall zugängig, wie auch die CameraController, welche die Kamera an einen Charakter binden und auch Post-Effekte verwalten und aktivieren/deaktivieren.
Eindeutig identifiziert wird die Kamera bzw. der CameraController eines Spielers durch den PlayerIndex.
Post-Effekte werden durch die CameraConfiguration gewechselt, in den Einstellungen deaktivierte Effekte werden dabei beachtet und ggf. nicht aktiviert.

Sektoren, Module & Levelgenerierung

Startet man ein Training oder Multiplayer Spiel wird zunächst ein Level generiert. Dies wird durch den LevelGenerator mit den in GeneratorSectorBowl festgelegten Sektoren und einem zufälligem Level-Seed generiert. Dabei wählt der LevelGenerator zunächst die genutzten Sektoren, vom Seed abhängig, zufällig aus und diese wiederum wählen die Module für den finalen Sektor ebenso zufällig. Dabei wird zwischen Level und LevelData, Sector und SectorData unterschieden. Die Data-Klassen halten die für die Laufzeit generierten Module und Sektoren, die Anderen werden nur für das Erstellen von Sektoren und Modulen im Editor genutzt.

Levelelemente

Folgende basieren alle auf einer abstrakten Element-Klasse:

- **ConnectorElement**: verbindet zwei Module miteinander, wird bei der Levelgenerierung gebraucht um die Module in der richtigen Relation zueinander zu platzieren.
- **Action & Trigger**: Aktionen werden durch Trigger aktiviert. Trigger können verschieden ausgelöst werden, etwa durch das Eintreten in ein Trigger-Volumen oder durch Schaden, einen Raycast oder eine Partikel-Kollision. Ausgelöste Aktionen können Schaden machen, Sound abspielen, Partikel feuern oder andere Aktionen oder Trigger aktivieren.
- **Path**: bestimmt wie durch eine Ebene (Luft, Boden, Wasser) navigiert wird. Die Klasse ControlPointElement definiert die Punkte in der 3D-Welt. Außerdem werden entlang dieses Pfades ScoreElemente platziert, falls dies im Unity-Editor so eingestellt wurde.
- **ScoreElement**: kann vom Charakter eingesammelt werden, Punkte und Partikel-Effekt können im Unity-Editor eingestellt werden.
- **Spawn/Despawn**: spezielle Levelelemente die dem Spawn-Modul bzw. dem Fight-Modul im Unity-Editor bekannt gemacht werden müssen. Spawn-Punkte werden zum Spawnen eines Spieler Charakters genutzt, falls dieser den Sektor betritt oder stirbt. Despawns sind Regionen, die beim betreten durch einen Charakter den entsprechenden Spieler in den nächsten Sektor teleportiert oder das Level beenden.
- **Zones**: definieren die Ebenen als Volumen, fügen optional Schaden zu für Charaktere die nicht in die Ebene gehören (etwa Boden im Wasser).
- **ParticleEffect**: definieren Effekte die durch Actions benutzt werden können. Werden hauptsächlich zu eindeutigen Identifikation gebraucht.

Dynamische Levelelemente

Spezielle Levelelemente können sich bewegen, rotieren, verschwinden und mehr. Dazu gehören etwa Minen, Laser oder Plattformen.
Sie werden durch eine komplexe Kette an Triggern und Aktionen definiert, die nur einmal abgespielt wird und falls ein Spieler stirbt, diese nicht erneut passieren muss. Die Logik hinter den Elementen wird komplett im Unity Editor erstellt und ist nur durch die definierten Trigger und Actions beschränkt.

Sound

Sound war zwar von Beginn an eingeplant und konzipiert jedoch war es erst sehr spät im Projekt möglich sich damit zu befassen. Funktionale Ansätze sind vorhanden, aber wurden noch nicht vollständig in das Spiel eingebunden

Splitscreen Problematik

Üblicherweise ist das Setup in einer Szene ein Audiolistener an einer Kamera, der auf alle vorhanden Geräuschquellen reagiert.
In unserem Fall besitzen wir im Mehrspieler Modus 2 Kameras und eine Unity Szene unterstützt nur einen Audiolistener.
Dies ist besonders problematisch weil sich die Spieler an komplett verschiedenen Standorten befinden können, man aber dennoch eine gewisse Immersion beim Spieler erreichen möchte.
Bei der Recherche zu diesem Problem hat sich ein üblicher und simpler Lösungansatz ergeben.
In jeder Szene befindet sich ein Audiolistener im Koordinatenursprung. Alle Audioquellen werden in jedem Frame auf ihre Distanz zu beiden Spielern überprüft und relativ zur kürzeren Distanz zum Audiolistener positioniert.
Dadurch ist zumindenst sicher gestellt das jeder Spieler die Audioinformationen in voller Lautstärke erhält, die einen selbst betreffen und kann diese seinem Kontext entsprechend einordnen.
Damit die richtige Positionierung der Audioquellen sicher gestellt wird, werden nicht nur die Positiondifferenzen des eigentlichen Objekts betrachtet, sondern auch die der Parentobjekte, so können auch sich bewegende Audioquellen korrekt platziert werden.

Audiomanager

Der Audiomanager ist dafür zuständig die Hintergrundmusik abzuspielen und Szenenübergreifend zu Steuern.
Er besitzt statische Platzhalter für bestimmte Lieder wie z.B. die Titelmusik und eine Playlist mit mehreren Titeln die zufällig abgespielt werden können.
Innerhalb einer Szene kann über einen Trigger die Musik geändert werden.
Der Audiomanager nutzt zwei Audioquellen um zwischen zwei Tracks Crossfaden zu können.
Momentan ist dieser allerdings noch nicht vollständig eingebaut, außerdem sorgen Performance Probleme während des Szenenaufbaus sorgen für unsaubere Musikübergänge. Hier müsste nochmal angesetzt werden und man könnte bei der Gelegenheit seine Funktionalität um Ambiance Sounds erweitern.

Charakterdesign - Gorcrasp, Beecrocha, Rhishoktopus, Bideezeel

Christoph Duda

Allgemein

Das Erschaffen eines Hauptcharakters ist meiner Meinung nach eine der wichtigsten Aufgaben in der Spieleentwicklung. Oft entscheidet der Hauptcharakter darüber, ob der Spieler das Endprodukt gut oder schlecht findet, denn der Spieler muss sich mit dieser Figur identifizieren können, sie interessant finden, und durch diese Figur in die virtuelle Welt eintauchen. Findet man den Hauptcharakter uninteressant oder sogar schlecht, ist die Story um diese Figur auch nur Nebensache.
In unserem Spiel gibt es so etwas wie "den einen Hauptcharakter" nicht. Es gibt die Hauptfigur des verrückten Professors, der verantwortlich für die Existenz der Kreaturen und der Testsektoren ist, weil er seine Kreaturen in diesen auf ihre Überlebenskünste testet, um die perfekte Kreatur zu erschaffen. Allerdings spielt der Spieler nicht diesen Professor, sondern eben dessen geschaffene Kreaturen.
Zusätzlich wechselt der Spieler von Sektor zu Sektor die Kreatur und muss sich dadurch in regelmäßigen Abständen auf komplett neue Gegebenheiten anpassen. Nicht nur die Kreatur und dessen Fähigkeiten und Umgebung ändert sich, sondern auch die Steuerung. Dadurch wird genau genommen der Spieler zum "Testsubjekt", der seine Überlebenskünste gegenüber dem Gegenspieler beweisen muss.
In diesem Fall mussten also mehrere Charaktere geschaffen werden, die den Anspruch als "Hauptcharakter" erfüllen, denn der Spieler sollte sich möglichst bei jedem Wechsel der Kreatur auf den folgenden Charakter freuen. Es sollte möglichst vermieden werden, dass der Spieler einen Gedanken wie "Och nö, jetzt muss ich das Vieh spielen..." formt. Die Charaktere sind zudem nicht menschlich, sondern Mischwesen, welche die Vorzüge von realen Tieren in einer Kreatur vereinen.
Es galt also als erstes diese Vorzüge zu finden und aus diesen Einzelteilen viele Kombinationen zusammenzustellen um sich dann für ein Mischwesen zu entscheiden. Die Recherche erfolgte überwiegend im Internet, aber auch die ein oder andere Tierdokumentation im Fernsehen hat zur Ideenfindung beigetragen.

Die erste Idee für einen Charakter war die Bodenkreatur "Gorcrasp", ein Gorilla mit dem Hinterleib einer Spinne, acht Spinnenbeinen, den Punktaugen einer Spinne und den Scheren einer Krabbe.
Der Name stellt sich aus "GORilla", "CRAb" und "SPider" zusammen und sollte vorerst während der Entwicklungszeit verwendet werden.

Dieser Charakter war mein erster Versuch einen ausgefallenen, interessanten Charakter zu erschaffen. Die Mischung der Tiere sollte sich nicht auf eine bestimmte Art von Tieren beschränken. Gerade die Vereinigung von Elementen, welche auf dem ersten Blick zu unterschiedlich wirken, steigert den Wiedererkennungswert und macht einen Charakter erfrischend anders. In diesem Fall ist es die Zusammenführung von Gliederfüßern, hier Kieferklauenträger und einem Krebstier, und einem Primaten. Die Behaarung einer Vogelspinne nimmt den Platz des Gorillafells ein und wird an den Armen durch die glatte Panzerung der Scheren ersetzt, was den Charakter optisch etwas auflockert. Zu seinen Attacken werden im Nahkampf die Scheren eingesetzt und im Fernkampf kann ein Spinnennetz abgefeuert werden, welches die Gegner für einige Sekunden am Boden festklebt. Jeder Charakter kann auch eine Spezialattacke, dieser kann seine Kraft durch lautes Gebrüll und Brustklopfen demonstrieren, was die Gegner in Angst versetzt und sie für einige Sekunden panisch flüchten.

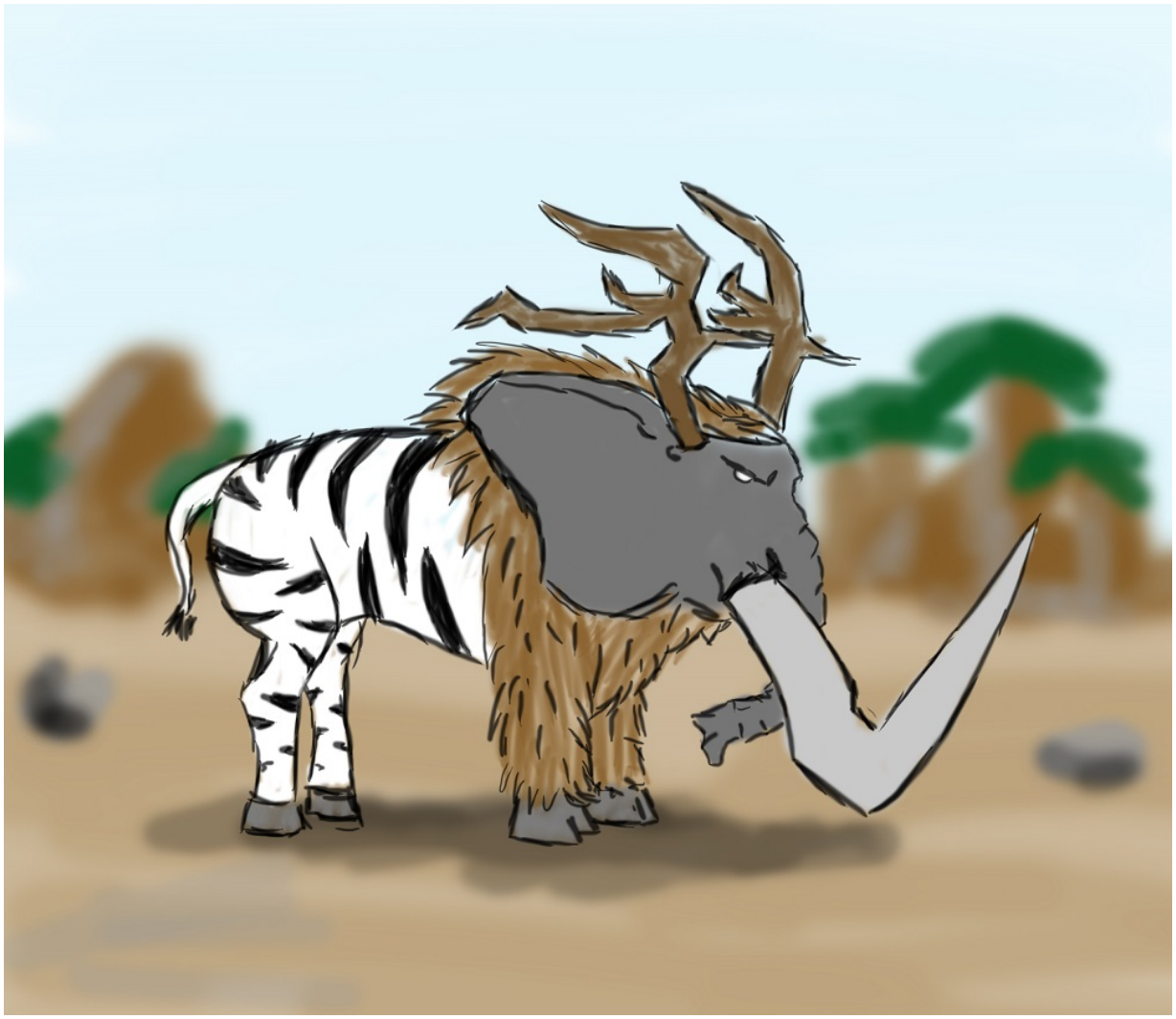
Für die erste Version des Spiels wollten wir für jede spielbare Ebene (Luft, Boden, Wasser) zumindest einen Spielbaren Charakter. Die Bodenkreatur wurde vom Team abgesegnet und nun mussten noch Luft und Wasser besetzt werden.
So entstand die zweite Idee: "Beecrocha" (BEE, CROCodile, HAwk), ein Krokodil mit den Schwingen eines Adlers und dem Unterleib einer Biene, da die Luftwesen nicht nur im Nahkampf sondern auch Fernkampftüchtig sein sollen.

Auch hier war die Idee, die Komponenten so unterschiedlich auszuwählen, dass es im ersten Moment verrückt erscheint, dem aber trotzdem eine Glaubwürdigkeit durch die Proportionen und Verschmelzung zu verleihen. Die Verbindung von Greifvogel, Insekt und Reptil macht den Charakter befremdlich, aber einzigartig in seinem Erscheinungsbild. Der Krokodilpanzer vermischt sich mit der gewölbten Form des Hinterleibs der Biene und nimmt dabei die schwarz-gelben Streifen auf. Die

Spannweite der Adlerflügel sollte nicht unrealistisch wirken aber auch nicht über proportioniert.
Der Bienenstachel stellt ein Gleichgewicht zwischen dem Aggressiven Kopf und dem sonst eintönigen Hinterleib dar und erfüllt gleichzeitig die Visualisierung der Nah- und Fernkampfattaken. Der Krokodilbiss ist die offensichtliche Nahkampfattacke, der Stachel lässt sich für den Fernkampf abfeuern, wobei die Kreatur dadurch nicht stirbt, was man von einer Biene erwarten würde, sondern in schnellem Tempo gleich neue Stachel hervorbringt. Als Spezialattacke wird ein starker Flügelschlag genutzt, um anrückende Gegner wieder zurückzustoßen.
Diese beiden Charaktere wurden für die erste Version auch umgesetzt, die beiden nächsten haben es aus Zeitgründen nicht geschafft.
Die dritte Kreatur, die es in die Auswahl geschafft hat, ist der "Rhishoctopus" (RHInoceros, SHark, OCTOPUS). Ein mit Stacheln des Kugelfisches besetzter Hai mit dem Nasenhorn des Nashorns und den Tentakel des Oktopus.

Dieser Charakter sollte im Wasser eingesetzt werden, sein Nasenhorn ist für den Nahkampf bestimmt, die Tentakel lassen ihn dabei schnell nach vorne schießen. Zur Verteidigung kann er sich zum Kugelfisch aufblasen, dabei entsteht jedoch ein Auftrieb und er lässt sich kaum noch steuern. Als Fernattacke kann er einen Tintenwolke ausscheiden, welche seine Gegner verwirrt.
Die vierte Idee entstand durch den Gedanken, dass noch ein Vierbeiner fehlt, eine starke Kreatur, die vielleicht die Fähigkeit hat durch Bäume und Wände zu stampfen. So entstand "Bideezeel" (BIson, DEEr, ZEbra, ELephant), dessen Stärke und

Fähigkeiten sich komplett auf den vorderen Teil des Körpers konzentrieren, dem Elefantenkopf mit dem Hirschgeweih am Bisontorso.



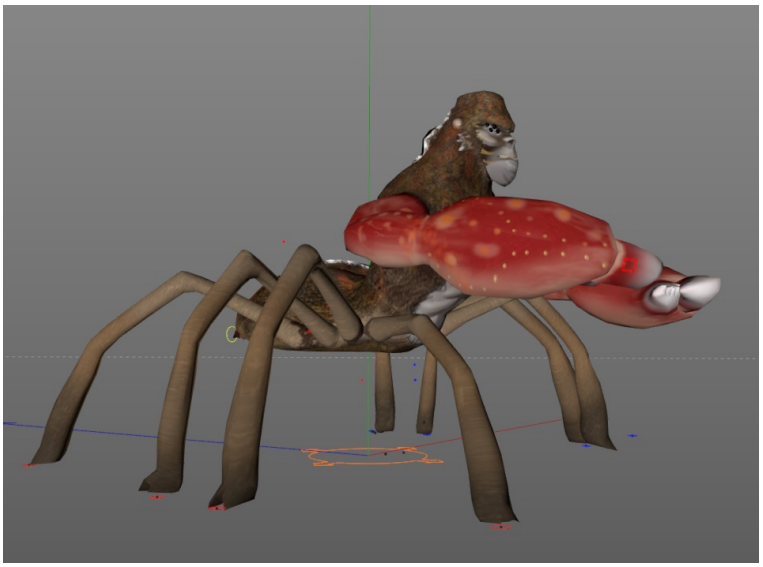
Seine Schwachstelle ist der Hinterleib des Zebras. Seine Stoßzähne und das Geweih werden für den Nahkampf eingesetzt, mit dem Rüssel kann er vom Boden aufgesaugte Steine, Wasser oder Matsch für den Fernkampf verschießen. Seine Spezialattacke ist ein mächtiger Stampfer, der die Gegner im Umkreis betäubt.

Jeder Charakter sollte seine eigenen Vor- und Nachteile erhalten, in den Geschicklichkeitspassagen, sowie im Kampfabschnitt und sich unterschiedlich spielen, damit auch hier der Spieler immer wieder neu gefordert wird. Um das zu erreichen muss jedoch auch das Leveldesign entsprechend angepasst werden, z.B. muss von Anfang an beachtet werden, dass Bideezeel bestimmte Objekte kaputtstampfen kann und durch Hindernisse ab einer bestimmten Geschwindigkeit hindurchrennt, was Gorcrasp allerdings nicht kann. Dafür wiederum kann Gorcrasp höher springen, weil

das seine Spinnenbeine erlauben und das ermöglicht ihm Levelabkürzungen zu erreichen.
Aber zum Leveldesign kommen wir später. In diesem Abschnitt möchte ich meinen Workflow zur Umsetzung der Charaktere beschreiben und darauf eingehen, welche Schwierigkeiten es gab und welche Lösungsansätze verfolgt wurden.
Dieses Projekt ermöglichte mir zum ersten Mal einen kompletten Prozess der Charakterentwicklung zu durchlaufen, von der Idee, über die Skizzen bis zum Texturieren und Animieren für den Einsatz im Spiel. Ich habe zwar schon viel im Bereich "Characterdesign" in meiner Freizeit erarbeitet, bisher musste ich jedoch nicht auf die Polygonmenge achten, oder einen Charakter von Kopf bis Fuß riggen und Sequenzen animieren, die immer wieder zur ursprünglichen Pose zurückfinden.
Ein großes Anliegen war mir auch, dass meine erstellten Charaktere und Objekte "Aus einem Teil" bestehen, also nicht aus einzelnen Armen, Beinen, einem Körper und einem Kopf bestehen, die einfach zusammengesteckt werden. Durch diese neue Herausforderung konnte ich vieles dazulernen, bin aber auch an unerwartete Grenzen gestoßen und musste, z.B. Im Falle von Gorcrasp drei Mal von neuem Anfangen.

Von der Skizze zum Modell

Wenn man sich das 3D-Modell von Gorcrasp im Vergleich zur Skizze anschaut, fallen viele Änderungen auf. Die Farben, die Anzahl der Beine und die gesamte Körperhaltung wurde während der Umsetzung angepasst und verändert. Dabei wurden die in der Teambesprechung vorgeschlagenen Änderungen übernommen und im Prozess frei designte Alternativen übernommen, wenn diese ein besseres Ergebnis lieferten.



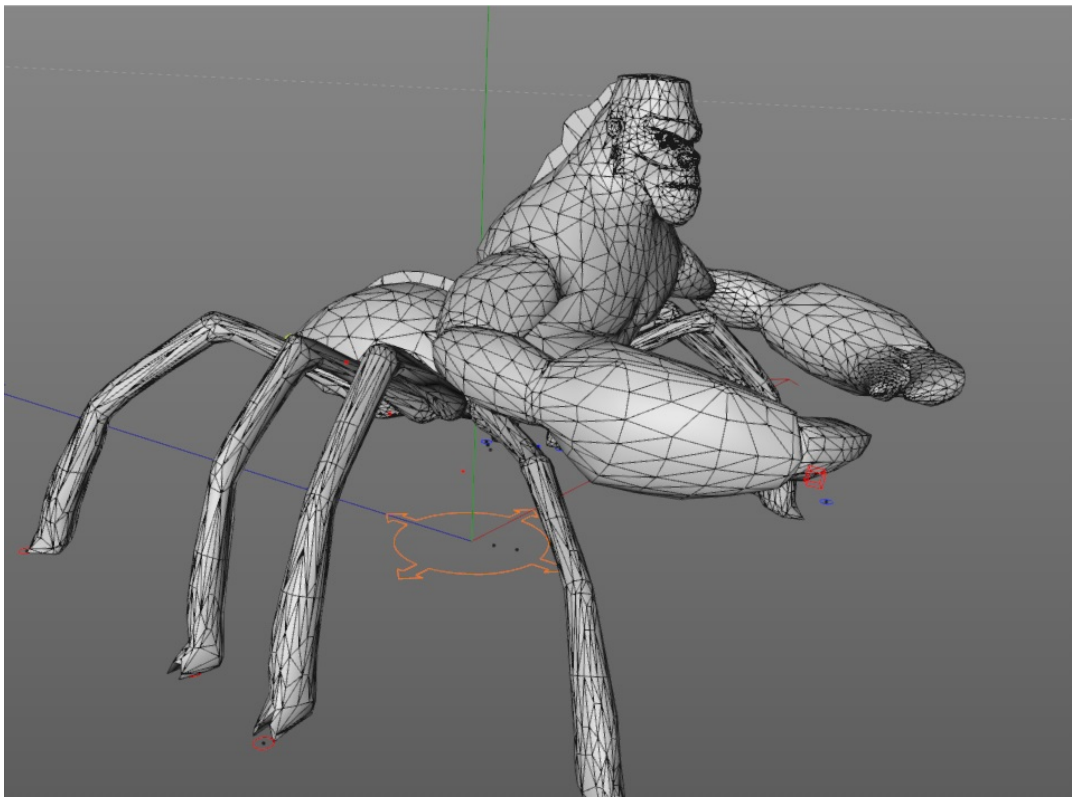
Die Umsetzung teilt sich in mehrere Schritte und beinhaltet das erstellen des 3D-Mesh (Polygonmodell), das Rigging (siehe Register) und Skinning als Grundlage für die

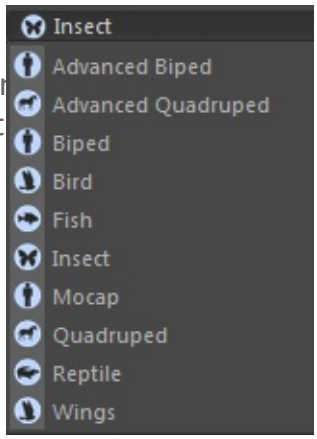
darauf folgende Animation und das Texturieren des Modells in drei Ebenen (Colormap, Normalmap, Alphamap).
In meinem Fall wurden alle Schritte mit der Animationssoftware "Cinema 4D" umgesetzt und mein Workflow beschreibt das Vorgehen mit den entsprechenden Tools, wie zum Beispiel "Bodypaint 3D" zum Texturieren direkt auf dem Mesh. Ergänzend wurde "Photoshop" für die Überarbeitung und Anpassung kritischer Stellen bei den Texturen eingesetzt.

Das Mesh

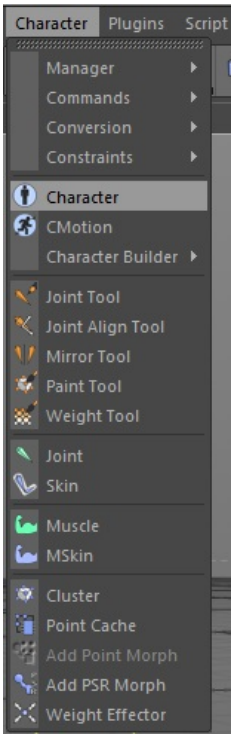
Christoph Duda

Bei der Erstellung von Charaktermodellen beginne ich gerne mit dem Kopf. Das Gesicht bildet die Grundstimmung des Charakters und gibt eine Richtung für den Rest des Körpers vor. Dank der "Sculpting"-Funktion von Cinema 4D ist das erschaffen organischer Formen aus einem Grundmodell (meist eine Kugel) viel schneller möglich als durch das nutzen der Standard-Polygontools oder gar dem Erstellen eines Modells "aus dem Nichts", also durch händisches Erstellen von Punkten und Verbinden dieser zu einem Polygonmodell.
Beim Sculpting können recht schnell sehr detaillierte Modelle erstellt werden. Nicht selten können dabei Polygonmengen im Millionenbereich entstehen. In unserem Fall jedoch, brauchten wir möglichst wenig Polygone, das bedeutet, es musste an unwichtigen Stellen stark reduziert werden, an wichtigen Stellen sollte man so weit reduzieren, dass die Qualität nicht zu sehr leidet. Besonders das Gesicht hat viel Zeit in Anspruch genommen, denn oft mussten Polygongruppen gelöscht und neu angeordnet werden, weil man immer wieder stellen entdeckte, wo noch reduziert werden konnte. Hilfreich ist in diesem Fall das "Symmetrie-Tool", welches das Modell spiegelt und man so nur an einer Hälfte arbeiten muss, da die andere Hälfte alle Änderungen gespiegelt übernimmt und man dadurch eben ein "symmetrisches" Modell bekommt. Das war besonders hilfreich als es um die acht Spinnenbeine ging.



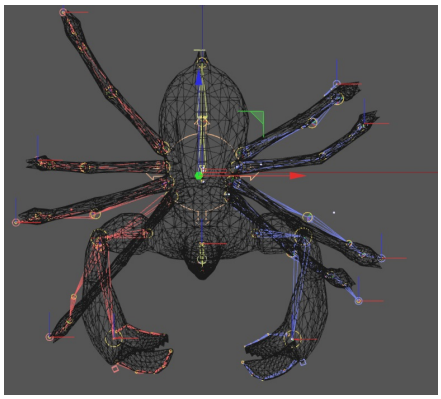


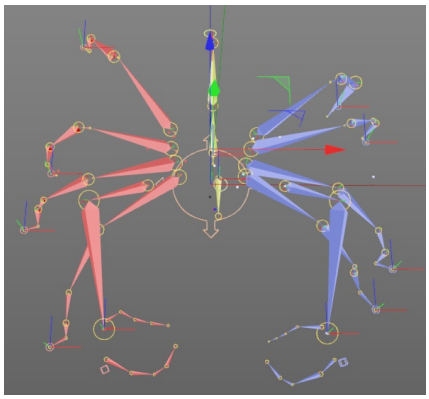
Bevor ich das Mesh texturiere, erstelle ich gerne das "Rig" und die Animationen. Untexturiert lassen sich beim Animieren Fehler und Überlappunggen einfacher erkennen und manchmal kommt man beim Animieren auf weitere Ideen, welche das Aussehen der Figur betreffen und man kann diese in die Textur einbeziehen. Das Riggen wurde in den letzten Versionen von Cinema 4D ebenfalls vereinfacht und kann unter anderem durch das "Character-Tool" realisiert werden. Hier wird eine Hierarchie von "Bones" nach bestimmten Vorlagen erstellt. "Bones" beschreiben, wie der Name bereits sagt, die "Knochen" des Modells. Vom Zweibeiner, über Vögel bis Insekten kann man mit diesem Tool ein individuelles Rig für jede erdenkliche Figur erstellen.



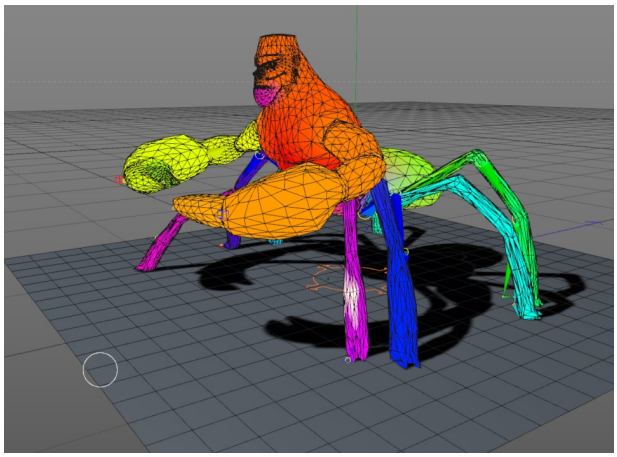
Man beginnt mit der Wirbelsäule samt Kopf und hängt beliebig viele Arme und Beine daran. Darauf folgen Hände und Füße, wahlweise auch Finger und Zehen. Optional ist auch das Riggen von Augen und dem Kiefer. Bei bestimmten Tierarten sind anstelle der Hände Zangen vorhanden oder anstelle von Armen eben Flügel. Das Problem hierbei bestand nun darin, dass unsere Kreaturen Mischwesen sind und somit keines der Templates perfekt passt. Dennoch habe ich mich für diese Methode entschieden, weil das Character-Rig im Gegensatz zum selbst erstellten Rig viel schneller erstellt und gleichzeitig mit hilfreichen Kontrollern ausgestattet ist und über gekoppelte "Inverse Kinematics" die Animation vereinfacht. Im Falle von Gorcrasp wurde das "Insect"-Template verwendet, da hier mehrere Beine, zwei Zangen, ein Kopf und der Spinnenkörper als abstrakter Schwanz zum Einsatz kommen. Sobald das Rig auf die Figur angepasst wurde, ging es darum den "Bones" zu sagen, welchen Teil des Körpers sie nun bewegen sollen. Dies geschieht im so genannten "Skinning". Dabei wird das Mesh mit Wichtungen "bemalt".

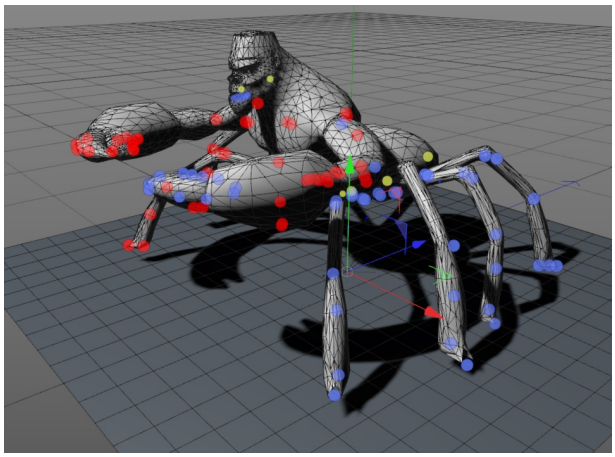
Man wählt einen der Bones an und färbt die Bereiche des Models ein, welche sich mit diesem Bone bewegen sollen. Dabei bestimmt die Deckkraft der Farbe die Stärke des Einflusses auf das Mesh. Dadurch werden sanfte Übergänge möglich und die einzelnen Komponenten bewegen sich fließender.





Der Vorgang wird mit jedem Bone durchgeführt und zum Schluss werden alle Bones ausgewählt um fehlerhafte Stellen zu beseitigen oder die gesamte Gewichtung zu glätten. Bei dieser Arbeit fallen oft nochmals Fehler im Mesh auf, zum Beispiel erkennt man, dass Bewegungen an Gelenken nicht funktionieren, weil das Mesh an dieser Stelle nicht richtig unterteilt ist oder einfach zu wenig Polygone für eine Bewegung besitzt. Hätte man jetzt bereits eine Textur erstellt müsste man beim Optimieren des Modells ebenfalls die UV-Koordinaten für die Textur anpassen.





Nachdem man diese Fleißarbeit hinter sich gebracht hat, beginnt der spaßige Teil der Charaktererstellung:

Animationen

Christoph Duda

Das Tolle am Animieren ist, dass man eine Idee, die erst ganz Abstrakt im Kopf entsteht, schließlich zum "Leben" erweckt. Eine grobe Skizze nimmt durch das Modellieren Gestalt an, doch erst die Animation kann aus einer Statue einen tiefgründigen Charakter machen. Die Bewegungen, Reaktionen und Gesichtsausdrücke beeinflussen unseren Eindruck über die Figur.
Da wir es in unserem Spiel mit Kreaturen zu tun haben, die ihren Ursprung in der Tierwelt finden, sollten auch zumindest die Laufanimationen aus der Tierwelt stammen. Dafür ist Richard Williams' "The Animator's Survival Kit" eine große Hilfe gewesen und kann allgemein für das Thema Animation empfohlen werden. Die 2D Animation ist der Grundstein der 3D Animation und sollte meiner Meinung nach immer noch als Grundlage beachtet werden.
Wie bereits erwähnt mussten die Animationen für unser Spiel immer wieder in ihrer Ursprünglichen Position enden, damit jede Animation von der "Idle" Animation abgelöst werden kann. Die "Idle"-Bewegung zeigt das Verhalten der Kreatur, wenn sie Still steht und der Spieler keine Eingaben tätigt. Zusätzlich zu dieser Idle-Animation wurden noch die Lauf-/ Flugbewegung, das Springen, Sterben und Schaden erleiden,

die Nah- und Fernattacke, die Spezialattake und "Langeweile" animiert. Die "Langeweile"-Animation kommt zum Einsatz, wenn der Charakter eine längere Zeit nicht bewegt wird. Eine Herausforderung waren die "walk cycles", also die Animation der Fortbewegung und die Sterbeanimationen. Das Laufen einer Spinne und das Fliegen habe ich in diesem Projekt zum ersten Mal animiert und brauchte vor allem beim Flügelschlag mehrere Anläufe. Die Animationen wurden hintereinander durchanimiert, damit später eine einzelne Datei exportiert werden konnte, die alle Bewegungen beinhaltet. Die Animationen müssen anschließend in Unity zerlegt werden, damit einzelne Sequenzen aufgerufen werden und können in einem separaten Editor in ihren Übergängen angepasst werden.
Nachdem nun die Animationen umgesetzt und alle Fehler behoben wurden, konnte ich mich an das Texturieren der Kreatur machen.

Texturierung

Christoph Duda

Zum Erstellen der Texturen nutzte ich Cinema 4Ds "Bodypaint 3D", und "Photoshop". Bodypaint 3D ist ein starkes Tool zum Editieren von UV-Koordinaten und direktem Malen der Texturen auf dem Mesh. Die UV-Map eines Modells ist eine Art aufgeklappte 2D-Version vom 3D-Mesh. Damit die Texturauflösung stimmt und keine besonderen Übergänge zu sehen sind, muss diese UV-Map auf der vorhandenen Texturfläche geschickt angeordnet werden. Erst wenn alle Flächen des Modells von der Textur abgedeckt werden, kann man das Modell bemalen. Dabei gibt es eine Vielzahl von "Texturen". Die "Colormap" bestimmt die Farben der Oberfläche und ist somit die auffälligste aller Texturen. Ergänzt werden kann diese durch eine Bumpmap, Displacementmap oder Normalmap, welche auf verschiedene Arten weitere Tiefeninformationen auf das Modell legen um feinere Strukturen darzustellen, welche sonst nur durch ein hochauflösendes Mesh erreicht werden könnten.
Specularmaps geben die Möglichkeit zu bestimmen, an welchen Stellen das Modell glänzen oder reflektieren darf und Alphamaps können bestimmte stellen ganz unsichtbar machen, wie es zum Beispiel bei Haartexturen der Fall ist. In diesem Fall wird ein Bild auf eine rechteckige Fläche gelegt und alles, was nicht zu sehen sein soll, wird von der Alphamap abgedeckt bzw. transparent gemacht.
Da wir in unserem Spiel in Richtung Comic-Look gehen wollten, mussten jedoch nicht alle möglichen Effekte genutzt werden. In meinen umgesetzten Modellen wurden handgemalte Color-, Alpha- und Displacementmaps eingesetzt.
Der Workflow in Bodypaint 3D ähnelt dem Vorgehen in Photoshop, denn im Grunde funktioniert alles auf die selbe Art. Man kann mit Layern arbeiten, Bereiche selektieren, drehen, skalieren und bewegen und selbst die Tools ähneln stark dem Vorbild "Photoshop". Es werden auch .PSD Dateien gelesen und können als solche gespeichert und in Photoshop geöffnet werden, wobei Bodypaint ebenfalls eine 2D Zeichenfläche neben dem 3D-Modelpainting bietet. Es entsteht also ein fließender Übergang und man kann recht schnell Texturen erstellen.

Rückblick

Christoph Duda

Da Gorcrasp mein erstes Modell dieser Art war, kann ich im Nachhinein sagen, dass er eine Art Übungscharakter war. Es gibt sicherlich noch Verbesserungspotential im Mesh, den Texturen und der Laufanimation. Dies hat sich meiner Meinung nach stark bei der Umsetzung des zweiten Charakters "Beecrocha" verbessert, da ich nun viele Probleme von Beginn an umgehen und dadurch mehr Zeit in die Optimierung stecken konnte.

Unser Zeitplan war ziemlich straff geplant und unsere Programmierer brauchten schnell einen Charakter um die grundlegenden Elemente zu erstellen und zu testen. Das war anfangs eine grobe, untexturierte und nicht animierte Version von Gorcrasp, welche ein bestimmtes Zeitfenster für die Überarbeitung bekam, da ja schließlich geplant war, diesen Vorgang drei Mal hintereinander zu durchlaufen und im Anschluss noch Levelelemente und das Leveldesign anstanden.
Wie eng schließlich der Zeitplan war, stellte sich bald heraus und nach dem die ersten drei Charaktere fertig waren, mussten wir uns um die Levelobjekte und das Leveldesign kümmern.

Charakterdesign - Wozilla, Squrtle-gator, Bathowk, Locatergontula, Gorilla-Bot

Daniel Glebinski

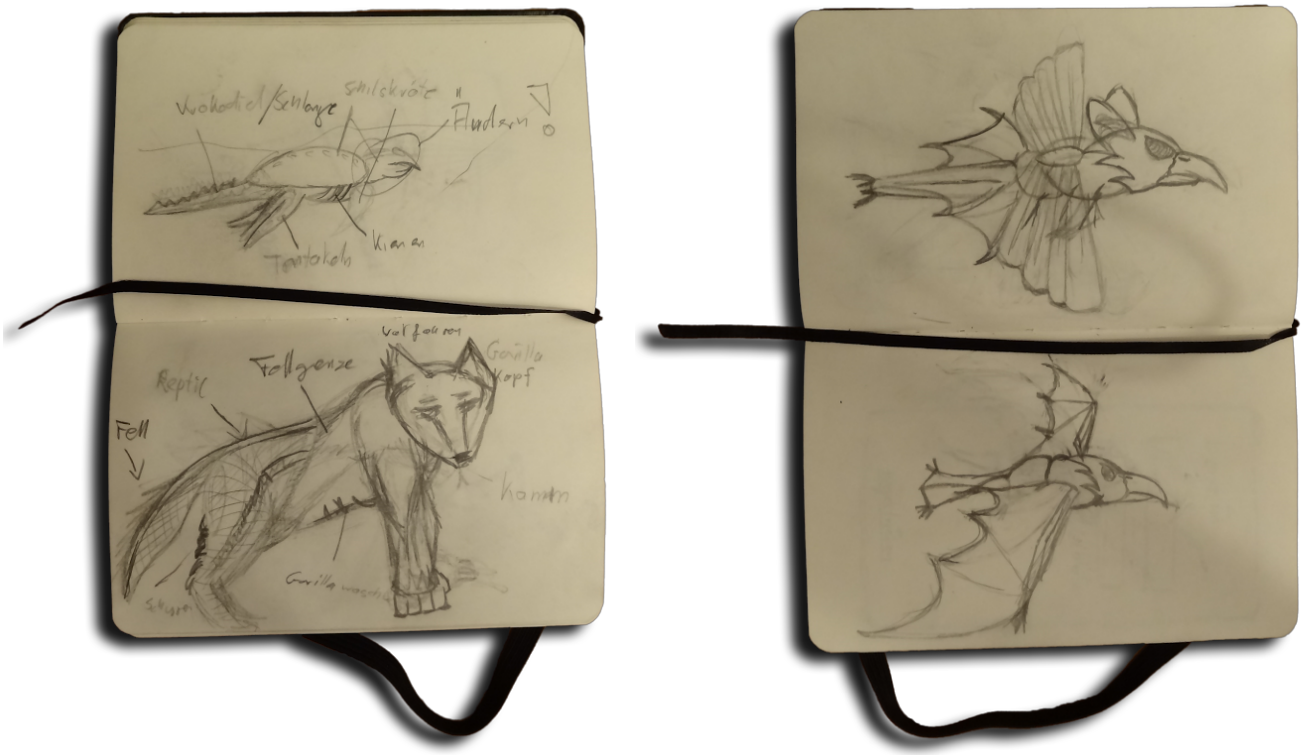
Allgemein

Meine Hauptaufgaben lagen im Bereich der Character-Modellierung und Animation sowie in der Erstellung von visuellen Effekten (Shader für Blätter und Laser, verschiedene Effekte für das Wasserlevel) und Soundeffekte, die aber leider noch keine Anwendung gefunden haben.
Ursprünglich war angedacht, dass drei Mitglieder des Teams, die für Modellierung eingeteilt waren, je einen Charakter für jedes Level erstellen sollten. Später wurde je Level zunächst nur ein Charakter verwendet. Auch sollte in jedem Level ein Gegner zu bekämpfen sein, der ein halbmechanisches Tier - ebenfalls aus dem Labor des Professors - darstellte, allerdings wurden diese auch noch nicht eingefügt.

Im folgenden Bericht werde ich die Erstellung von drei Charakteren beschreiben (von denen einer bisher im fertigen Spiel vorkommt) von der Skizze zum Modell, der Texturierung, dem Rigging und den Animationen verschiedener Bewegungen. Anschließend wird beschrieben, wie einige visuelle Effekte für das Wasserlevel für eine authentische Unterwasseratmosphäre sorgen.
Des Weiteren wurden verschiedene Shader für eine ansprechende Gestaltung der Bäume und des Lasers eines tödlichen Hindernisses benötigt. Die Lösung dieser Probleme mit Hilfe des Unity-Plugins “ShaderForge” soll im letzten Abschnitt des Berichts erläutert werden.
Zuletzt soll noch auf einige weitere Aufgaben eingegangen werden, die zur Fertigstellung des Spiels beigetragen haben, aber keinem eigenen Aufgabenbereich zugeordnet werden können.

Von der Skizze zum Modell

Nach der gemeinsam erarbeiteten Idee von den Figuren, die aus der Kreuzung von je drei verschiedenen Tieren bestehen sollten, habe ich zunächst Bleistiftskizzen erstellt, auf der Basis von Fotos. Für die Landebene wählte ich Wolf, Gorilla und Schlange, als fliegendes Tier habe ich die Kombination von Fledermaus, Hornisse und Falke ausprobiert sowie Heuschrecke, Raupe und Libelle; für die Wasserebene wählte ich Tintenfisch, Schildkröte und Alligatorhecht. Die Ausarbeitungen der Scribbles sind anschließend mit einem Grafiktablett und Art Rage gemacht worden.



Namensgebung

Mir war wichtig, das jeder meiner Kreaturen einen individuellen Namen trägt und aus 3 gänzlich verschiedenen aber der gleichen Spielebene (Wasser, Erde und Luft) angehört. Für die Luft habe ich nur fliegende Wesen benutzt, genauso für Erde nur am Land lebende Tiere und für Wasser nur im Wasser lebende Tiere.
Die Namensgebung ist ein Portmanteau-Wort basierend auf den englischen Wörtern für die Tiere, welche durch Amalgamierung zu einem neuen Namen verschmolzen werden.

Beispiel Wozilla:
Bestandteile:
Gorilla Wolf Snake

Auftrennung in sinnvolle Laute:

S nake	W olf	G oril la
Sn ake	W ol f	Go ril la
Sna ke	Wo lf	Go rill a
	Wol f	Gor [1]illa
		Gor ill a

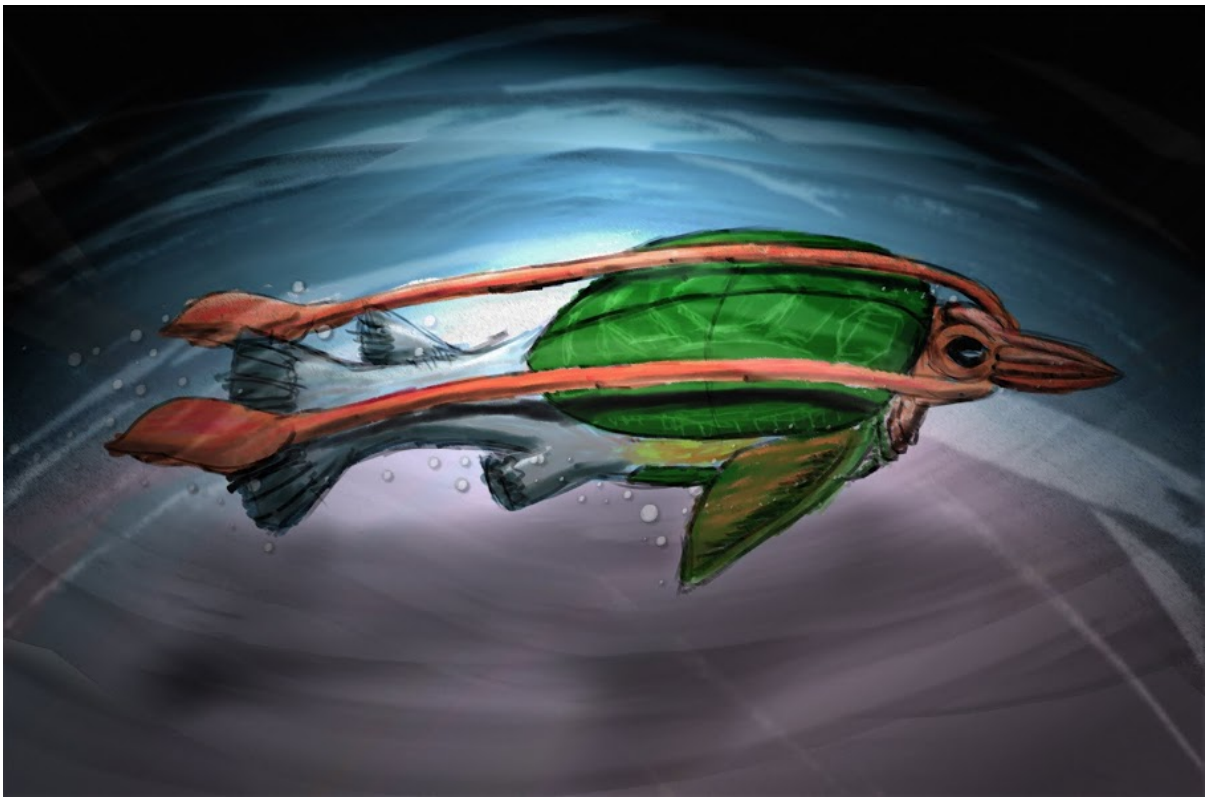
Kombinationen:

Go wo ke	Wol rill ke
Go wo nake	Wo rilla ← Snake fehlt
Wo go ke	Wo sn illa
Sna w rill	Wo s illa
S ril lf	
Wo rill ke	Wo z illa←Favorit: mit z (stimmhaftes s), da w ebenfalls stimmhaft ist

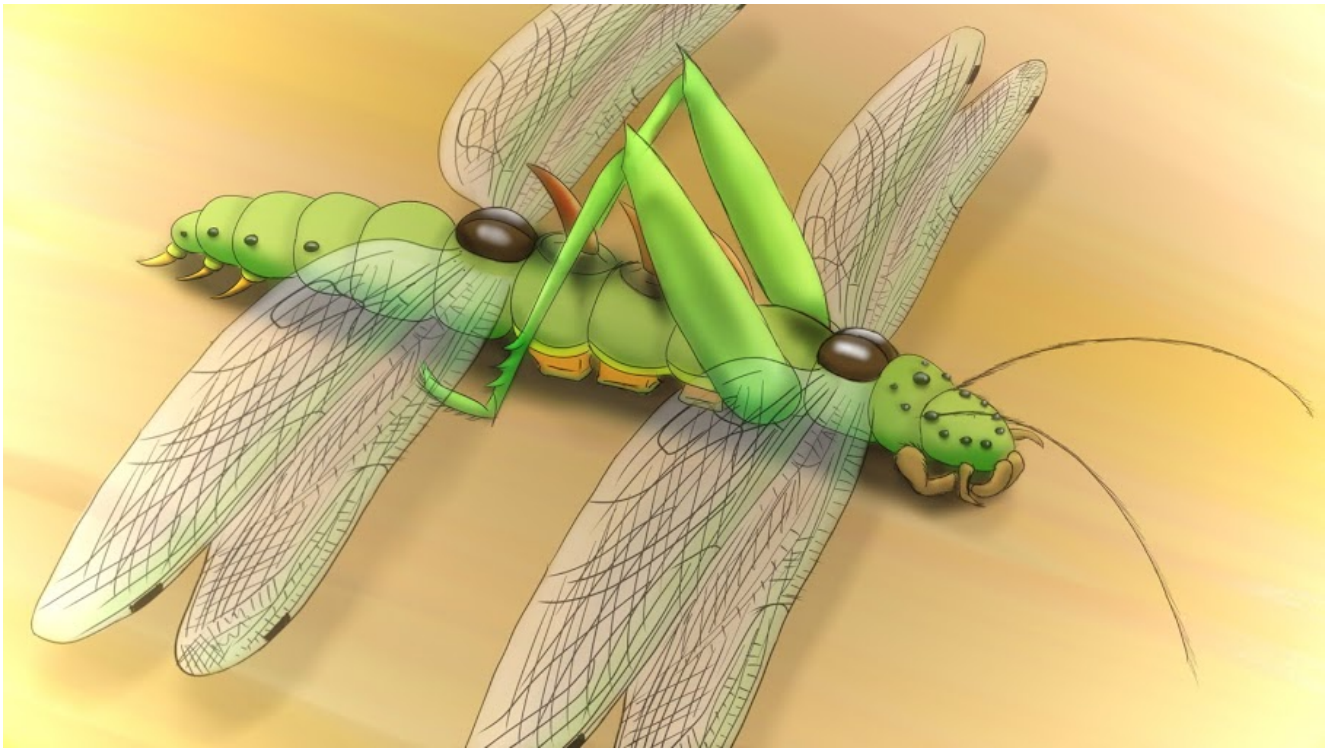
Genauso wurden auch die Namen Squrtlegator (squid, turtle, alligator) und Locatergontula (Locust Caterpillar Dragonflie Tarantula) entwickelt.

[1] Gor kam nicht in Frage da es bereits den Gorcrasp das (Landwesen) gab

Bathowk (Bat-Hornet-Hawk)

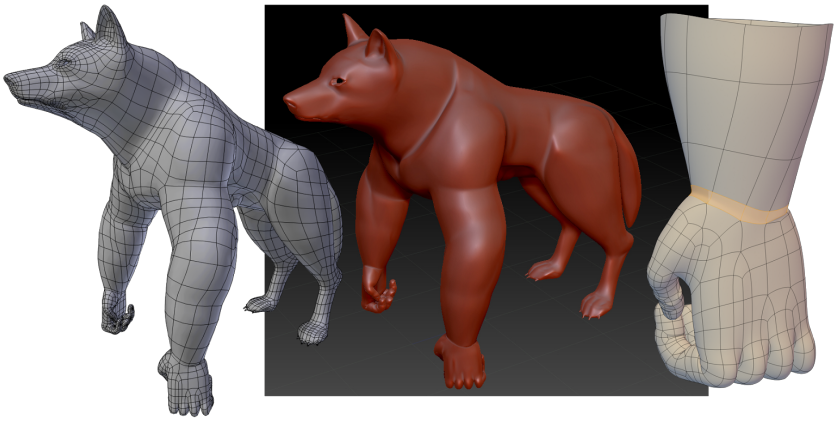


Squrtle-gator (Squid-Turtle-Alligator gar)



Locatergontula (Locust-Caterpillar-Dragonflie-Tarantula)

Das Mesh



Alle meine 3D Modelle sind in Blender entstanden, als erstes das Modell des Wozillas. Bei der Modellierung eines Meshes muss besonders auf die Topologie und den Edgeflow geachtet werden. Damit bei der Animation keine Fehler auftreten muss der Edgeflow über die Topologie des Meshes sinnvoll erhalten werden. Dazu müssen Triangles vermieden werden. Besonders in den Bereichen wie z.B. dem Handgelenk, wo Hand und Unterarm zusammenfließen. Die Hand hat eine höhere Polygondichte als der Unterarm und ist nicht so einfach miteinander zu verknüpfen.

Texturierung

Erste Texturierungsversuche wurden mit der (mir zuvor unbekannten) Software ZBrush (siehe Regi gemacht und haben es auch zu einem guten Resultat gebracht. Die Einarbeitung in ZBruch war sehr langwierig und es mangelte an einem guten Workflow mit ZBrush und Blender. Texturen konnten nicht einfach exportiert werden. Da sich die Texturkoordinaten in Zbrush zu Blender geändert haben, war es mir nicht möglich die Texturen auf das in Blender modellierte Modell anzuwenden. Daher habe ich bei den anderen Modellen zur Texturierung nicht mit ZBrush gearbeitet, obwohl der große Vorteil von ZBrush ist, dass das 3D-Modell vor dem Texturieren nicht zerschnitten und aufgefaltet werden muss um endgültige UV-Koordinaten zu enthalten. Die Farbwerte werden zuvor nicht auf UV-Basis

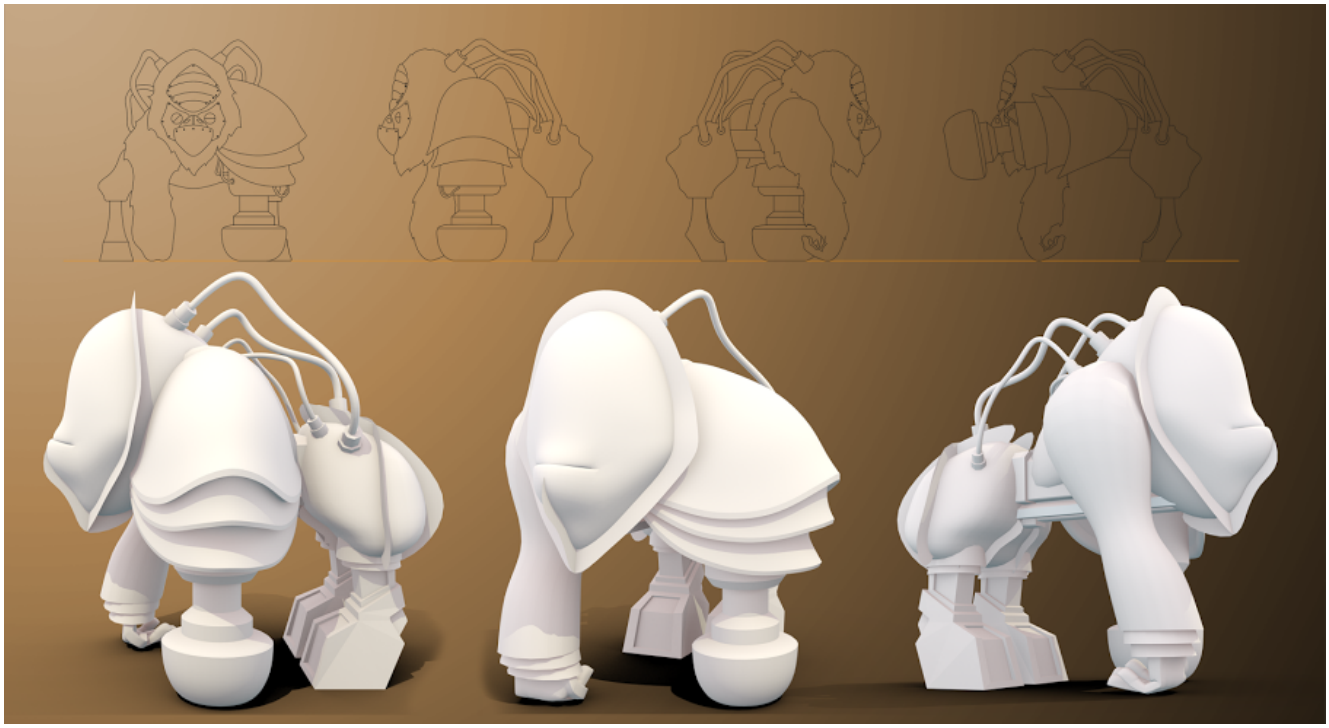
gespeichert sondern auf Basis der Vertices. Je feiner das Polygonnetz des 3D-Modelles ist desto mehr Farbinformationen kann es aufnehmen und desto feiner ist die Auflösung der Textur.
Weitere Texturen wurden “back to the roots!” mit klassischem UV-unvrapping in Blender und Photoshop erstellt. Als Texturquellen habe ich die Google Bildersuche und cgtextures.com verwendet, zudem eine eigene Zusammenstellung von eigenen Texturen.

Silhouetten helfen sehr dabei zu Entscheiden ob eine Kreatur als solche erkannt wird und dementsprechend funktioniert.

Auf diese Arbeit habe ich mich sehr gefreut, da sie mich vor eine neue Herausforderung gestellt hat. Eine eigene Skizze anzufertigen um sie dann in 3D umzusetzen ist meiner Meinung nach einfacher da man selbst weiß, welche Vorstellungen über Aufbau und Fähigkeiten des Tieres der Skizze zu Grunde liegen. So kann es bei Zeichnungen anderer sein, dass man die Zeichnung ganz anders interpretiert als der Zeichner selbst. Vieles in einer Zeichnung steckt noch in den Köpfen der Zeichner und ist somit für andere nicht gleich zu sehen. Ein 3D-Modell hingegen kann von allen Seiten betrachtet werden und ist konkreter in Dimension und Aussage.
Des Weiteren können sich daraus Probleme ergeben, dass man viele Sachen zwar zeichnen kann, diese aber als Modell nicht funktionieren, nicht gut aussehen oder einfach nicht umsetztbar sind.

- Beispielsweise Fell: in einer Game Engine können “noch” keine Haare als Partikel-System benutzt werden, Haare können nur über eine Textur realisiert werden, daher sind zusätzliche Flächen nötig, die mit einer Alpha Textur Transparenz zulassen
- Die Hinterbeine des GorillaBots waren aus der Skizze ersichtlich ohne Gelenke, also steif, woraus sich natürlich Animationsprobleme ergaben

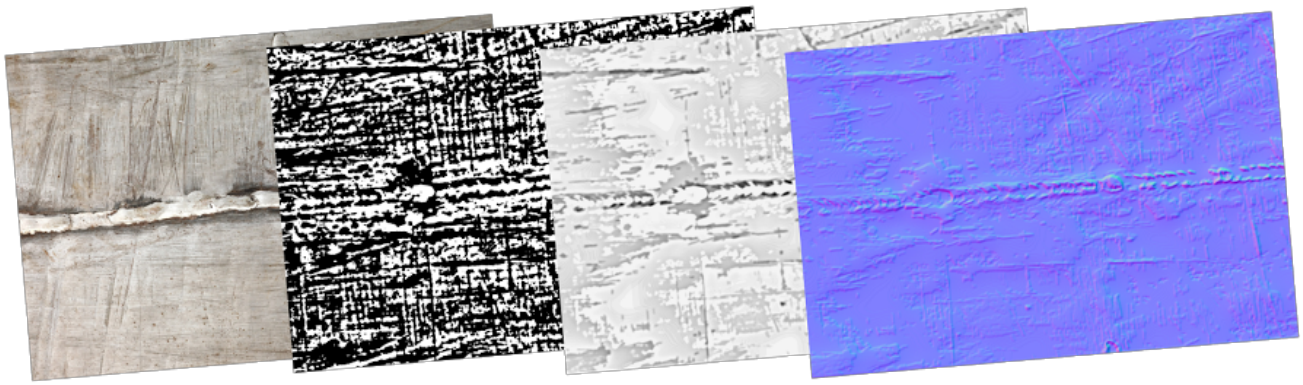
- Mechanische Schulter: Der Übergang von steifen metallischen Gegenständen zu organischen bereitete ebenfalls Probleme bei der Animation.
- Kabel: Auch die Kabel mussten besonders animiert werden, zudem bereitete der Export zu Unity Probleme.
- Metalltexturen sind zwar realisierbar, aber sind sie auch Game Engine tauglich?



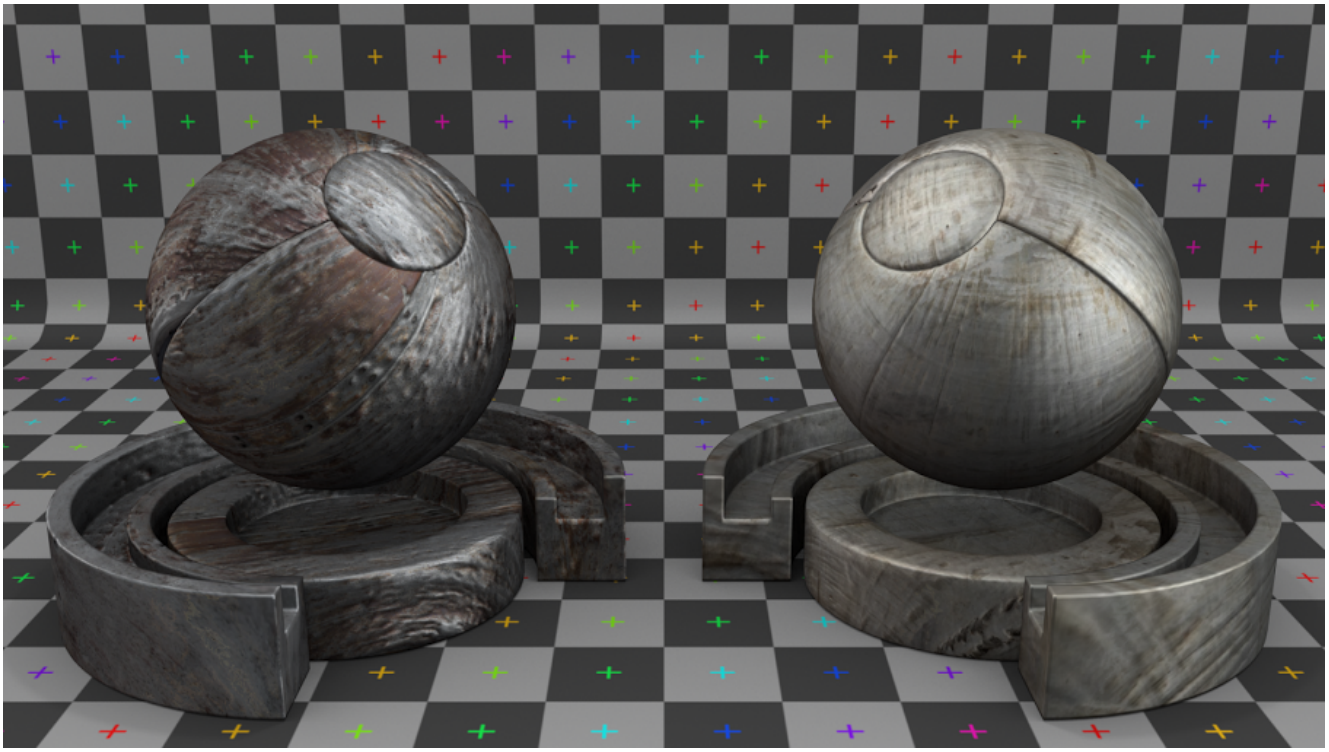
Die Umsetzung eines Characters nach einer Skizze von Stefan Hartwig (Gorilla-Bot)

Eine Metalltextur macht noch lange kein Metall! Ohne die richtigen Einstellungen sieht es aus wie Beton.

Wie sieht Metall aus? Welche Farbe hat es? → Textur
Wie verhält es sich bei Licht? Wie reflektiert es? → Reflection Map (Light Probes)
Wo reflektiert es? → Dirt Map
Wie glänzt es?
Wo glänzt es? → Specularity Map
Welche Struktur hat es? → Normal Map

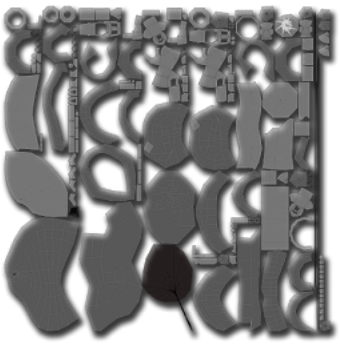


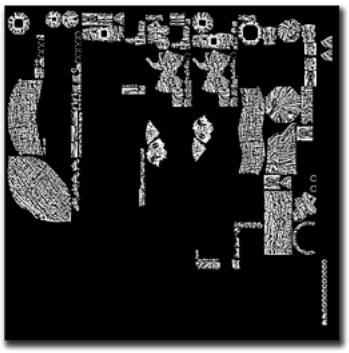
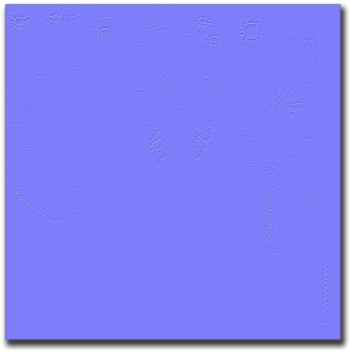
Diffuse Map | Specularity Map | Dirt Map | Normal Map



Mit diesen Überlegungen kam ich zu einem zufriedenstellenden Game Engine tauglichen Metall Look.

Diese Überlegungen flossen ein in die Bearbeitung der Textur, die ich mit Shader Map, einem Normal Map Generator, erstellt habe. Darauf folgte das Baken der fertigen Textur.

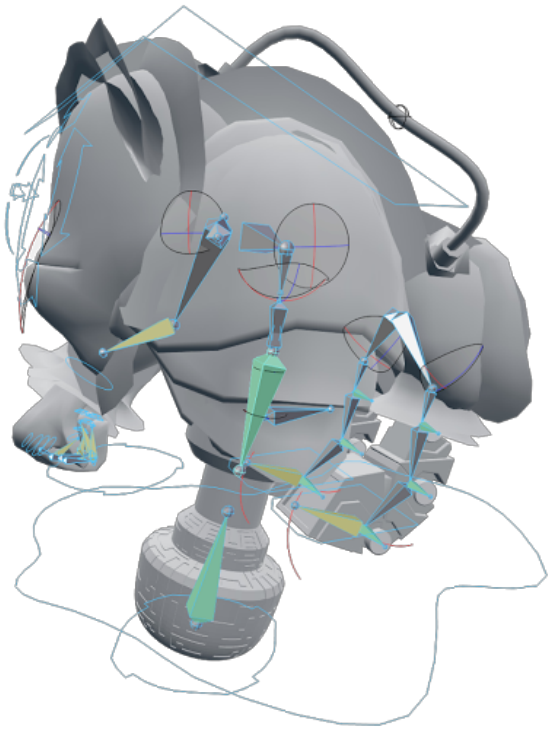



Animationen

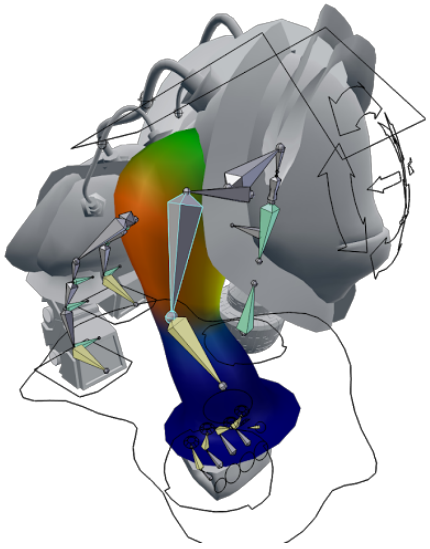
Rigging

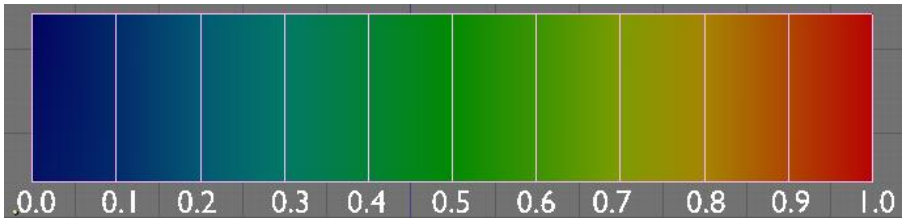
Der GorillaBot besteht aus über 35.000 Triangles, 55 Bones und einer 1024 x 1024 Pixel großen Diffuse Map, Specularity Map und Normal Map. Die Kabel wurden mit Splines erzeugt und über deren Knotenpunkte animiert. Es stellte sich später heraus, dass diese Art der Animation nicht zu Baken ist, in Unity 3D nicht funktioniert und zu erheblichen Problemen führte. Das Problem konnte nur dadurch behoben werden, in dem jedes Kabel für sich mit Bones versehen wird um es zu animieren. Je mehr Bones pro Kabel desto natürlicher das Verhalten. Das hat zu 16 weiteren Bones geführt also insgesamt 71 Bones. Mehr Bones bedeuten mehr Arbeit und mehr Speicher. Jedes Vertex kann nur von Maximal 4 Bones beeinflusst werden (Unity 3D Limits). Man sagt 30 Bones pro Character sind gut für PCs, für mobile Anwendungen noch weniger. Ob diese Vielzahl an Bones tatsächlich zu Problemen im Spiel führt, konnte leider noch nicht getestet werden, da der Charakter noch nicht eingefügt wurde.



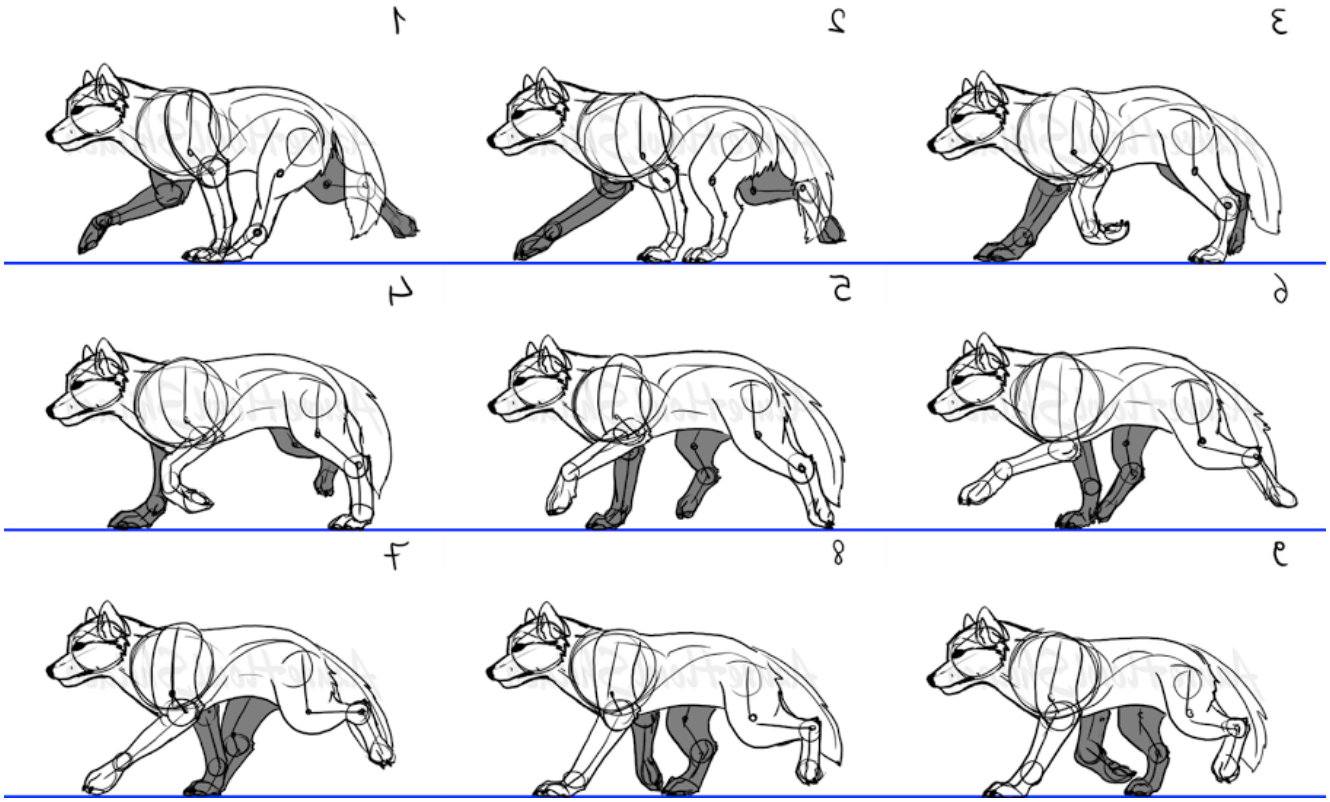
Skinning

Nachdem das Skelett bzw. Rig erstellt wurde, kann es mit dem Polygonnetz gekoppelt werden. In einem weiteren Arbeitsschritt, dem so genannten Skinning, sind oft noch einige kleinere Fehler auszubügeln, die beim Koppeln des Meshes an das Rig aufgetreten sind. So ergaben sich zum Beispiel an der Metallschulter des GorillaBots unschöne Durchdringungen (Intersections) bzw. Fehler, die durch Weight Painting behoben werden konnten, indem festgelegt wurde, dass der entsprechende Bone auf den Bereich an der Schulter keine Einwirkung haben sollte, sondern nur auf den unteren Teil des Beins.





Animation



Jedes Tier sollte für ihre normale Bewegung (laufen bzw. schwimmen oder fliegen) animiert werden sowie in einer sog. Idle-Bewegung (in Pausen, in den die Figur nicht bewegt wird), eine Hit-Animation (wenn die Figur getroffen oder beschädigt wird), einer Die-Bewegung sowie einer Attack-Bewegung. Alle bis auf letztere wurden für die Charaktere umgesetzt.
Der Walkcycle wurde nach dem "the animator's survival guide" umgesetzt. Schließlich mussten die Charaktere und die Animationen aus Blender gebaked, als *.fbx Dateiformat exportiert und in Unity importiert werden. Das Baken und Exportieren in das *.fbx Format brachte nochmal seine eigenen Probleme mit sich und nahm sehr viel Zeit in Anspruch da das nicht mit einem Klick zu handeln war. Blenders *.blend Dateiformat wird von Unity zwar unterstützt, aber das Handling schien sich vom fbx format zu unterscheiden. Da wir im Design Team mit unterschiedlicher Software

modellierten, haben wir uns auf das FBX Dateiformat geeinigt. So konnte das Programmierteam mit allen 3D-Modellen gleich verfahren.

Leveldesign

Allgemein

Marcel Müller

Wie in jedem Spiel üblich, muss sich das Level beziehungsweise das Leveldesign dem Spielprinzip und den Interaktionsmöglichkeiten des Spielers mit der Umwelt anpassen. So ist es für ein 2D- oder 2.5D-Jump'n'Run nicht nötig das Level in die Tiefe auszumodellieren, für ein Rennspiel macht es keinen Sinn Strecken mit engen und verwinkelten Gängen zu entwerfen. Das Level muss garantieren, dass der Spieler beim Erreichen des Levelziels herausgefordert wird, es darf ihn auf der anderen Seite aber auch nicht so weit behindern, dass der Spieler den Spaß und die Lust am Spiel verliert.

In unserem Fall sind dies die grundsätzlichen Spielmechanismen:

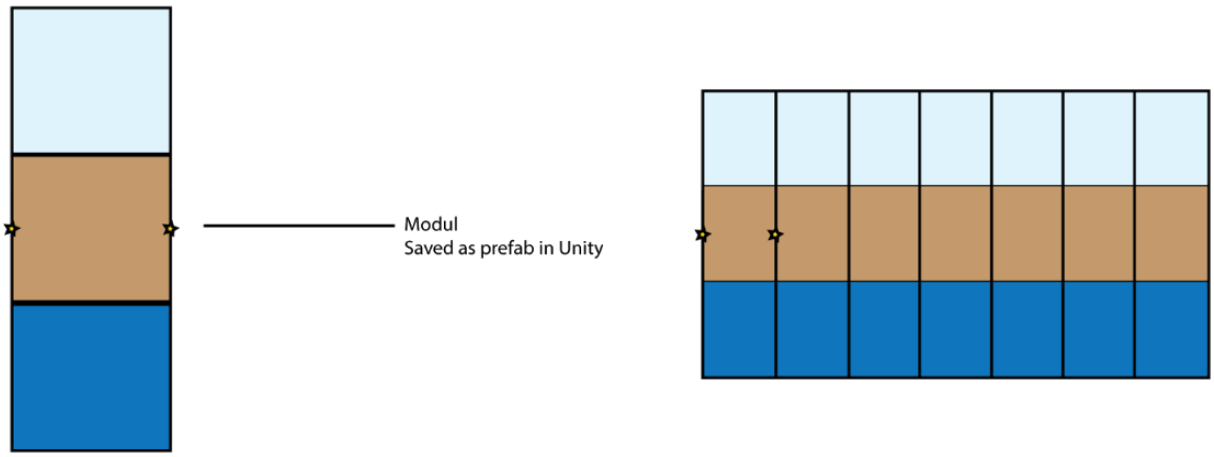
- Die Steuerung der Spielercharaktere unterschiedet sich zwischen den Luft-, Erd- und Wassercharakteren erheblich.
- Die Spielercharaktere bewegen sich nicht ohne Eingaben des Spielers (Ausnahmen: siehe Punkt 4).
- Der Spielercharakter bewegt sich wie auf Schienen von links nach rechts durch das Level. Eine Bewegung in die Tiefe ist, sofern sich die "Gleise" nicht in die Tiefe bewegen, nicht möglich.
- Die Bewegung in die Höhe ist möglich. Im Falle des Bodencharakters fällt er nach einem Sprung allerdings sofort wieder auf den Boden. Der Luftcharakter muss durch ständige Eingaben seitens des Spielers am Runterfallen gehindert werden.

Zu Beginn des Projektes haben wir uns verschiedene Spiele angeschaut, die nach einem ähnlichen Spielprinzip, nämlich das schnelle Erreichen des Levelendes, wie unser Spiel funktionieren und wie die Level in diesen Spielen aufgebaut sind. Ziel war es herauszufinden, wie die Level allgemein in allen drei Dimensionen strukturiert und verteilt sind, wie lange diese Level und Abschnitte dauern und natürlich welche Herausforderungen sie bieten.

Levelaufbau in "Crazy Doc's Evolution Run"

Anders als bei unserem Spiel, sind die Level in Spielen wie "Trials" oder der "Sonic"-Reihe allerdings statisch aufgebaut und bestehen nicht aus mehreren zufällig aneinander gereihten Abschnitten und übereinander liegenden verschiedenen Ebenen für mehrere Spieler. Unsere Level sind wie folgt aufgebaut:

- Ein Level besteht aus fünf bis sieben Sektoren und dauert ungefähr drei Minuten.
- Sektoren bestehen aus unterschiedlich vielen, zufällig aneinander gereihten Modulen, mindestens aber aus zweien, dem Spawnmodul am Anfang des Sektors und dem Fightmodul am Ende.
- Der Spieler startet im Spawnmodul. Im Fightmodul muss der Spieler Gegner bekämpfen um zum Despawnpunkt am Ende des Moduls zu gelangen.
- Erreicht der Spieler den Despawnpunkt am Ende eines Sektors wird ihm ein zufälliger neuer Charakter zugeordnet, der im Spawnmodul eines neuen Sektors wieder auftaucht.
- Stirbt der Spieler zwischen dem Spawn- und dem Fightmodul, so wird er zum Checkpoint am Anfang des Spawnmoduls zurück gesetzt.
- Stirbt der Spieler während des Kampfes am Ende des Levels im Fightmodul, so wird er an den Checkpoint am Anfang des Fightmoduls zurück gesetzt.
- Ein Modul besteht aus der Luft-, Erd-, und Wasserebene.



Aufteilung der Level in Module und Sektoren

Daraus ergeben sich folgende Restriktionen beim Aufbau des Levels:

- Alle Module brauchen eine gemeinsame Schnittstelle, das heißt die Levelgeometrie am Anfang und am Ende eines Levels muss deckungsgleich sein, damit es beim zufälligen Aneinanderreihen der Module keine sichtbaren Kanten oder Löcher gibt.
- Wegen der "Stapelung" der drei Ebenen aufeinander können die Level in den einzelnen Ebenen nicht vertikal aufgebaut sein. Die Laufwege der Charaktere dürfen nicht viel höher oder tiefer liegen als ihre Spawnpunkte. Besonders eingeschränkt in seiner Bauhöhe wegen der anderen Ebenen darüber und darunter ist die Erdebene.
- Wegen der verschiedenen Filter auf der Kamera des Wassercharakters muss es zwischen der Erd- und der Wasserebene einen sichtbaren Trenner geben, damit der Spieler auf der Erdebene nicht in das Wasser schauen kann.

Gemeinsam haben die Spiele, die wir zur Referenz heran gezogen haben, wie die meisten 2D-Jump'n'Run Spiele, dass die Level aufgrund ihres Aufbaus herausfordernd sind, soll heißen die Schwierigkeit ergibt sich aus dem Überwinden des Levelaufbaus, nicht etwa aus Kämpfen oder Rätseln. So müssen Höhenunterschiede überwunden werden, ohne an den Kanten anzuecken oder Fallen durch geschicktes Springen ausgewichen werden, das Ganze während man sich mit einer relativ hohen Geschwindigkeit vorwärts bewegt.

Die einfachste Methode solche Schwierigkeiten im Level umzusetzen sind Dekorationsobjekte wie Kisten und Fässer. Höhenunterschiede direkt in der

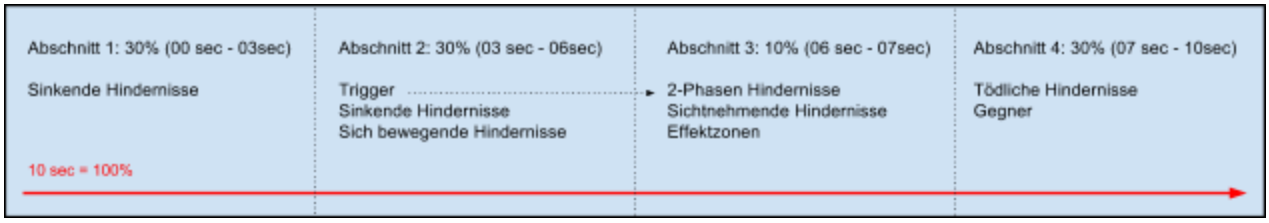
Levelarchitektur, also im Boden oder Decke erschweren das Aneinanderpassen der verschiedenen Module und schränken die Anzahl und Art der Dekorationsobjekte ein, die man darauf platzieren kann. .
Aus dem Gedanken der passiven, sich nicht bewegenden Hindernisse sind auch Ideen für Objekte entstanden, die ein bestimmtes Verhalten zeigen. Um den Programmierern, die die zugehörigen Scripts erstellen mussten, die Arbeit zu erleichtern, aber auch um verschiedene Verhaltensweisen beliebig oft auf unterschiedlichen Objekten verwenden zu können, wurden zu Beginn der Projektarbeit die Hindernisse ihrem Verhalten entsprechend in verschiedene Kategorien eingeordnet:

1. Passive Hindernisse zeigen kein Verhalten. Kollidiert der Spieler mit ihnen, verliert er seine Geschwindigkeit. Beispiele sind Kisten, Fässer, Gebäude oder Steine.
2. Passive/Zerstörbare Hindernisse zeigen kein Verhalten und verlangsamen den Spieler bei einer Kollision, können aber von ihm zerstört werden, etwa mit einer Attacke. Beispiele sind Kisten, Zäune und Wände.
3. Tödliche Hindernisse töten den Spieler bei Berührung sofort. Beispiele sind Tret- oder Seeminen, explosive Ballons oder komplette Bereiche im Level, die der Spieler nicht betreten soll (etwa eine Lavagrube). Der Tod der Spielfigur wird ausgelöst, sobald er eine bestimmte Trigger- oder Hitbox betritt. Im Falle des Todes des Charakters wird er an den letzten Checkpoint zurück gesetzt. Alternativ zum sofortigen Tod kann dem Spieler auch ein bestimmter Schadenswert pro Sekunde die er sich in der verbotenen Zone aufhält zugefügt werden.
4. Sich bewegende Hindernisse sind zum Beispiel durch das Level fliegende Raketen oder Vogelschwärme. Sie verlangsamen den Spieler, sobald er den Bereich dieses Objektes betritt und ziehen ihm gegebenenfalls Energie ab.
5. Sinkende Hindernisse fügen sich in den Levelboden ein und beginnen zu sinken, sobald der Spieler darauf steht. Sinkt das Hindernis mit dem Spieler unter einen gewissen Punkt, stirbt die Spielfigur oder nimmt Schaden je tiefer er sinkt. Beispiele sind auf dem Wasser treibende Baumstämme oder auch Treibsand.
6. Effekt-Zonen ändern einen Parameter des Spielercharakters. So kann er schneller und stärker oder langsamer und schwächer werden, Energie verlieren oder die Sicht des Spielers wird durch einen Filter gestört.
7. 2-Phasen-Hindernisse bewegen sich zwischen zwei Zuständen hin und her. Beispiele sind Türen oder Schranken die sich öffnen und schließen.

Jede dieser Kategorien beschreiben nur ein grundlegendes Verhalten, das durch ein Script gesteuert wird. Natürlich können auch mehrere Scripte auf einem Objekt liegen. So bewegt sich die oben bereits erwähnte Kreissäge an den Teleskoparmen immer zwischen zwei Zuständen (eingezogen und ausgefahren,

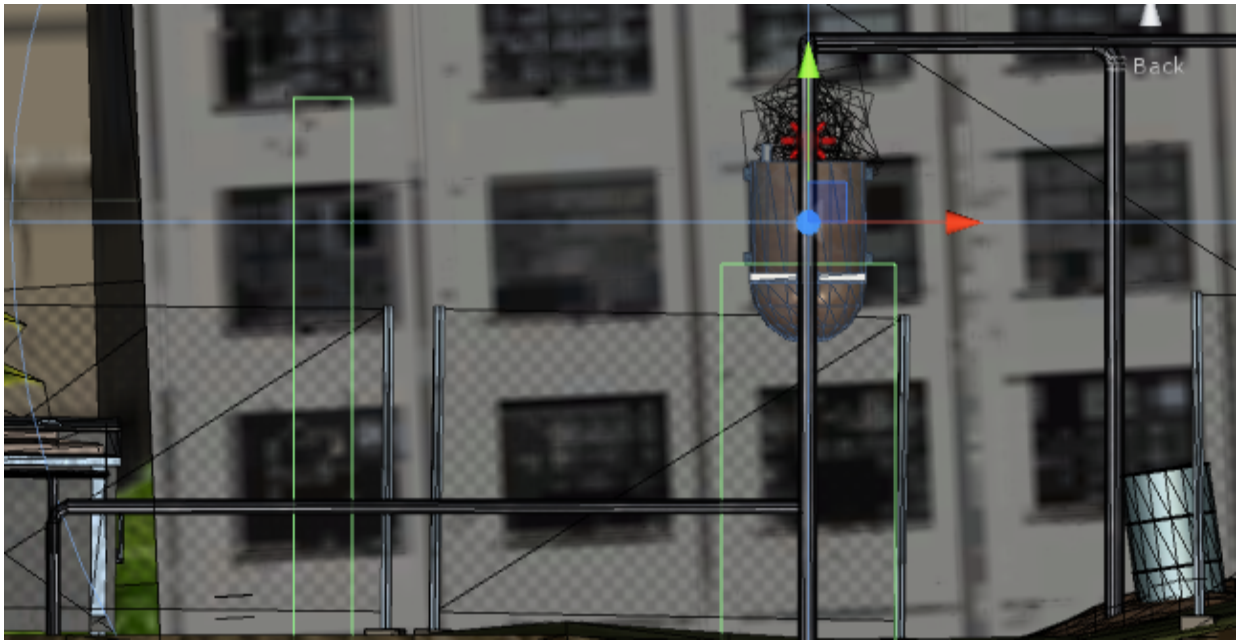
2-Phasen-Hindernis) Trotzdem verliert der Spieler auch Gesundheit sollte er das Sägeblatt berühren, was dem Verhalten einer Effekt-Zone oder auch eines tödlichen Hindernisses entspricht.

Schwierigkeiten bei der Erstellung und der Platzierung solcher Hindernisobjekte und Dekorationsobjekte macht unter Anderem die Aufteilung des Moduls in drei Ebenen. Insbesondere für die Luftebene fällt es schwer sich logisch erklärbare Hindernisse einfallen zu lassen. Da prinzipiell alles was nicht irgendeine Art von Auftrieb hat unweigerlich von der Schwerkraft zu Boden gezogen wird, beschränkte sich die Auswahl an Dekorationsobjekten für die Luft anfangs auf Maschinen mit Triebwerken oder Vögel, von denen auch einige umgesetzt wurden. Ein Äquivalent von Erhebungen oder Löchern im Boden, also Hindernisse bedingt durch die Levelarchitektur, gibt es für die Luftebene nicht.
Die Lösung dieser Probleme brachte die Überlegung, dass alle Levels bloß Testumgebungen des Professors sind und sich somit nicht in der realen Welt befinden müssen. Daher ist es durchaus möglich, dass alle Sektoren von einem Dach begrenzt werden von dem Plattformen herab hängen können. Auf diesen Plattformen lassen sich dann die gleichen Hindernisse platzieren, die auch auf dem Boden oder unter Wasser zum Einsatz kommen.



Ursprüngliche Aufteilung des Levels nach Hindernissen

Ursprünglich war geplant die Hindernisse ihrer Art nach im Sektor zu verteilen. So sollten passive Hindernisse nur im ersten Drittel des Sektors auftauchen, tödliche nur im letzten Drittel. Mit dieser Aufteilung wollten wir ein stets forderndes aber auch faires und frustfreies Spielen garantieren. Wegen der Änderungen bezüglich der zufälligen Generierung der Sektoren war diese statische Aufteilung der Level allerdings nicht mehr möglich.
Stattdessen haben wir darauf geachtet, dass Hindernisse gleicher Art in einem Modul nicht zu nah aufeinander folgen oder direkt am Anfang oder Ende des Moduls, und so potentiell direkt am Anfang des Sektors, liegen.



Sobald der Spieler den Trigger betritt (linker grüner Bereich) öffnen sich die Behälter rechts. Betritt der Spieler den rechten grünen Bereich bei offenen Behältern nimmt er Schaden

Ausgelöst werden die meisten Hindernisse beziehungsweise dessen Wirkung auf den Spieler durch einen Trigger. Eine einfache Art und Weise Trigger umzusetzen sind leere Spielobjekte mit einem Collider. Collider sind bestimmte Volumen im Raum, die registrieren sobald sich andere Objekte in sie hinein oder hinaus begeben. An ein Script gekoppelt, das beispielsweise dafür sorgt, dass der Spieler Energie verliert oder ein Objekt, zum Beispiel eine Tretmine zerstört, dienen sie als Aktivierung der verschiedenen Hindernisse.
Die leeren Spielobjekte können überall im Level verteilt werden und müssen sich nicht in der Nähe des Effekts befinden, den sie auslösen. Ein Empty auf einem Bodenschalter kann so das Schließen einer Tür an einer anderen Stelle im Level auslösen. Ein Beispiel aus dem Spiel sind an Rohren hängende Schmelztiegel, die sich öffnen wenn der Spieler auf einen Bodenschalter ein paar Meter vor den Tiegeln tritt.

Texturierung

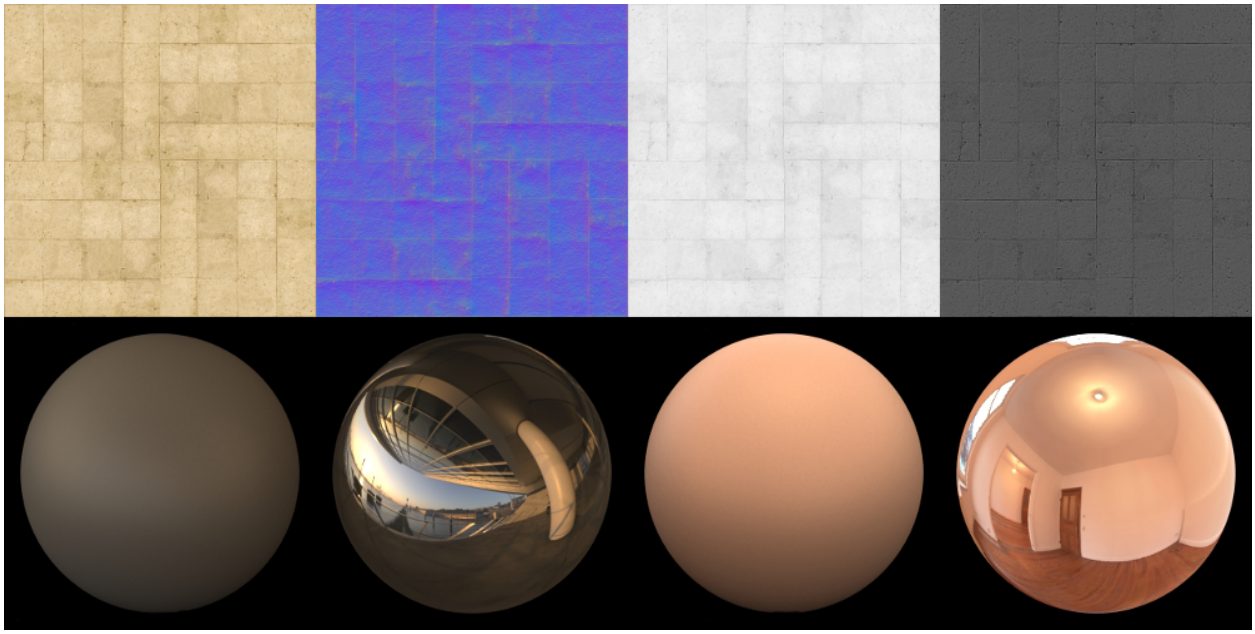
Marcel Müller

Die Modellierung der Hindernis- und Dekorationsobjekte ist in erster Linie eine Fleißarbeit und verlangt je nach Objekt relativ wenig kreativen Einsatz. Besonders in einem Plattformer dienen solche Objekte eher der Bildung der Laufstrecke als der optischen Gestaltung des Levels.
Anhängig von der Art des Objektes nimmt die Modellierung unterschiedlich viel Zeit in Anspruch, organische und runde Objekte sind sicherlich aufwendiger zu modellieren, als geradkantige und eckige Objekte, wie Kisten, Wände oder Rohre.

Zusätzlich ist es relativ unüblich und auch unklug allzu viel Arbeit in ein Dekorationsobjekt zu stecken. Mehr Arbeit bedeutet meist auch mehr Polygone. Gerade bei Modellen die eher im Hintergrund stehen und in der Geschwindigkeit des Spiels leicht übersehen werden können, rechnet sich der Mehrverbrauch an Rechenkapazität nicht. Ich gehe in diesem Teil der Dokumentation daher genauer auf die Texturierung und die Erstellung von Animationen Nicht-Spielercharakter-Modelle ein, insbesondere da sich diese Arbeitsschritte teilweise doch erheblich von denen der Erstellung von Texturen und Animationen für Spielercharaktere unterscheiden.

Nachdem die Geometrie eines Objektes also fertig ausgearbeitet ist folgt für gewöhnlich das Texturieren. Sollten sich bestimmte Teile der Geometrie während einer Animation strecken oder stauchen, so passiert das gleiche auch mit der Textur. Um diesen Effekt besser beobachten und ihm eventuell durch die Bearbeitung der Texturen oder das Verstecken der entsprechenden Geometrie entgegenwirken zu können, erfolgt das Texturieren für gewöhnlich vor dem Animieren.

Im folgenden erkläre ich den Prozess des Texturierens der von mir erstellten Levelobjekte in Blender und das darauf folgende Importieren und Erstellen der passenden Materialien und Shader in Unity. Wegen der großen Menge an Objekten, die für den Leveleditor erstellt werden mussten habe ich darauf verzichtet für jedes Objekt eine eigene Textur zu erstellen. Stattdessen habe ich fertige, für diesen Zweck vorgesehene Texturen aus dem Internet benutzt. Natürlich habe ich darauf geachtet, dass die entsprechenden Bilddateien frei für unsere Zwecke benutzt werden konnten. Dies gilt allerdings nur für die Diffuse Textur. Alle weiteren Texturen wurden aus der Diffuse Map generiert.



Von oben nach unten und links nach rechts: Diffuse Map, Normal Map, Height Map, Specular Map, Environment Map

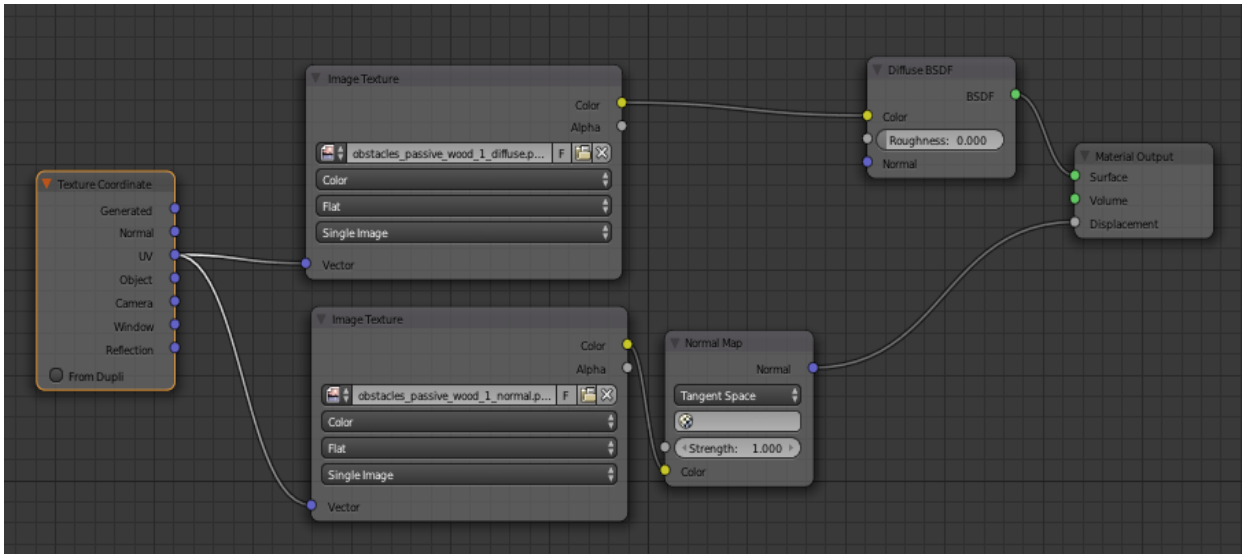
Je nachdem wie ein Objekt oder nur ein Material auf einem Objekt aussehen soll, kommen unterschiedliche Arten von Texturen und unterschiedliche Shader, die mit diesen Texturen umgehen können, zum Einsatz. Im Folgenden eine Auflistung der verschiedenen Arten von Texturen, die bei der Erstellung der Level zum Einsatz gekommen sind:

1. Diffuse Map: Die Diffuse Map ist die nachher sichtbare Textur, sie wird ohne weitere Um- oder Verrechnung auf das Objekt "geklebt". Man kann sie sich wie die Farbe auf einer Leinwand vorstellen.
2. Height Map: Die Height Map ist eine Art der Bump Map und besteht ausschließlich aus Grauwerten. Die verschiedenen Helligkeitswerte repräsentieren dabei ein Höhenrelief, weiß steht für die höchste Stelle, schwarz für die niedrigste. Je dunkler die Farbe an einer bestimmten Stelle ist, desto dunkler wird die Oberfläche des Objektes an dieser Stelle gerendert.
3. Normal Map: Die Normal Map besteht aus RGB-Farbwerten von denen jeweils ein Kanal eine Richtung eines Normalenvektors darstellt. Anhand der Normalen berechnet der Shader welche Stellen der Diffuse Map bei welchem Betrachtungswinkel wie stark aufgehellt oder verdunkelt werden müssen. So entsteht der Eindruck von Schatten und somit mehr Geometrie.
4. Specular Map: Die Specular Map besteht ausschließlich aus Grauwerten. Je heller der Farbwert einer bestimmten Stelle, desto mehr des einfallenden Lichtes an dieser Stelle wird reflektiert. Die Farbe Schwarz bedeutet, dass kein Licht an dieser Stelle reflektiert wird. Unter anderem bei gerostetem

Metall kommen diese Texturen zum Einsatz. So kann man kontrollieren, dass das noch unbeschädigt Metall Licht (in einem gewissen Maß) reflektiert, die rostigen Stellen allerdings nicht.

5. Reflection Map/Environment Map: Fügt man einem Objekt eine Reflection Map hinzu, entsteht der Eindruck, als würde das Objekt seine Umgebung reflektieren. Tatsächlich wird aber nur die Reflection Map verzerrt auf dem Objekt dargestellt.

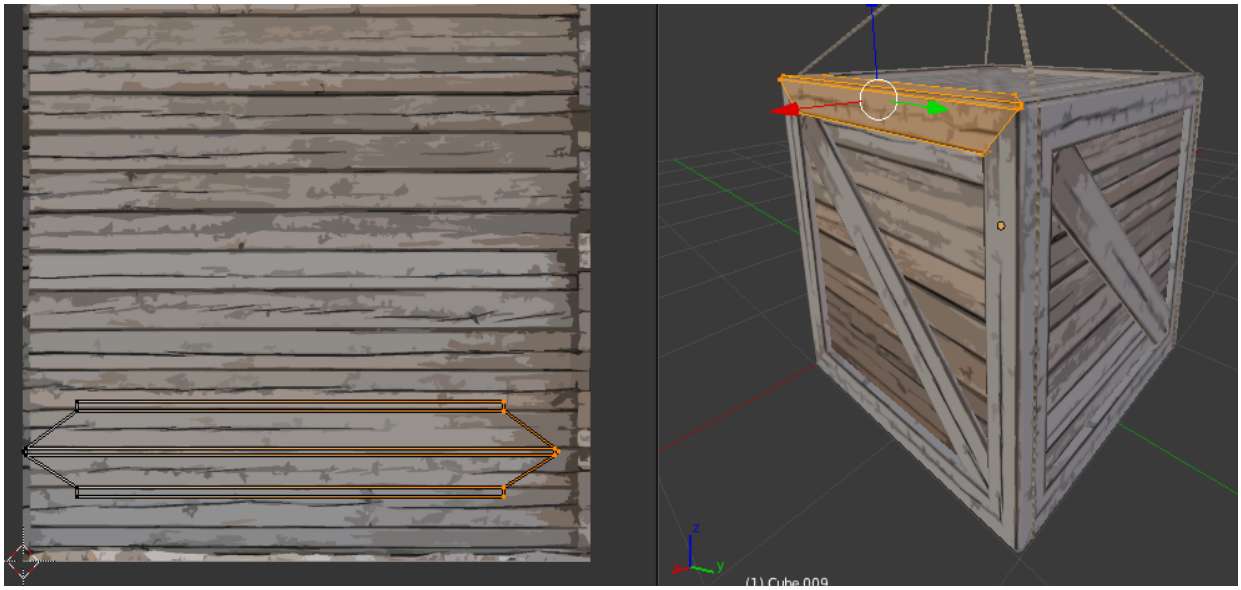
Bevor man die Textur auf das Objekt mappen kann, muss man ein oder mehrere Materialien für das Objekt anlegen und den gewünschten Flächen zuordnen. Materialien lassen sich in Blender durch das Verbinden verschiedener Nodes anlegen und manipulieren. Nodes steuern unter anderem Farben, Texturen, verschiedene Eigenschaften der Oberfläche, wie ihre Reflektivität oder ihre Glossiness oder eine Mischung all dieser Werte. Da Unity aber keine Materialien aus Blender mit dem Objekt importiert macht es keinen Sinn, das Material bereits in Blender zu bearbeiten. Die UV-Map allerdings wird direkt auf dem Objekt gespeichert und von Unity erkannt. Daher reicht es in Blender einen Texture-Coordinate Node und einen Diffuse-Color Node mit Image Input anzulegen.
Der Texture-Coordinate Node steuert wie die Textur (oder die Texturen, sollte man für jede Textur den gleichen Texture-Coordinate Node nutzen) auf das Objekt gemappt wird. Verbindet man den UV-Ausgang dieses Nodes mit dem Vektor-Eingang des Image-Texture Nodes, bildet der Shader die im Image-Texture Node definierte Textur der UV-Map des Objektes entsprechend darauf ab. Der Image-Texture Node muss daraufhin mit dem Color-Eingang des Diffuse-Nodes verbunden werden. Zuletzt wird der Ausgang dieses Nodes mit dem letzten, dem Material-Output Node, verbunden. Dieser Node steuert wie die ihm zugetragenen Informationen mit dem Objekt verrechnet werden, ob sie etwa auf seiner Oberfläche abgebildet werden oder zum Beispiel dazu verwendet werden, seine Oberfläche zu verformen.



Der Node-Editor in Blender

Um die Texturen aber auf dem Objekt abbilden zu können, braucht es noch die UV-Map. Die UV-Map ist eine zweidimensionale Repräsentation der Oberfläche des Objektes. Dabei wird die Geometrie des Objektes so auseinander gefaltet, dass eine möglichst kompakte 2D-Struktur entsteht. Jedem Punkt auf der Objektoberfläche genau einem Punkt auf der UV-Map zugeordnet. Indem man wieder jedem Punkt auf dieser UV-Map einem Punkt auf einer Textur zuordnet, stellt man eine Verbindung zwischen den Punkten auf der Oberfläche des Objektes und den Punkten auf der Textur her. So lassen sich zweidimensionale Texturen auf dreidimensionale Objekte mappen.

Beim UV-Unwrapping (dem Auseinanderfalten der dreidimensionalen Oberfläche in die zweidimensionale UV-Map) ist darauf zu achten, dass das Objekt in so wenig wie möglich einzelne Teile aufgeteilt wird um sichtbare Kanten auf der Textur zu vermeiden. Bei vierbeinigen Charakteren wählt man daher zumeist den Bauch als Schnittkante, da er zum Boden zeigt und so später nicht sehr gut zu sehen ist. Bei statischen Objekten wie Kisten oder Ähnlichem eignen sich neben der Unterseite auch spitze Kanten zwischen zwei Flächen. Kontrollieren lässt sich die Faltung durch das händische Setzen von Schnittkanten, sogenannten Seams.



Links: UV-Map mit dahinter liegender Diffuse Map, rechts: Viewport mit texturiertem Objekt

Im Grunde gibt es nun zwei Möglichkeiten weiter daran zu arbeiten, die Textur auf das Objekt zu mappen, die sich hauptsächlich darin unterscheiden, ob man die Textur selbst erstellt oder eine fertige Textur verwendet.

1. In Blender ist es möglich die UV-Map als eigenständige Bilddatei zu exportieren. Lädt man dieses Bild dann dann in ein Bildbearbeitungsprogramm kann man es als Schablone für eine Textur verwenden, die man eigenhändisch darauf zeichnet. Ist die Zuordnung der einzelnen Bereiche der UV-Map zu den den einzelnen Teilen des Objektes, lassen sich im Falle eines Tieres Körperteile wie Augen, Krallen, Haare etc. direkt auf die UV-Map oder ein darüber liegendes Bild malen. Lädt man dieses Bild dann in Blender als Textur und wählt die UV-Map als Output im Texture-Coordinate Node, so liegt die Textur perfekt auf dem Modell.
2. Alternativ kann man eine bereits fertige, nicht auf die UV-Map optimierte Textur als Textur in den Image-Texture Node laden. Danach lässt sie sich sich im UV-Image Editor als Hintergrund hinter die UV-Map legen. Verschiebt man nun die Vertices der UV-Map beziehungsweise des auseinandergefalteten Modells auf der Textur hin und her, verschiebt man die Textur direkt auf dem Modell. Im Viewport sind die Auswirkungen direkt sichtbar.

Für gewöhnlich reicht es nicht aus, einem Objekt nur eine einzelne Textur zuzuweisen um es realistisch aussehen zu lassen. Alle Oberflächen haben verschiedene Eigenschaften, wie etwa ihre Farbe, Reflektivität oder Struktur. Eine Diffuse Map ist bei den meisten Objekten und Oberflächen obligatorisch, ebenso wie eine Art von Bump Map, da fast keine Oberfläche völlig glatt ist. Damit alle

Texturen am Ende perfekt aufeinander passen, werden alle weiteren Texturen normalerweise aus der Diffuse Map generiert.
Das lässt sich im Falle der Specular Map und der Height Map händisch bewerkstelligen. Dazu bedarf es nur der Umwandlung der Farben der Textur in Grauwerte in einem Bildbearbeitungsprogramm und der Anpassung des Kontrastes und der Helligkeit bis das gewünschte Ergebnis erreicht ist.
Dies ist beispielsweise mit Normal Maps aber nicht möglich. Allerdings gibt es Programme wie insaneBumps oder nJob oder Plugins für Photoshop und GIMP die aus einer Diffuse Map alle weiteren gewünschten Texturen generieren. Da wegen der Abhängigkeit zur Diffuse Map alle Texturen die gleiche Größe und Auflösung haben, ist es möglich das Mappen auf das Objekt über den gleichen Node zu kontrollieren.

Für komplexe organische Objekte bietet sich noch eine weitere Möglichkeit der Texturierung an, die das Arbeiten mit UV-Maps weitestgehend überflüssig macht. Beim Texture Painting wird die Textur direkt wie mit einer Sprühpistole auf das 3D-Objekt aufgetragen. Die UV-Map dazu wird automatisch im Hintergrund erstellt und aktuell gehalten. Im in Blender integrierten Texture-Painting Tool lässt sich die so erstellte Textur als Bilddatei zur Weiterverarbeitung oder zur Erstellung anderer Texturen exportieren.

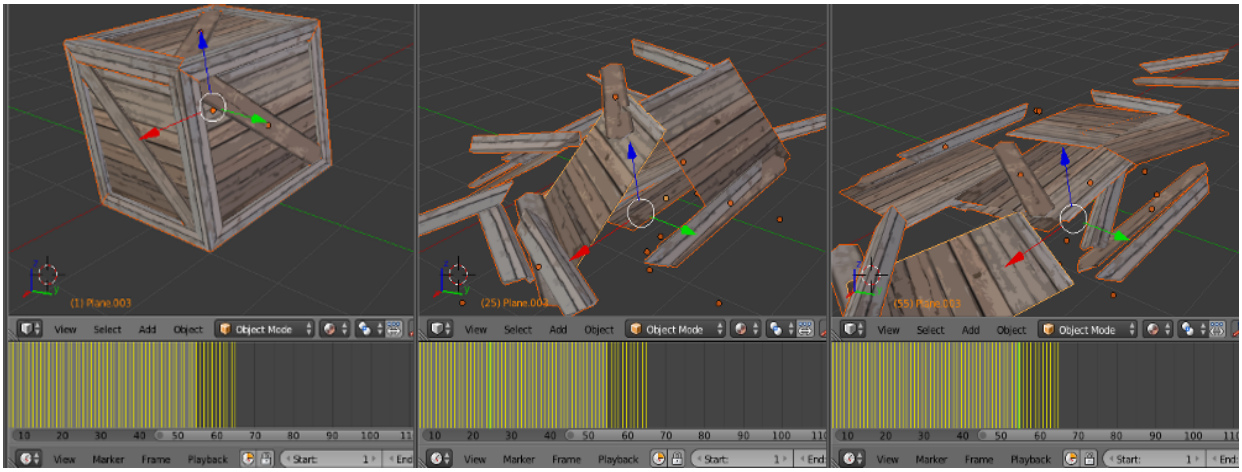
Animationen

Marcel Müller

In manchen Fällen sollen Umgebungsobjekte eine bestimmte Bewegung durchführen oder direkt auf den Spieler beziehungsweise den Spielercharakter reagieren. Beispiele sind hin und her schwingende Äxte, Kreissägen oder Kisten, die bei Berührung mit dem Spieler auseinander brechen. Solche einfachen Animationen lassen sich relativ leicht in Blender ohne den Einsatz von Rigs erstellen. Soll sich ein Objekt beispielsweise einmal komplett um die eigene Achse drehen reicht das Setzen von Keyframes zu unterschiedlichen Zeitpunkten. Ein Keyframe beschreibt den Zustand und alle Positions-, Rotations- und Skalierungsdaten eines Objektes zu einem gewissen Zeitpunkt.

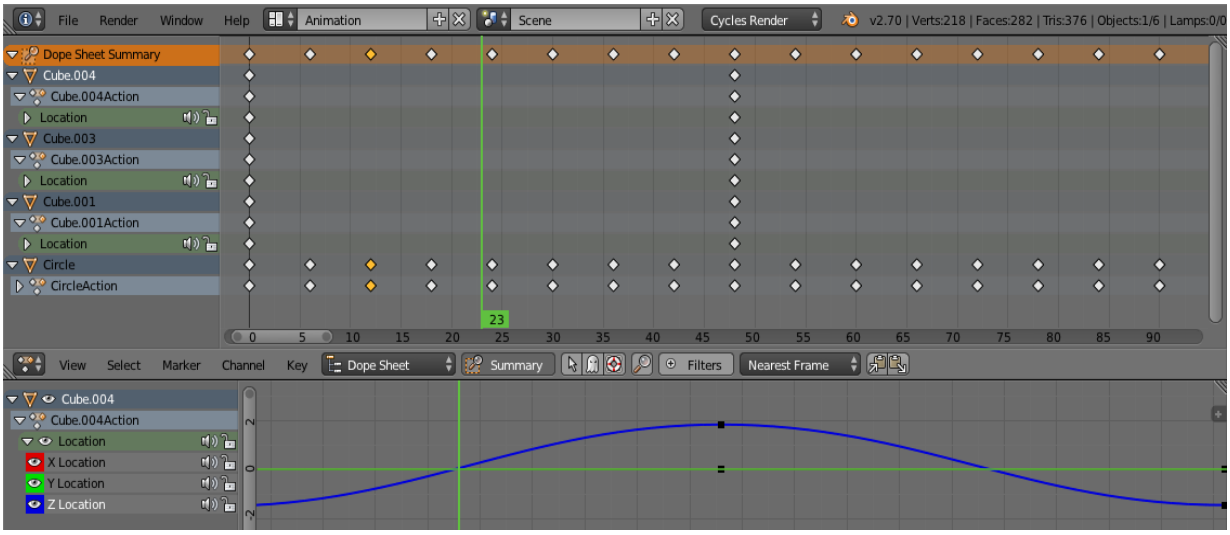
Zu Beginn wird der Ausgangszustand des Objektes “gekeyed”, das heißt ein Keyframe auf diesen Zeitpunkt 0 gesetzt. Danach verschiebt man den Zeitregler beispielsweise auf 10 Frames, rotiert das Objekt um 90 Grad um die gewünschte Achse weiter und setzt einen neuen Keyframe (Damit es sich nicht erst um 180 Grad und dann wieder um 180 Grad in die entgegengesetzte Richtung dreht, sollte man in diesem Fall die Rotation nach 90, 180, 270 und 360 Grad keyen). Das Ganze wiederholt man bis sich das Objekt wieder im Ausgangszustand befindet. Spielt man die Animation nun ab, dreht sich das Modell einmal um die eigene Achse.

Für diese einfache Animation reichen vier oder fünf Keyframes aus. Je komplexer die Animation werden soll, desto mehr Keyframes sind notwendig.



Abspielen einer Animation in Blender. Zeitleiste mit gelben Keyframes im unteren Bereich

Die Kreissägen im Spiel beispielsweise bestehen aus zwei Armen und einem dazwischen besfestigten Sägeblatt. Das Sägeblatt dreht sich kontinuierlich um die eigene Achse, die Arme fahren sich einem Teleskopstab ähnlich in gewissen Intervallen ein und aus. Arme und Sägeblatt müssen also getrennt voneinander animiert werden. Zusätzlich bleibt die Geschwindigkeit mit der sich die Arme auseinander schieben über ihren Weg nicht gleich. Zu Anfang und zu Ende bewegen sich die Arme langsamer. Dieses Verhalten lässt sich durch das Setzen von mehr Keyframes erzeugen, indem man die Methode nach der das Animationstool die Bewegung zwischen zwei Keyframes interpoliert ändert, oder indem man die Animationskurven im Dopesheet Editor oder im Curve Editor per Hand ändert. Der Dopesheet Editor listet alle Keyframes für alle Meshes in allen Dimensionen. Die durch den Keyframe definierten Werte lassen sich hier einfach durch die Eingabe der gewünschten neuen Werte ändern.



Oben: Dopesheet Editor in Blender, unten: Curve Editor in Blender

Der Curve Editor ist eine zweidimensionale Repräsentation der Animation über die Zeit. Abgebildet wird jede Art der Transformation über jede Achse als Kurve. Im Falle einer Translation beschreibt die Kurve beispielsweise die Position eines Objektes in eine Dimension über die Zeit. Durch das Verschieben oder Einfügen neuer Keyframes, das Stauchen oder Strecken der Kurven, ändert lässt sich die Animation im Detail bearbeiten.

Unity selbst enthält auch einen Dopesheet und Curve Editor, solche simplen Animationen lassen sich also auch dort direkt erstellen. Mit welchem Programm man in diesem Fall arbeiten möchte ist letztendlich eine Frage der Präferenz und auch der Vorerfahrung mit diesem oder jenem Programm. Der Satellit im Spiel beispielsweise wurde direkt in Unity animiert, ebenso wie die Genmanipulatoren. Die Animationen aller weiteren Modelle wurden in Blender erstellt.

Eine andere Art Animationen zu erstellen macht sich die integrierte Physik-Engine Blenders zunutze. Blender unterstützt unter Anderem die Berechnung realistischer Verformungen von Objekten bei Krafteinwirkungen, Kraftzonen und Fluidsimulationen.
Im Spiel gibt es eine Holzkiste die bei Berührung mit dem Spieler zerbricht. Die Kiste besteht nicht aus einem zusammenhängenden Mesh sondern aus rund 36 einzelnen Meshes. Diese Animation wurde nicht per Hand über das Setzen von Keyframes erstellt und wird auch nicht in Echtzeit erst in Unity berechnet. Markiert man alle diese einzelnen Meshes als Rigid Body werden sie von der in Blender simulierten Schwerkraft beeinflusst, sobald man die Simulation startet. Das heißt, über einer Plane platziert, die die starr im Raum liegt, fällt die Kiste sobald die Schwerkraft wirkt darauf und danach auseinander. Die einzelnen Meshes beeinflussen sich dabei in ihrer Bewegung realistisch gegenseitig und

bleiben danach auf der Plane liegen. Diesen Vorgang kann man automatisch keyen lassen, das heißt Blender setzt über die gesamte Dauer der Simulation auf jeden Frame einen Keyframe.
Um beim Abspielen der Animation im Spiel Ressourcen zu sparen, ist es klug einige Keyframes wieder manuell zu entfernen. Unity und auch Blender interpoliert für jeden Raum zwischen zwei Keyframes die Änderung der Stausdaten der einzelnen Meshes. Das heißt, entfernt man die Hälfte der Keyframes müssen auch nur rund die Hälfte der Berechnungen angestellt werden.

Nach einem ähnlichen Prinzip wie die simulierte Schwerkraft arbeiten Force Fields. In diesem Fall wird die Kiste allerdings nicht von der Schwerkraft nach unten, sondern von einem Kraftfeld, das in eine bestimmte Richtung zeigt, in diese Richtung gedrängt. Ein Force Field ist eine umgekehrte simulierte Schwerkraft in eine beliebige Richtung.
Platziert man das Feld horizontal entsteht so der Eindruck, die Kiste würde ausgehend von der Richtung in der der Ursprung des Kraftfelds liegt auseinander gesprengt. Auch dieser Vorgang lässt sich automatisch keyen.

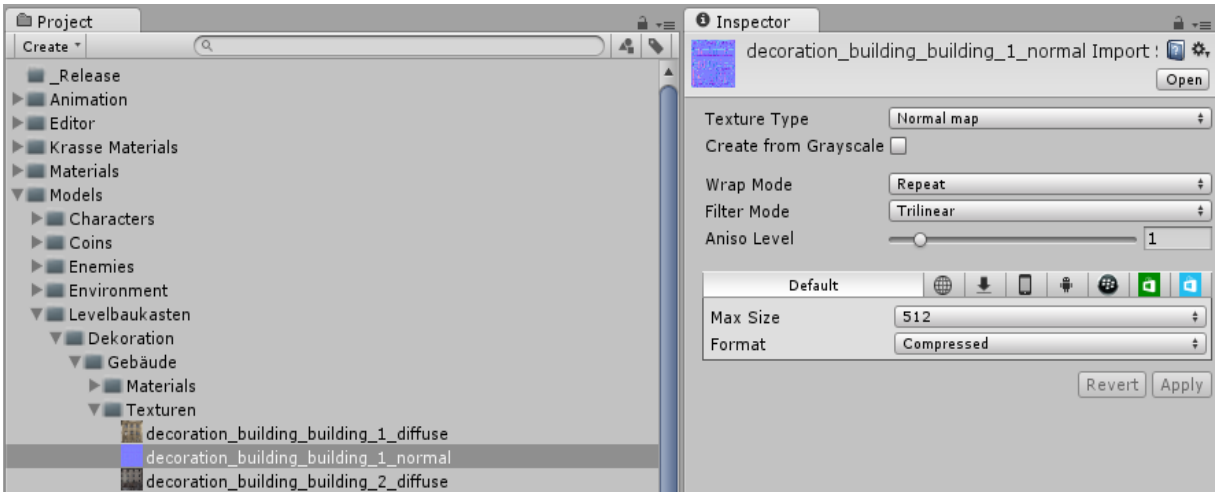
Das fbx-Format speichert die Position der Keyframes und den Zustand des Objektes unter jedem Keyframe mit auf dem Objekt. Unity weiß beim Import des Modells also bereits über die Animation Bescheid und kann sie, wenn vom Nutzer gewünscht, abspielen.

Import nach Unity

Marcel Müller

Ist das Objekt texturiert (und eventuell animiert), folgt das Importieren nach Unity. Praktischerweise importiert Unity Blender-Dateien (.blend) direkt ohne vorige Umwandlung. Im Falle anderer Programme muss man das Objekt beziehungsweise die Szene erst in eine fbx-Datei exportieren. Dabei ist zu beachten, dass das Objekt normalisiert sein muss, das heißt die Skalierung und die Rotation müssen auf 1 und 0 für alle Achsen stehen, da sonst nicht gewollte Verformungen und Verschiebungen in Unity auftreten können.

Der Import nach Unity gestaltet sich recht einfach: Man zieht die gewünschten Dateien (.blend oder .fbx) aus dem Explorer einfach in das Projektpanel in Unity. Das Projektpanel listet alle sich im Projekt befindlichen Dateien und Ordner. Unity kennt nun das Objekt und die zugehörige UV-Map, die auf dem Objekt gespeichert wurde. Es fehlen noch die benötigten Texturen, die ebenfalls per Drag&Drop in das Projektverzeichnis kopiert werden können. Im Inspector lässt sich die Textur als Diffuse-, Normal-, Specular Map markieren. Unity weist den Nutzer allerdings automatisch darauf hin, sollte er versuchen eine nicht als Normal Map gekennzeichnete Textur als solche zu verwenden.



Projekt- und Inspectorpanel bei ausgewählter Textur-Datei in Unity

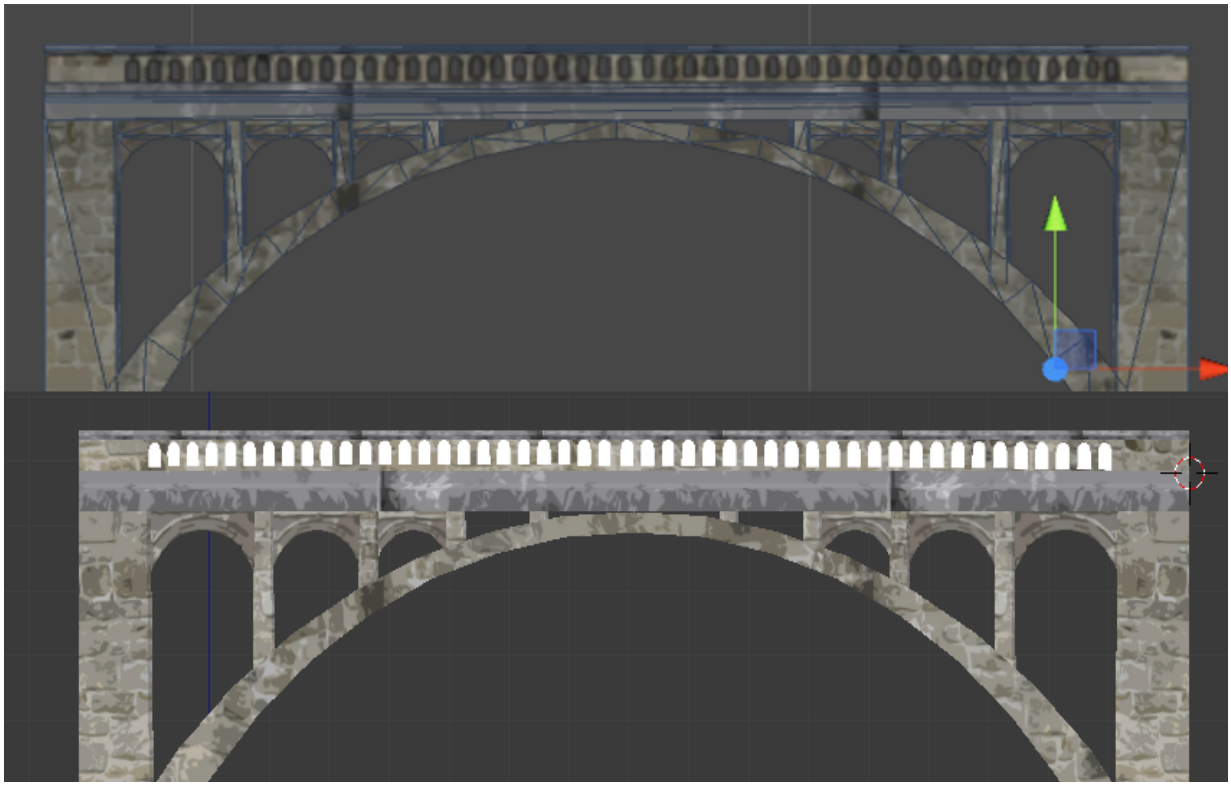
Das Aussehen beziehungsweise die Darstellung eines Materials wird durch einen Shader berechnet, dieser Shader definiert das Material des Objekts. Jeder Oberfläche muss also ein Shader zugeordnet werden und jeder Shader lässt sich mehreren Objekten zuordnen. Wurde einer Oberfläche kein Shader zugeordnet, reagiert sie nicht auf Lichteinfall, das heißt sie erscheint schwarz.
Über ein Drop-Down Menü im Inspector lässt sich der passende Shader für jede Oberfläche auswählen. Anfangs sind nur die in Unity bereits enthaltenden Shader in dieser Liste enthalten. Selbst geschriebene oder per Plugin dazu geladene werden an die Liste angehangen.
Jeder Shader ist in der Art und Anzahl der von ihm benötigten Texturen einzigartig. Der Bumped-Diffuse Shader beispielsweise bietet einen Slot für eine Diffuse Map und einen für eine Bump Map. Die Texturen lassen sich aus dem Projektpanel einfach in diese Slots ziehen. Entsprechend der auf dem Objekt gespeicherten UV-Map projiziert Unity die Textur auf die Geometrie. Die Zuordnung der Shader und Texturen zu einem Objekt wird von Unity gespeichert, das heißt, dass jede weitere Instanz dieses Objektes bereits mit den richtigen Shadern und Texturen versehen ist.

Verhalten ausgelöst durch Scripte werden dem Objekt ähnlich zugeordnet. Aus dem Projektpanel zieht man per Drag&Drop das gewünschte Script in das Inspectorpanel. Erforderliche Parameter des Scripts erscheinen dann unter dem Scripteintrag im Inspector als Eingabefelder. Parameter können wie in den meisten Programmiersprachen numerische oder logische Datentypen sein, aber auch andere Scripte, Spielobjekte oder Partikeleffekte. Erfordert ein Script ein Parameter lässt er sich entweder direkt per Tastatur in das Eingabefeld eingeben oder aus dem Projektpanel hinein ziehen.

Dekoration und Ressourcenoptimierung

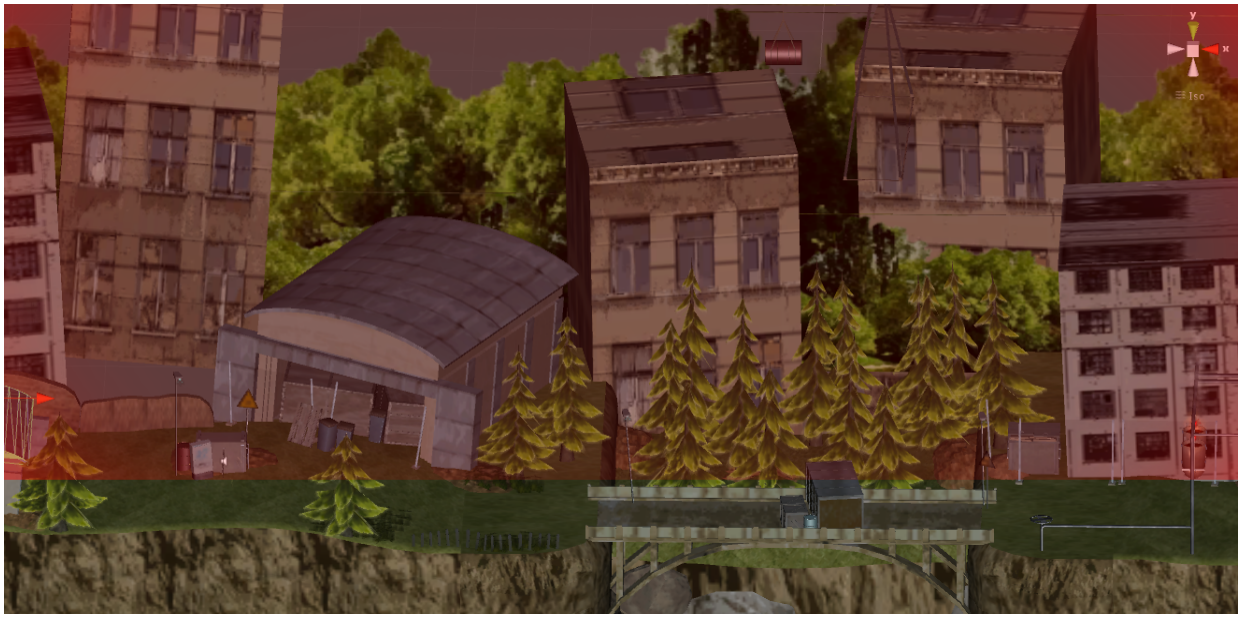
Marcel Müller

Neben der Herausforderung an sich spielt natürlich auch die optische Gestaltung der Levels eine große Rolle, wenn das Spiel dem Spieler gefallen soll.
Möchte man Eindruck einer realitätsnahen Welt erwecken, sind Details in der Levelgestaltung sehr wichtig. Dekorationsobjekte wie Mülleimer, Laub oder verschiedene Arten von Vegetation erinnern an die reale Welt und lassen den Spieler die Spielwelt leichter akzeptieren.
Um den Arbeitsaufwand bei der Erstellung der Modelle zu verringern, macht es Sinn, für die Dekoration die gleichen Objekte zu verwenden, die als Hindernisse im Vordergrund des Levels stehen. Bei der Erstellung und Platzieren solcher Objekte muss natürlich immer die Nützlichkeit und Nötigkeit bedacht werden. Jedes Objekt, das nachher auf dem Bildschirm zu sehen ist muss gerendert werden, verbraucht also Rechenkapazität.
Daher ist es gängige Praktik vom gleichen Objekt mehrere Versionen unterschiedlicher Detailstufen zu erstellen (In manchen Fällen berechnet die Spielengine automatisch niedriger aufgelöste Versionen eines Objektes, je nach seinem Abstand zur Kamera). Die Gebäude die in manchen Modulen im Hintergrund zu sehen sind, bestehen beispielsweise nur aus einem Würfel, das Gras besteht aus einer einzigen Plane mit Textur. Erstellt man Objekte mit Löchern, etwa einen Fensterrahmen, kann es nach Art des Modells mehr Sinn machen, eine Plane mit einer darauf liegenden Alpha-Textur zu verwenden, statt die Löcher im Mesh auszumodellieren, da für jedes zusätzliche Polygon Berechnungen angestellt werden müssen.



Brücke, die eine Alphatextur für das Geländer verwendet. Oben: Unity, unten: Blender

Auch die Texturen spielen eine große Rolle bei der Einsparung von Rechenzeit aber auch Speicherplatz. Je größer und höher aufgelöst die Texturen sind, desto mehr Speicher verbrauchen sie, desto länger braucht es sie bei Spielstart zu laden und desto länger dauert das Rendern jedes Frames.
Grundsätzlich sollte die Textur ungefähr genau so groß sein wie die UV-Map des Objektes auf das sie gemappt wird. Die Auflösung sollte je nach Größe des Modells gewählt werden. Eine Textur mit einer Full-HD Auflösung auf einem Objekt, das im Spiel nicht größer als ein paar Millimeter sein wird, hat keinen Nutzen oder optischen Mehrwert und sollte vermieden werden.
Um Speicherplatz zu sparen können Texturen (sowie alle anderen Ressourcen wie Scripts oder Partikeleffekte) natürlich auch von mehreren verschiedenen Objekten verwendet werden, insbesondere wenn die beabsichtigte optische Wirkung ähnlich ist.

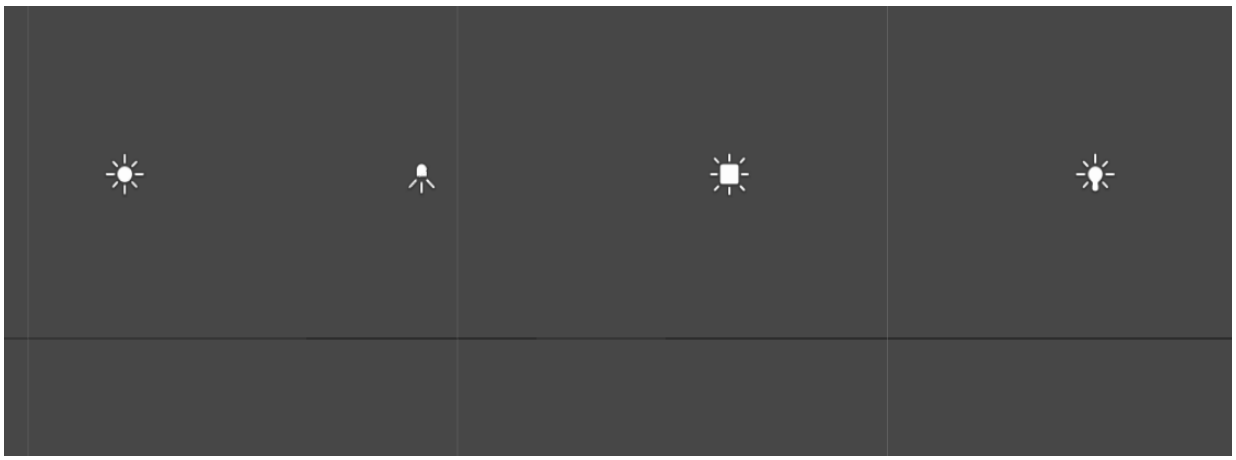


Aufteilung des Levels in die Tiefe. Roter Bereich: Hintergrund, vorderer Bereich: Laufzone

Damit der Spieler die hinteren Kanten der Bodenmodelle nicht sehen kann, müssen sie durch andere Objekte versteckt werden. Dazu bieten sich große Modelle wie Gebäude an, lassen den Hintergrund aber auch eintönig erscheinen.
Der Fokus muss also durch eine unauffällige Gestaltung des Hintergrunds, Beleuchtung oder Partikeleffekte im Vordergrund vom hinteren Teil des Levels genommen werden. Ist der Spieler auf das Geschehen im vorderen Teil des Levels konzentriert, reicht auch ein halbdurchsichtiger Zaun, um die gerade Kante des Bodens zu kaschieren.
Hilfreich ist es auch, wenn der hintere Teil der Laufebene eine ähnliche Farbe wie der Hintergrund dahinter hat, wenn der Übergang also nicht leicht zu erkennen ist. Im Spiel befinden sich hinter den letzten Objekten in den Modulen daher meist noch senkrecht aufgestellte Ebenen mit einer undeutlichen Waldtextur. Hält man die hinteren Kanten der Laufebenen im Schatten, verschwimmt der Übergang zwischen Level beziehungsweise Laufebene und Hintergrund. Zusätzlich erzeugen diese Hintergrundplanes durch ihren Abstand zum eigentlichen Level und dem dadurch versetzten Scrollen bei Bewegung der Kamera eine gewisse Tiefe.

Weitere Werkzeuge zur optischen Gestaltung sind Lichter und Farben. Es ist möglich dem kompletten Sektor ein globales Licht oder eine globale Lichtstimmung zu übergeben. Diese Optionen lassen sich als Parameter im Sector Script (dazu später mehr) verändern.
Eine dem Sektor zugewiesene Color Correction manipuliert die Anzeige der Farben auf dem Bildschirm. Im Grunde gleicht diese Funktion einfachen Operationen auf einem 2D-Bild, wie das Verändern des Kontrasten, der Farbwärme oder der Hervorhebung bestimmter Farben.

Nützlich ist das Manipulieren der Farben wenn man darauf abzielt eine bestimmte Stimmung zu erzeugen. Im Halloween-Sektor zum Beispiel wird das Bild allgemein verdunkelt und Farben wie rot oder orange hervorgehoben, um eine gruselige Atmosphäre aufzubauen.



Die verschiedenen Lichtarten in Unity

Lichter oder Lichtquellen existieren in Unity in unterschiedlichen Formen. Directional Lights senden parallele Lichtstrahlen von einer unendlich großen Ebene. Point Lights strahlen von einem Punkt aus in alle Richtungen und Spot Light senden Licht von einem Punkt in eine bestimmte Richtung. Neben den spezifischen Einstellungen für jede Lichtart kann die Farbe, Stärke und Distanz jedes Lichts geändert werden. Während Lichter in der realen Welt immer einen Schattenwurf verursachen, wenn sie von einem Objekt aufgehalten werden, würde dieses Verhalten den Computer schnell an die Grenzen seiner Rechenkraft bringen, sollte es für alle Lichter aktiviert werden. Man muss daher abwägen, welchen optischen Effekt der Schatten haben würde. Er kann zum Beispiel helfen kleinere Fehler in der Modellierung oder Texturierung zu verstecken oder den Fokus durch Überbeleuchtung bestimmter Stellen von nicht wichtigen Stellen im Level zu nehmen, etwa der hinteren Kante der Laufebene.
In manchen Spielen wird Licht verwendet um bestimmte Objekte hervorzuheben oder es ist direkt in das Gameplay integriert. Da Lichter in unserem Spiel aber ausschließlich zu Dekorationszwecken oder als Hervorhebung eines Partikeleffektes vorhanden sind und das globale Licht in den Sektoren den Effekt von Schatten zusätzlich verringern würde, werfen nur sehr wenige Lichter in den Levels Schatten.

In Unity ist außerdem noch darauf zu achten, jedes Licht als “Important” zu markieren, da es in manchen Fällen sonst nicht weiter berechnet wird, wenn es sich zu nah an den Bildschirmrand bewegt, um Ressourcen zu sparen. Wenn sich

die Kamera also von links nach rechts durch das Level bewegt, entsteht der Eindruck, die Lichter würden flackern.
Lichter einer Szene, die in Blender erstellt wurden, werden beim Import der fbx-Datei nicht von Unity erkannt.

Modulerzeugung

Marcel Müller

Alle Module werden händisch aus den verschiedenen erstellten Umgebungsobjekten in Unity zusammen gebaut. Die Basis bilden dabei immer die Modelle, die den Boden ausmachen. Der Boden definiert die Laufhöhe und -position der Spielercharaktere. Alle weiteren Objekte werden um diese Modelle oder darauf platziert.

Um mehrere Module bündig aneinander reihen zu können, muss die Bodengeometrie am Anfang und am Ende deckungsgleich sein und sich für jede Ebene immer auf der gleichen Höhe befinden. Alle übrigen Dekorations- und Hindernisobjekte, Lichter und Partikeleffekte lassen sich auf den Böden oder frei im Level platzieren.
Soll der Spieler mit den Objekten kollidieren können, muss darauf geachtet werden ihnen einen Collider hinzuzufügen. Am ressourcenschonendsten ist ein sogenannter Boxcollider der einen Würfel um das Objekt aufspannt.
Ist das Modell zu komplex für einen Boxcollider oder kommt es auf die genaue Form des Modells an, verwendet man einen Meshcollider. Dieser hat die gleiche Geometrie wie das Objekt selbst und legt sich perfekt darüber. Als dritte Art bietet Unity den Spherecollider, der gleich dem Boxcollider auch vom Objekt versetzt im Raum platziert werden kann, statt eines Würfels aber eine Kugel oder ein Ei aufspannt. Ressourcentechnisch reiht er sich zwischen dem Box- und dem Meshcollider ein.
Da Unity für jeden Collider und Frame berechnet ob sich ein anderes Objekt in diesem Collider befindet, ihn betritt oder verlässt, sollte darauf verzichtet werden Objekten im Hintergrund, mit denen der Spielercharakter nicht in Berührung kommt, einen Collider zuzuweisen.
Collider fügt man wie jede andere Komponente eines Objektes hinzu, indem man bei ausgewähltem Objekt den Collider entweder direkt aus dem Projektpanel in den Inspector zieht oder ihn dem Objekt über die “Add Component”-Schaltfläche im Inspector zuweist.

Weiter gilt es das Modul in eine Luft-, Erd- und Wasserzone aufzuteilen. Diese Zonen, die aus einem mit einem Script versehenen leeren Objekt mit Collider bestehen, sorgen dafür, dass ein zonenfremder Charakter sofort stirbt, sobald er die Zone betritt. Dies ist zum Beispiel an Stellen nötig, an denen die Bodengeometrie nicht geschlossen ist, etwa unter Brücken. Da der Bereich in dem

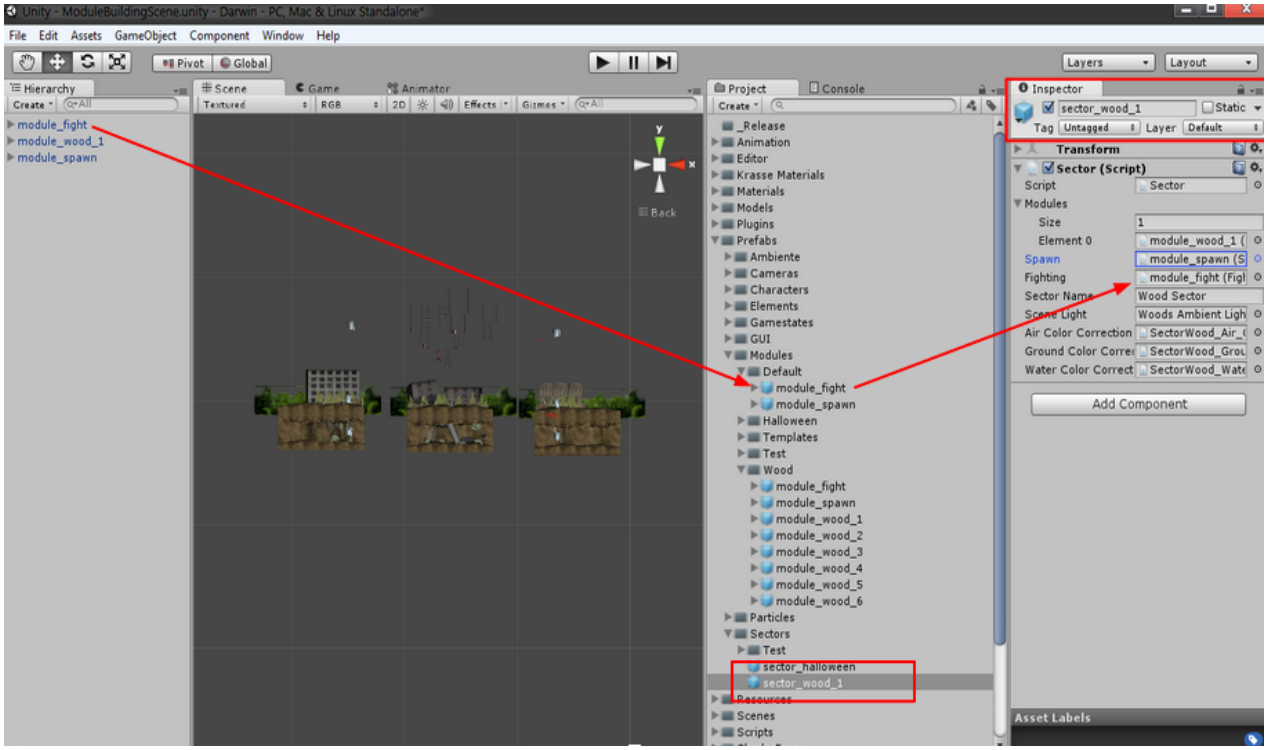
sich die Kamera nach oben und unten bewegen kann durch Kontrollpunkte beschränkt ist, würde sie dem Charakter nicht folgen, sollte er in eine andere Ebene fallen und nicht sterben.
Die Collider dieser Objekte werden über die gesamte zugehörige Ebene des Moduls skaliert.

Anhand eines Algorithmus werden die Module im Spiel zufällig aneinander gereiht. Als Verbindungspunkte benutzt er dafür leere Spielobjekte, sogenannte Empties, die jeweils am Anfang und Ende eines jeden Moduls immer an der gleichen Stelle platziert sind. Diesen Elementen muss das Connector-Script zugewiesen werden.

Weitere leere Objekte entlang des Levels bilden den Pfad dem die Kamera der zugehörigen Ebene später folgt. Für jede Ebene muss ein eigener Pfad definiert werden.
Den einzelnen Kontrollpunkten des Pfades muss das Control Point Element Script zugewiesen werden. Markiert man den “Has Score” Parameter in diesem Script als “true” erscheint auf der Position der Empties bei Spielstart eine Münze, die der Spieler einsammeln kann. Außerdem erlaubt es das Script einen Wert festzulegen, den sich die Kamera höchstens nach oben oder unten von der Position des Empties bewegen kann. So stellt man sicher, dass die Kamera nicht in die anderen Ebenen schauen kann. Die einzelnen Empties muss man unter einem weiteren Empty zusammen fassen, dem man das Path Script zuweist. Die Kontrollpunkte übergibt man diesem Script als Parameter.
Objekte gruppiert man unter anderen Objekten indem man sie im Hierarchypanel einfach per Drag&Drop auf das gewünschte Parent-Objekt zieht.

Im Spawnmodul muss außerdem noch ein Spawnpunkt, beziehungsweise ein Empty mit angehangenem Spawn Point Script, vorhanden sein, im Fightmodul entsprechend ein Despawn Punkt.

Nachdem man alle Objekte des Levels unter einem weiteren Empty, also einem obersten Node, zusammen gefasst hat und diesem Empty das Module Script zugewiesen hat, muss man dem Script die beiden Connectoren und die drei Pfade als Parameter übergeben. Im Falle des Spawn- und des Fightmoduls fallen das Spawnelement und das Despawnelement als weitere Parameter an. Das jetzt fertige Level lässt sich nun als Prefab speichern.



Erzeugung eines Sektors in Unity. Das Fightmodul wird dem gewählten Sektor als Parameter übergeben

Um mehrere zusammen gehörende Module einem Sektor zuzuordnen erstellt man zuerst ein Prefab im Projektpanel an der gewünschten Stelle.
Ein Prefab ist ein Container für andere Objekte, Partikel oder Scripts. Ordnet man zum Beispiel ein Modell mit Script einem Prefab unter, so bleibt es bestehen, selbst wenn das Ursprungsmodell gelöscht oder geändert werden sollte.
Diesem Prefab weist man das Sector Script zu und übergibt ihm die gebauten Module als Parameter.

Musik und SFX

Titelmusik und Kreaturengeräusche

Christoph Duda

Die Titelmusik zu "Crazy Doc's Evolution Run" entstand mithilfe von "FL Studio", einer Software für die Erstellung und Bearbeitung von Musik. Der Song wurde darauf ausgelegt als Schleife so lange wie nötig abgespielt zu werden. Mein Ziel war es das wilde, animalische mit futuristischen Sci-Fi Klängen zu vereinen und alles in einem schnellen Rhythmus zu einer Einprägsamen Melodie zu führen. Eingeleitet wird das Stück von einem pulsierenden Signal, welches zukünftig als Erkennungssound für den Genmanipulator genutzt werden kann. Dann folgt der Rhythmus von Trommeln und Rasseln im afrikanischen Stil und futuristischen Soundeffekten. Die Melodie wird von Geigen gespielt, sie sollen dem ganzen Tempo geben, Stress aufkommen lassen und Lust auf das Wettrennen machen. Die Geigen sollen dem Thema etwas gehobenes verleihen, leicht episch wirken und eine Brücke zwischen den urzeitlichen Trieben, welche die Mischkreaturen darstellen und der futuristischen Wissenschaft, welche vom Professor verkörpert wird, schlagen.

Die Geräusche der Charaktere, wenn sie Schaden erleiden, wurden ebenfalls in FL Studio aufgezeichnet. Dafür wurde das “Samson G Track” USB Studiomikrofon genutzt, um möglichst gute Ergebnisse zu erzielen. Es wurden mehrere Sequenzen, mit diversen Ausrufen, passend zu den Charakteren hintereinander aufgezeichnet. Anschließend wurden die besten Sounds ausgewählt und zurechtgeschnitten. Um den Aufnahmen tiefe zu verleihen wurden sie noch mit subtilen Effekten versehen.

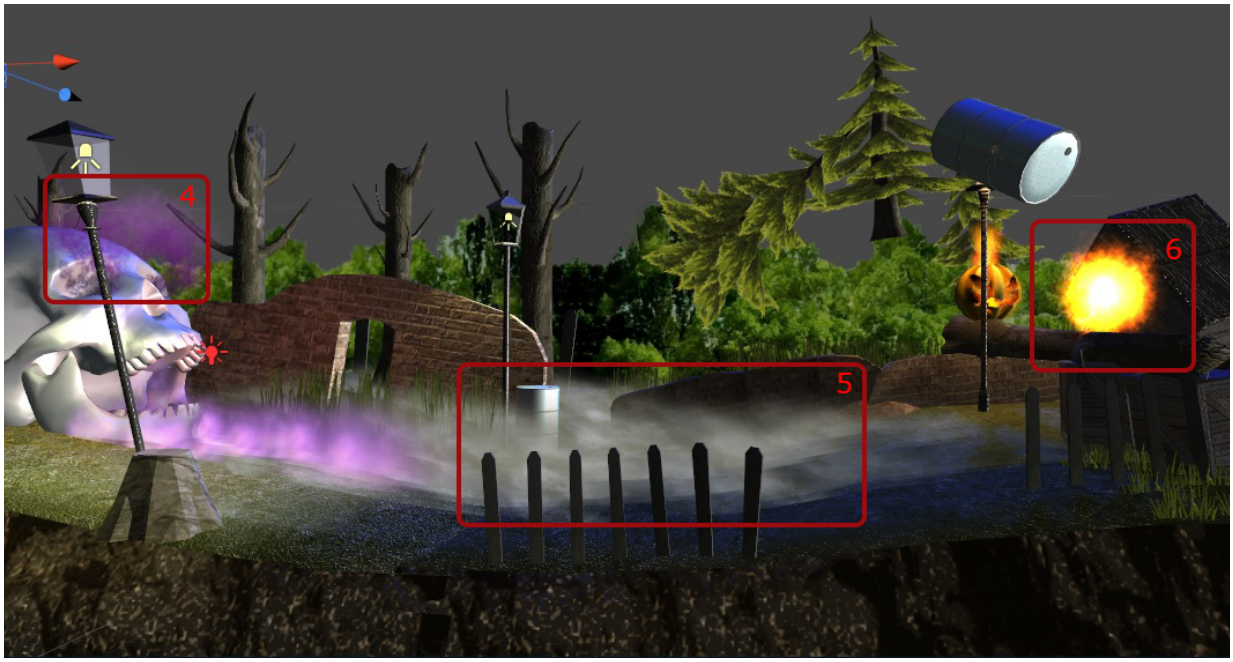
Shuriken Particle System

Christoph Duda

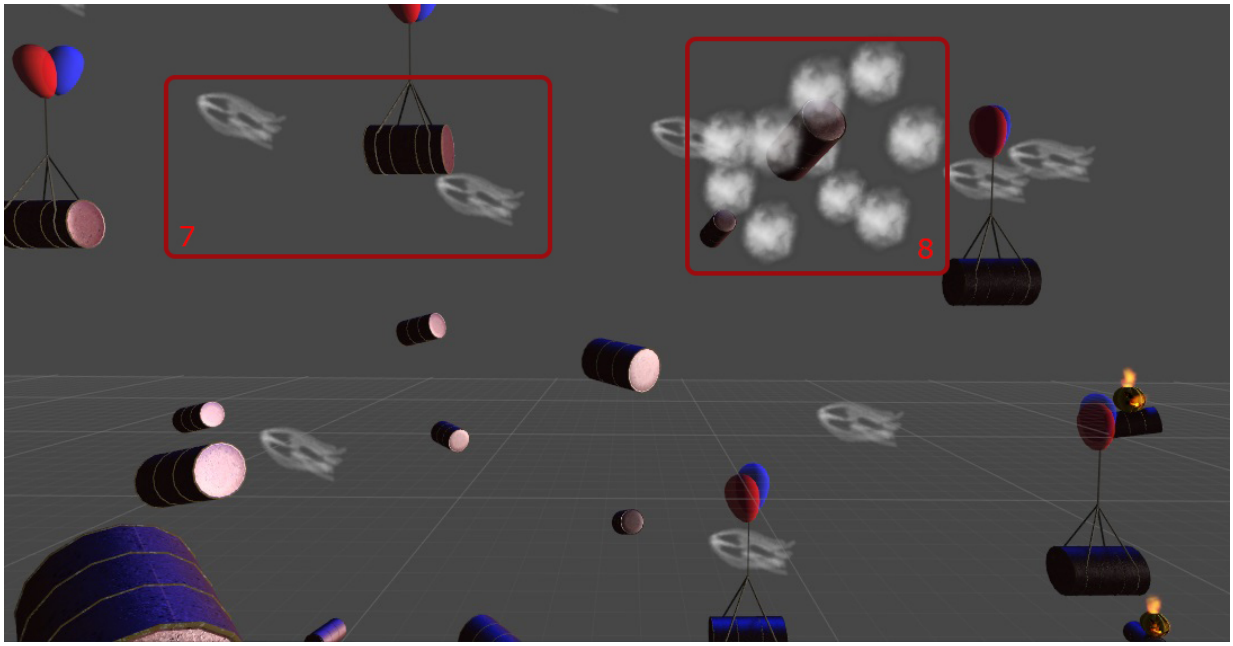
Eine weitere Tätigkeit in diesem Projekt umfasste die Partikeleffekte für das Spiel. Für die Umsetzung der Effekte habe ich das eingebaute “Shuriken Particle System” genutzt. Alternativ gibt es einen “Ellipsoid Particle Emitter” aus früheren Unity Versionen. Shuriken ist jedoch einfacher in der Handhabung und bietet mehr Einstellmöglichkeiten. Partikeleffekte werden in allen Bereichen genutzt. In unserem Spiel kommen sie zum Beispiel als Wolken am im Hintergrund vor, simulieren fließendes Wasser oder Wasserfälle, erschaffen atmosphärischen Rauch oder Nebel, wie im Halloween-Sektor zu sehen ist und tauchen auch als Feedbackeffekte auf, wie beim Einsammeln von DNA-Coins. Der Shuriken-Partikeleditor bietet eine Vielzahl von Einstellungsmöglichkeiten um jeden Effekt zu erzielen, den man benötigt. Dabei spielt die Textur eine große Rolle, denn diese gibt das Aussehen eines einzelnen Partikels vor. Anschließend erstellt man einen Effekt, in dem die Lebenszeit der Partikel, die

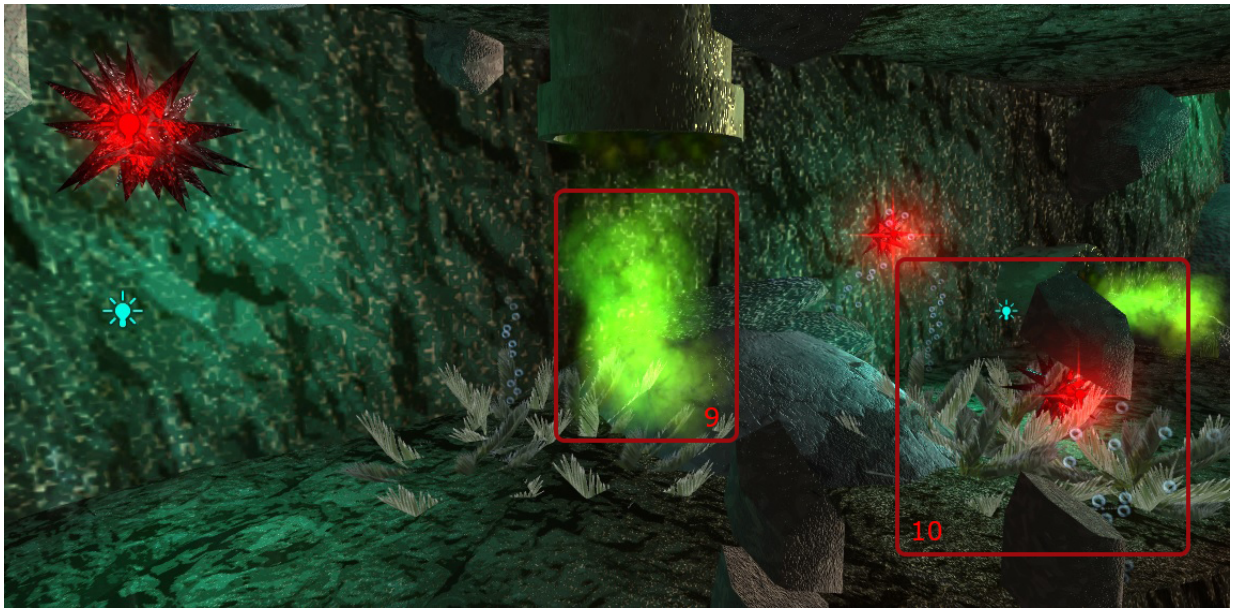
Emission, also die Anzahl der generierten Partikel pro Sekunde, die Form des Partikelemitters, die Farbe der Partikel nach Lebenszeit, deren Geschwindigkeit und eine Vielzahl weiterer Optionen festlegt. Dies soll kein Tutorial zum Partikeleditor werden, das würde den Rahmen sprengen, aber anbei zeige ich eine Auswahl meiner erstellten Partikeleffekte, um die Vielfältigkeit dieses kleinen Tools zu zeigen.

1. Wolken
   Die Wolken sind ganz simple, sich langsam bewegende Partikel.
2. Mystischer Nebel des Genmanipulators
   Hier wirken zusätzliche physikalische Kräfte auf die Partikel (Gravitation und Reibung)
3. Wasserfall
   Frühe Version, da nicht mehr im Spiel zu sehen



4. Lila Dampf
5. Nebelschwaden
6. Feuerbälle und Geister (7)
   Diese Partikel werden als Spielelemente genutzt und verursachen Schaden, wenn sie den Spieler treffen. Ein weiterer Effekt deutet auf den Treffer hin (8)





9. Giftige Wolken
   Machen ebenfalls Schaden am Spieler
10. Wasserblasen
    Treiben nach oben durch inverse Gravitation und bleiben an Steinen haften

Wasser Effekte in Unity

Daniel Glebinski

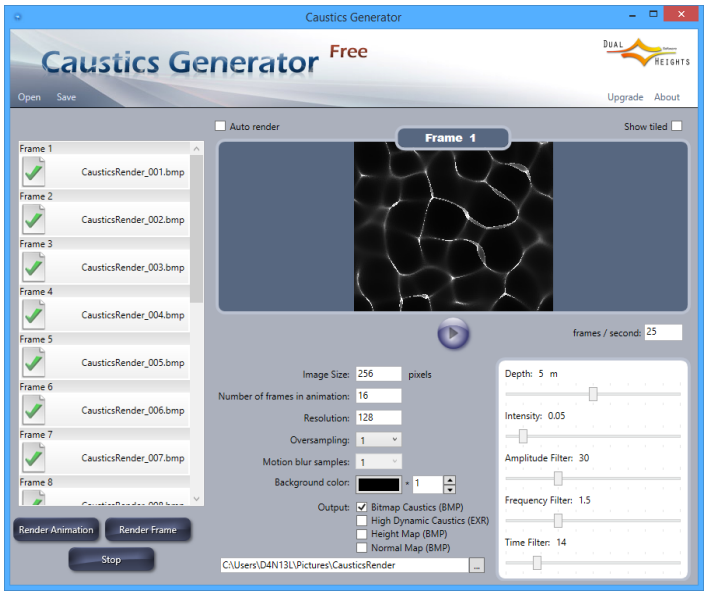
Internet Recherche:
Um das Wasser im Unterwasserlevel authentischer zu gestalten, habe ich eine Reihe von Effekten in Unity zur Anwendung gebracht:

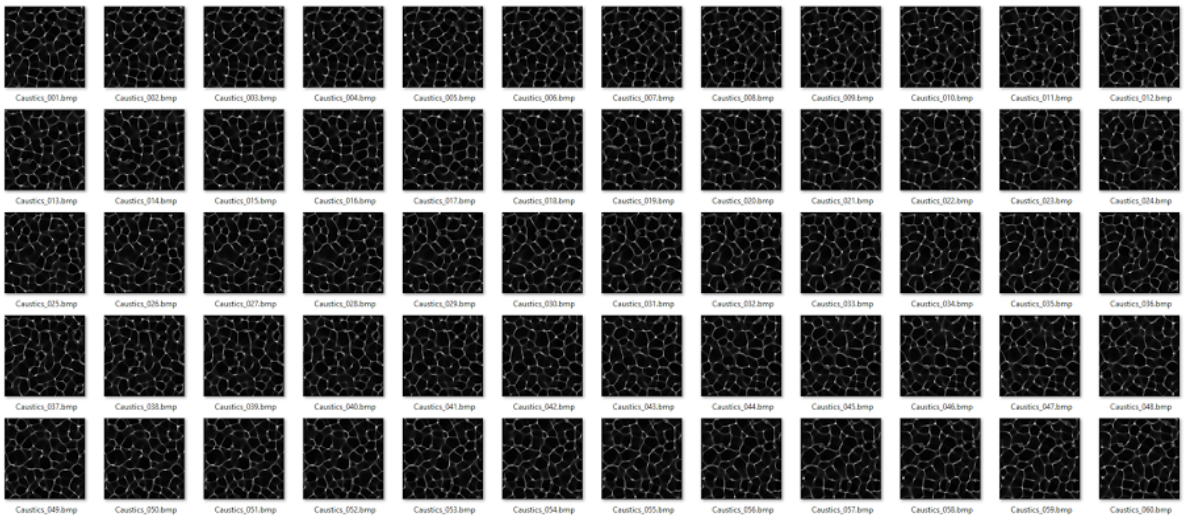
- Farbe: Mittels Color Correction Curves wurde die Globale Farbgebung angepasst
- Trübung: durch Global Fog wurde eine Trübung bzw. ein leichter Nebel hinzugefügt
- Reflektionen für die Unteseite des Wassers wurden mit einem Extra Script ermöglicht
- Wasseroberflächenreflektion: hierfür wurde Unity Water verwendet (Standard Asset)
- Caustics: Die typischen Lichtbrechungen wurden durch dieProjektion einer Caustic-Animation auf das gesamte Wasserlevel erzeugt
- Sun Shafts (Sonnenstrahlen)

Blur VS. Normalmap Distortion VS. Depth Of Field

Um den Effekt der leicht verschwommenen Optik und der vernebelten Farbverläufe zu erzielen, habe ich mehrere Möglichkeiten getestet:
Blur erzeugte ein einheitliches Weichzeichnen, das optisch nicht überzeugte.
Normalmap Distortion fügte animierte Wasserverwirbelungen hinzu, die zu einer verbesserten Optik beitrugen. Depth Of Field wiederum erzeugte eine gut einstellbare

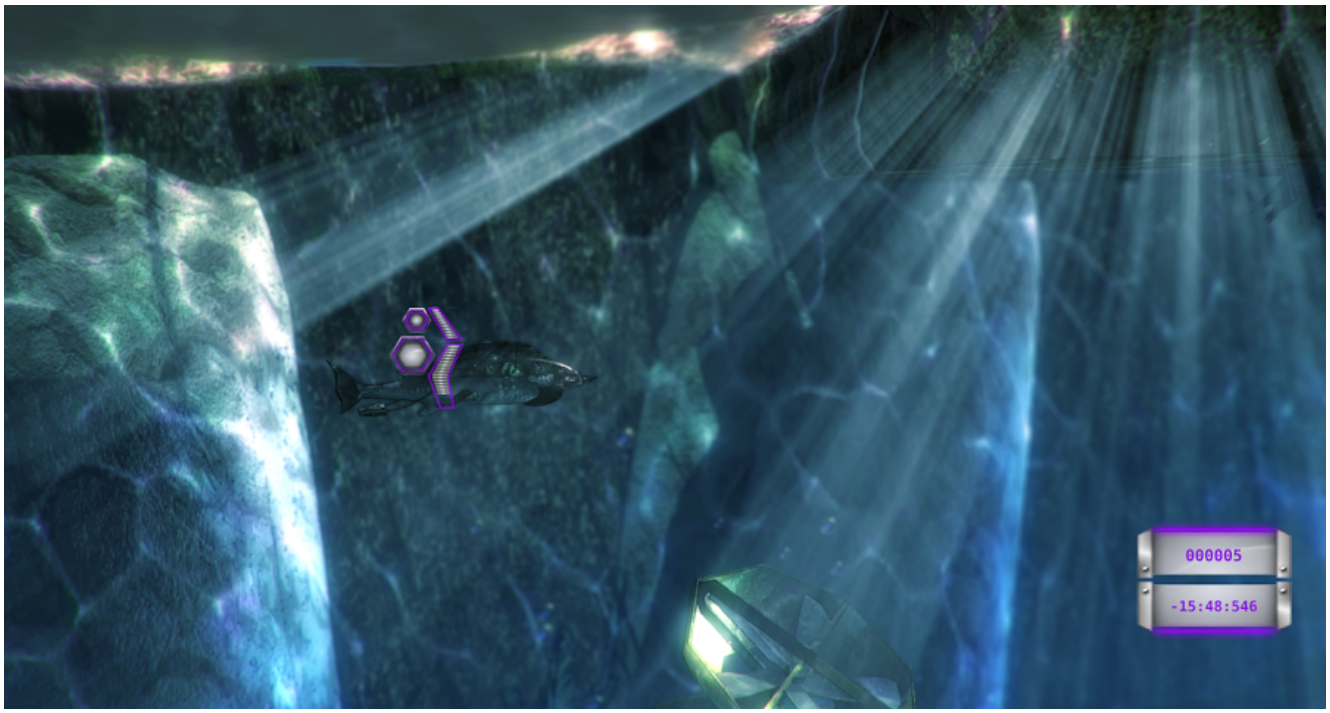
Tiefenunschärfe mit Fokus auf dem Player, allerdings war Normalmap Distortion mit Depth Of Field nicht kompatibel, denn die Tiefeninformation auf der Plane mit Normalmap vor der Kamera überschreiben die Tiefeninformation der sich dahinter befindlichen Objekte. Es kam also wieder zu einem globalen Blur.
Ein gutes Ergebnis wurde also mit Unity Water, Global Fog, Caustic Projektor, Collor Correction Curve und den Sun Shafts erzielt.
Projektion einer Caustic-Animation auf das gesamte Wasserlevel
Caustic kann mit prozeduralen Texturen gerendert werden. Dafür habe ich die Software Caustic Generator verwendet. Dazu werden die Kameras in Unity als Projektoren verwendet und projizieren die vorher erstellte Caustic-Animation, die aus 60 Frames besteht, auf die Oberfläche des Levels wie einen Film auf eine Leinwand.





Ohne Wassereffekte

Mit Wassereffekten

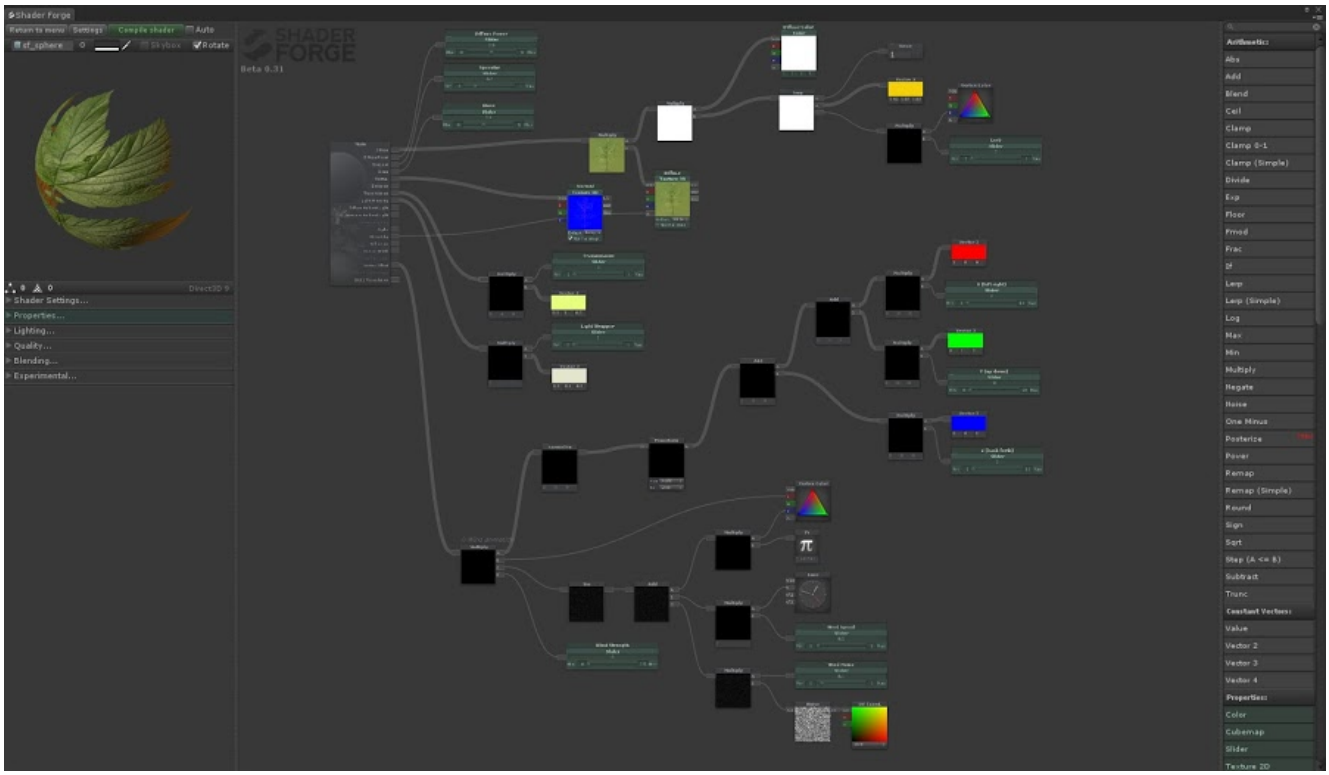


Teile der Wassereffekte konnten auch auf Air und Ground übertragen werden: dazu gehören Color Correction Curves, Depth Of Field (in abgeschwächter Form), Sun Shafts, Bloom sowie Vignette.

Shader Forge

Daniel Glebinski

Für die Gestaltung der Bäume und Blätter im Ground-Level wurde ein Blatt-Shader benötigt, der für eine natürliche Bewegung und Schattierung der Blätter sorgt. Üblicherweise müssen Shader für Unity mühsam Zeile für Zeile gecoded werden. Das Testen des Shaders ist sehr schlecht möglich da man erst nach dem fertigem Compilieren und Anwenden des Shaders auf ein Obejkt sehen kann was man da eigentlich fabriziert hat. Man kann aber auch ein Plugin verwenden:
Shader Forge (ein Node Based Shader Editor für Unity).
Shader Forge ermöglicht es eine Shader aus Node basierten Elementen zusammen zu fügen und liefert ein direkte Vorschau durch automatisches compilieren nach Änderung der verschiedensten Parameter.



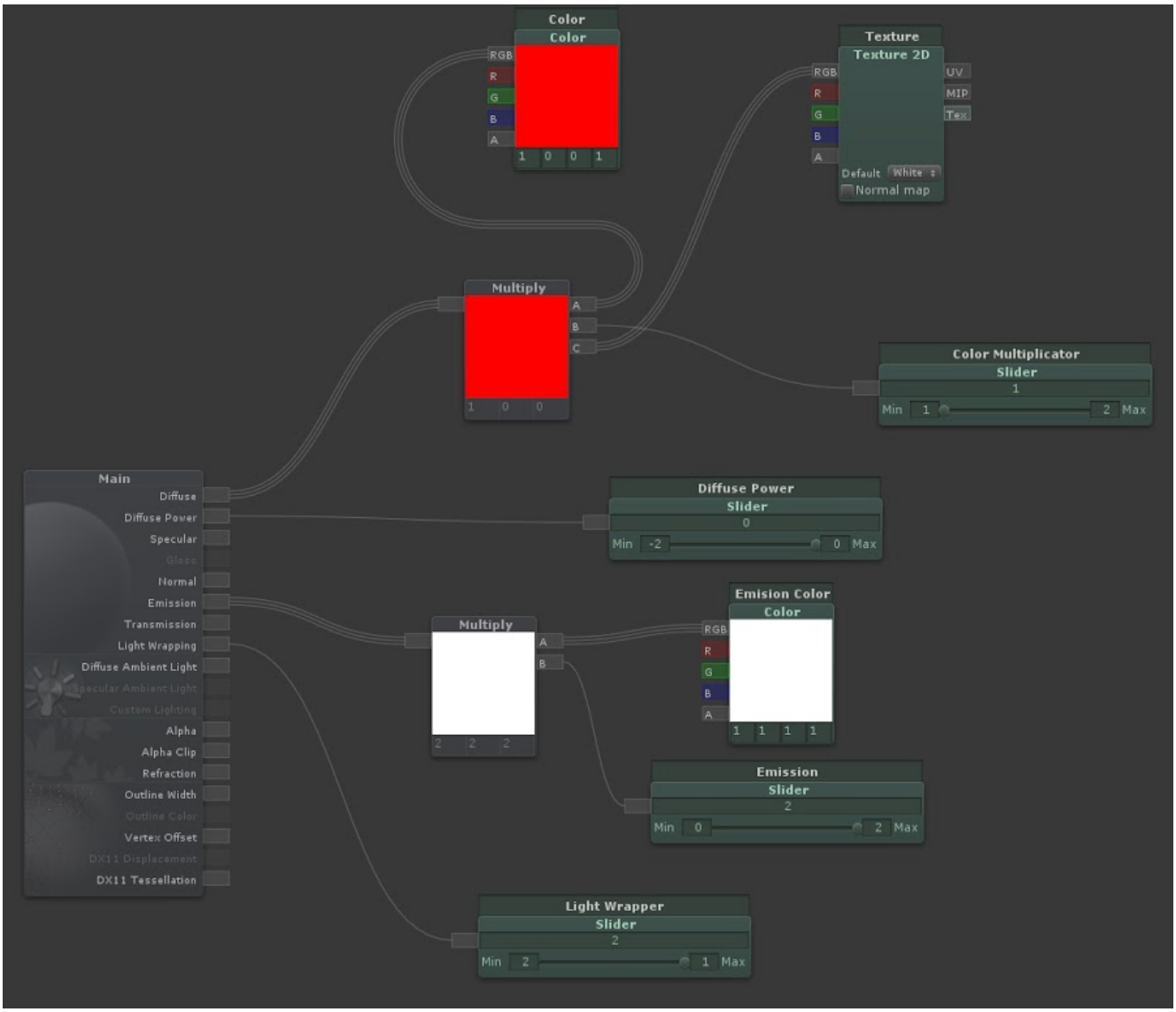
Mit diesem Plugin wurde der Leaf Shader für die Blätter durch die Manipulation folgender Parameter erweitert:

Specularity	Wind Noise
Gloss	Wind Direction XYZ
Diffuse Power	Light Wrapper
Wind Strength	Transmission
Wind Speed	Lerb

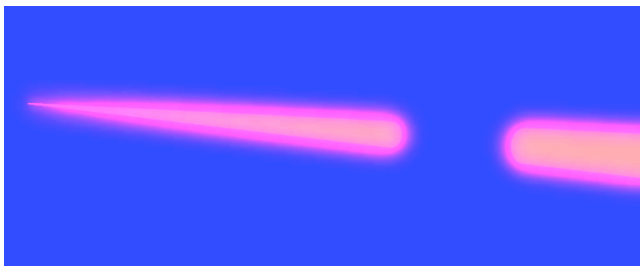
Das Plugin sorgt dafür, dass man die verschiedenen Operatoren in einer Hierarchie anordnen kann und so z.B. Parameter wie Windrichtung etc. für das Verhalten der Blätter berücksichtigen kann. Dieser Shader enthielt mehr als 400 Zeilen Code, die sonst manuell erstellt werden müssten.
Ein weiterer Shader wurde für einen Laser benötigt, der von einem tödlichen Hindernis ausgestoßen wird. Das Leuchten kann dadurch erreicht werden, dass die Farbwerte übersteuert werden und an der Unity Kamera ein Bloom Script aktiv ist. Da es verschiedene Möglichkeiten gibt Farben zu übersteuern und sie auch unterschiedliche Ergebnisse liefern, habe ich sie alle in den von mir entwickelten Laser Shader implementiert. Dazu gehören folgende Parameter, auf die man Einfluss haben kann:

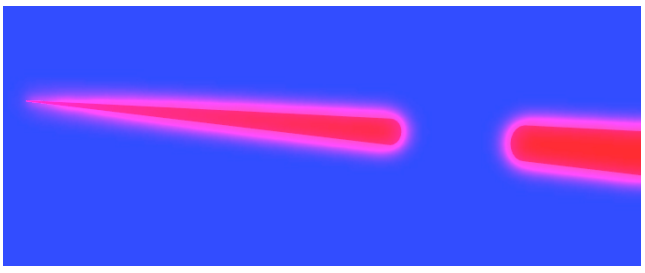
Color	Hier kann die Farbe eingestellt werden
Texture	Hiermit kann bestimmt werden, wo das Objekt leuchten soll und wo nicht
Color Multiplicator	Stufenlos einstellbar bis zu Verdopplung der Farbwerte (RGB)
Light Wrapper	Eine weitere Möglichkeit um zu übersteuern
Emission Color	Hier kann die Farbe des Laserkerns beeinflusst werden
Emission	Bestimmt die Emissionstärke

Laser Shader Aufbau

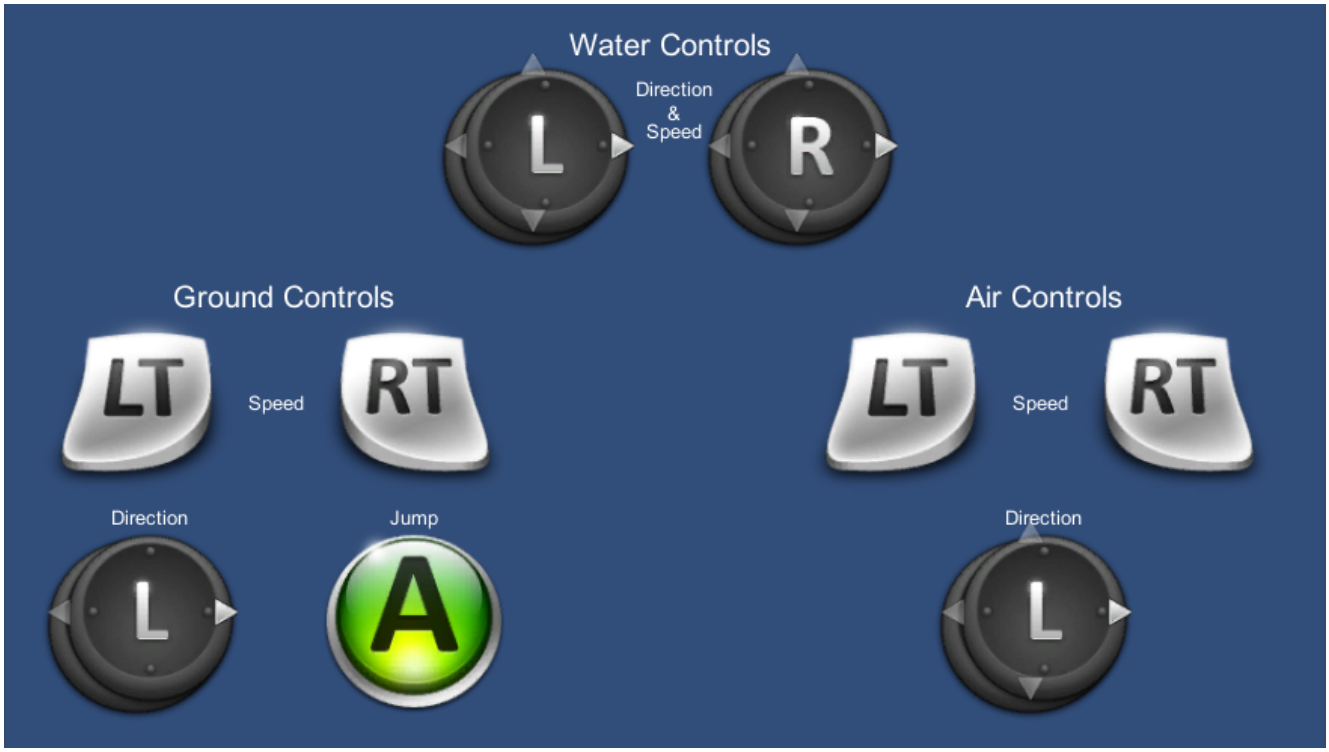


Laser Shader Resultate

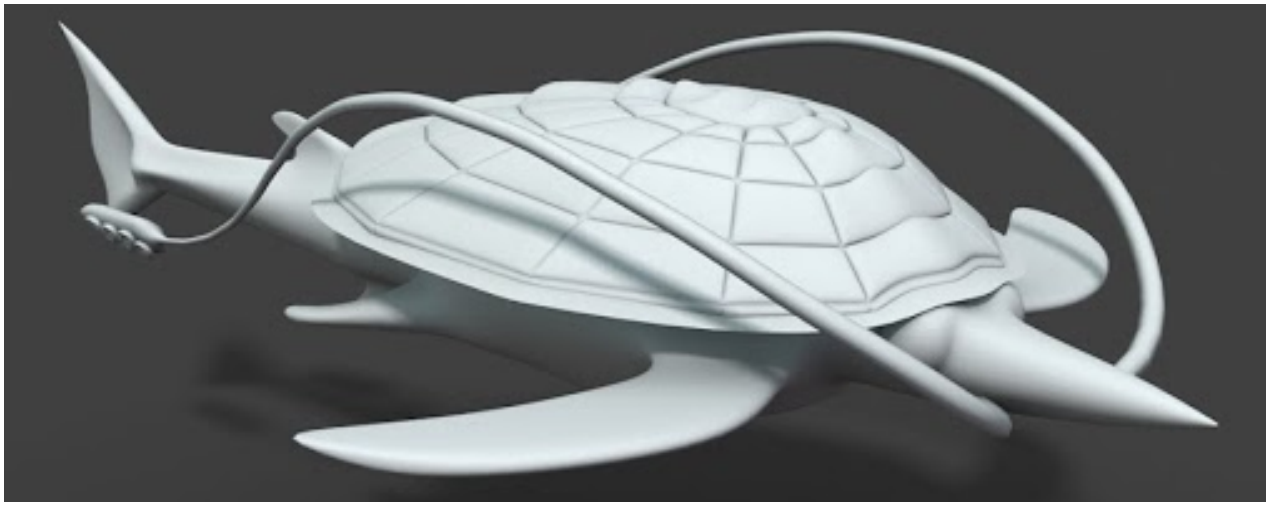


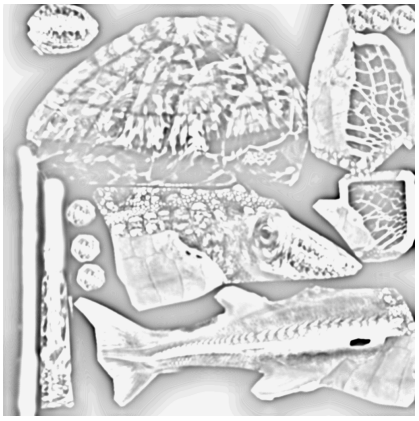


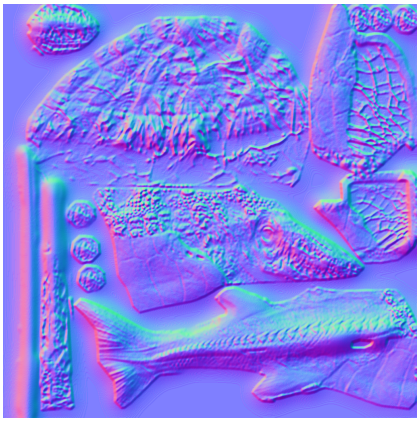
Animierte Steuer-Elemente



Squrtle-gator Model und Texturen







Weitere Arbeiten

Christoph Duda

Neben den bereits genannten, fielen immer mal wieder kleinere Aufgaben an wie das Erstellen des Splashscreens oder das Testen von Spielpassagen, Animationen und Effekten. Unter anderem war ich auch für die 3D-Umgebung und Kameraperspektiven im Hauptmenü zuständig. Auch kleinere Skripte, wie das "schweben" der Bäume und Fässer, während meiner Arbeit am Halloween Sektor, habe ich verfasst. Diese kleineren Aufgaben brachten viel willkommene Abwechslung in die Entwicklung, da man sonst nur repetitive Arbeitsabläufe hätte. Stattdessen konnte man sich in neue Problemstellungen hineindenken und musste Lösungen auf unterschiedlichen Ebenen finden und ebenfalls unterschiedlich umsetzen.
Auch wenn diese kleineren Aufgaben kaum den Umfang einer Charaktererstellung erreichen, so verschlingen diese dennoch mehr Arbeitsstunden als oft gedacht, tragen aber auch viel zum Endprodukt bei.

Daniel Glebinski

Darüber hinaus habe ich bei Bedarf bei Problemen anderer Teammitglieder ausgeholfen wie z.B. dem Einfügen einer kleinen Animation zur Erklärung der Spielsteuerung für jedes Level, bei der die Bewegung der verschieden Buttons animiert ist.
Meine Aufgabe war es auch, verschiedene Soundeffekte für die verschiedenen Aktionen zusammenzustellen. Diese konnten im fertigen Spiel aber bisher noch keine Anwendung finden.

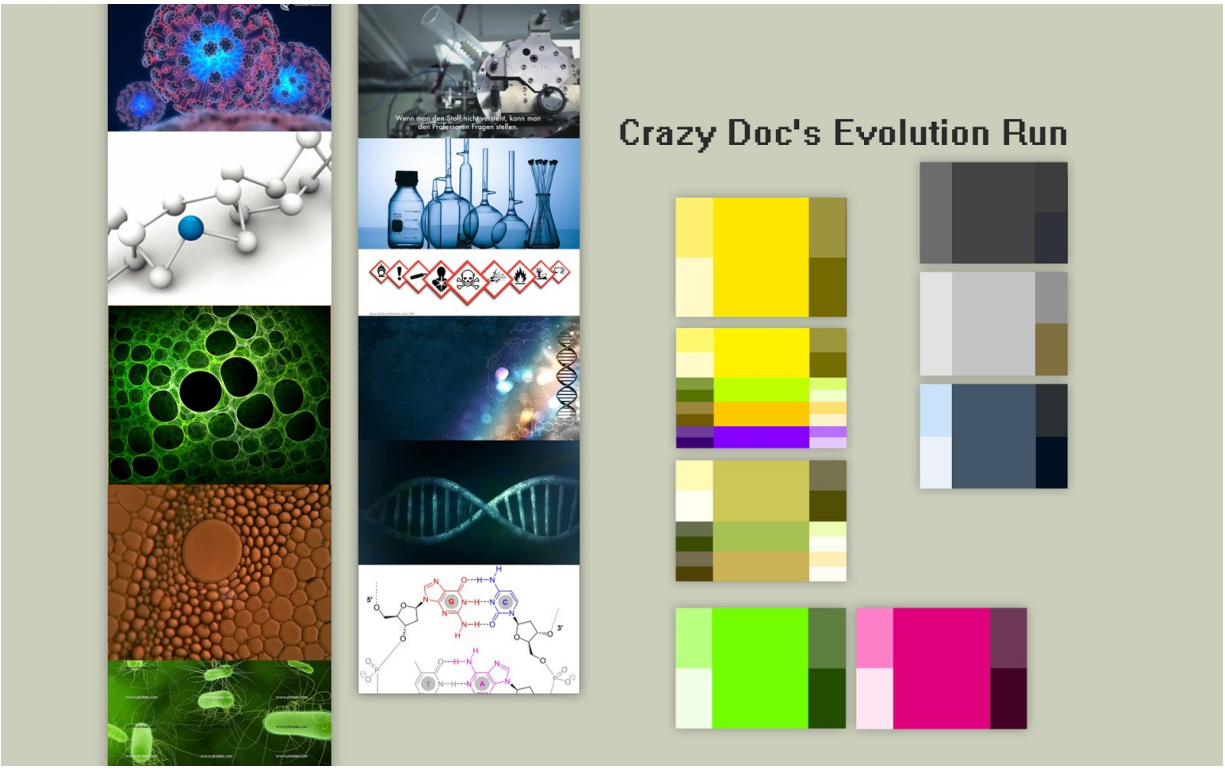
2D Artworks

Stefan Hartwig

Stilrichtung

Mit Hilfe eines Moodboards hat man direkt zu Anfang der Entwicklung versucht ein wenig gestalterische Grundbedingungen zu definieren, um in einem gewissen Entwurfsrahmen zu bleiben. Ein solcher Schritt ist von großer Bedeutung, da die Gestaltung des Spiels nach der gängigen Ordnung der Grundgestaltung stetig einheitlich erhalten bleiben soll. Auf dem Moodboard sind Kontraste, Elemente und Farbkombinationen sichtbar, all dies soll bei jeglichen Entwürfen für das Spiel anregen und einen gewissen "Schubs" verleihen. Desweiteren kann eine solche Darstellung als Referenz und sicherlich auch als eine Instruktion für den gesamten Gestaltungprozess dienen. Nach dem man über die Vorschläge in einer Diskussionsrunde im Team

gesprochen hat und die Alternativen deutlich analysiert hat, ist es möglich gewesen eine feste Vision davon zu erhalten, wie letztendlich das gesamte Ausschaupaket aussehen soll. Weiterführend ist diese Sammlung ein sicherer Ausganspunkt für die Entscheidung der Hauptelemente der Gestaltung gewesen, die dem Spiel einen eigenen Charakter verleihen soll.



Nach dem eine überschaubare Bibliothek der Farben und die feste Vorstellung des gesamten Entwurfs fixiert worden war hat man sich an die weiteren Entwurfsetappen gewagt.

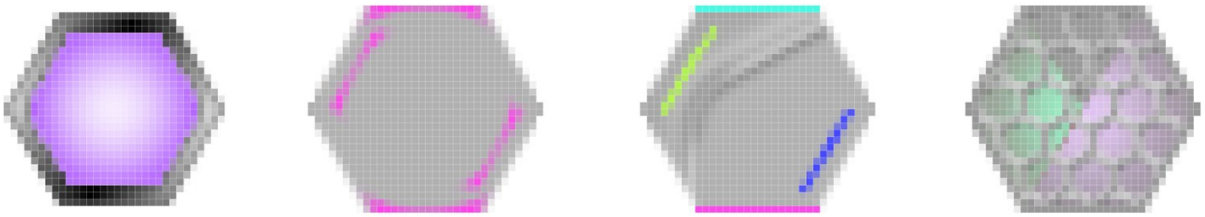
Anwendungsicon

Sicherlich war dieser Schritt der Gestaltung nicht wegzudenken. Und zwar ist es das, wie schlicht für jede Software oder ein Computerbasiertes Produkt, vorgesehene Erkennungssymbol. Auch für das Team war es wichtig ein solches zu besitzen. Man hat also angefangen Entwürfe zu produzieren und da durch vorherigen Überlegungen bestimmte Formen bei Seite gelegt worden waren, hat man sich schnell für die folgende Darstellung entschieden. Es sollte ein Sechseck darstellen und dient dem Icon als ein Rahmen der entsprechend gestalterisch angepasst die Spielinitialen enthalten soll. Die Initialen “CDER” stehen für: “Crazy Doc’s Evolution Run”, also den

hauptsächlichen Namen des Produktes. Der Name ist deutlich lang für jegliche Eingrenzung, die in dem Fall eine einzige Möglichkeit gewesen war, überhaupt eine relevante Referenzierung abzubilden. Es war von vorne rein verständlich dass lediglich die Initialen auf dem Icon zu sehen sein werden, daher existieren auch keine weiteren Varianten die die für die Gestaltung des Icons als eine Grundidee dienen soll.
Es sind lediglich vier unterschiedliche Exemplare des Anwendungsicons entstanden und haben nur darauf gewartet ausgewählt zu werden, zu dem es leider aus zeitlichen Gründen nicht gekommen ist.
Die Icons haben eine gängige Originalgröße und sind auch in der selben Größe während der Entstehung geblieben, um nicht der Hauptsicht im Wege zu stehen, die am Ende der Einbindung dem Betrachter geboten sein wird.

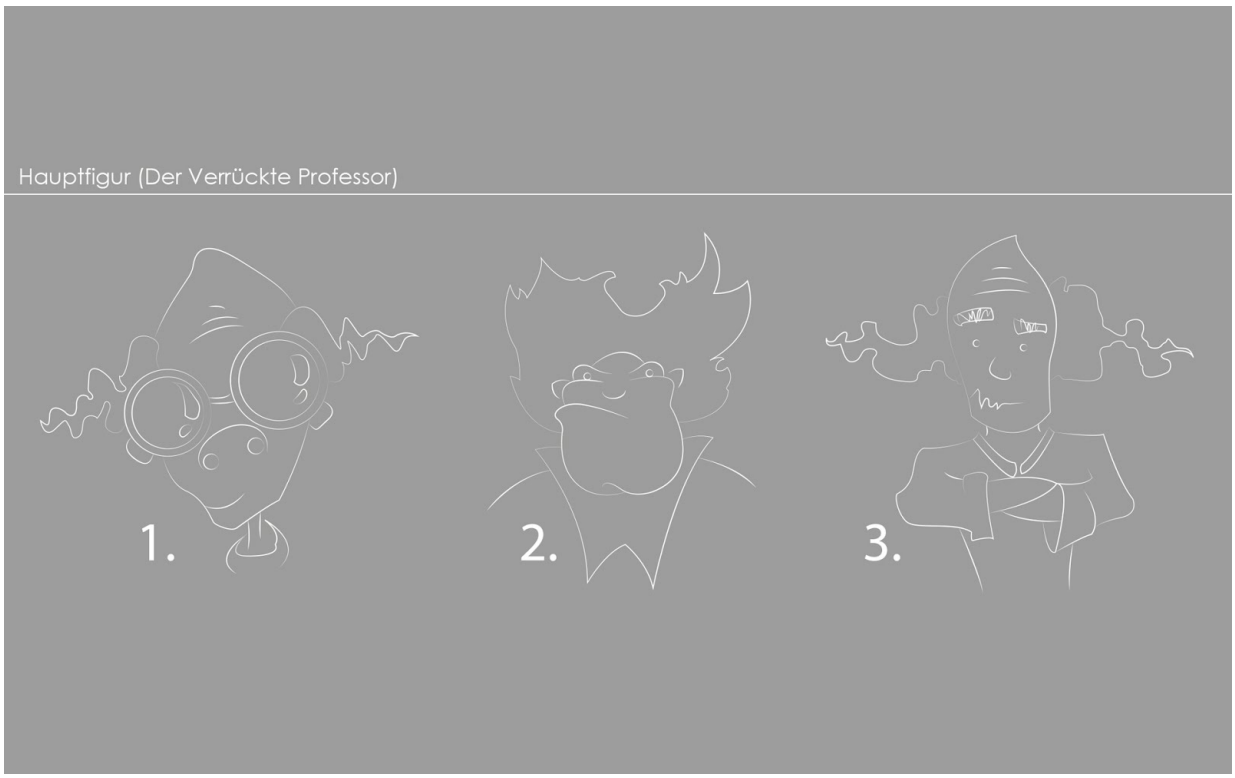
Anwendungsicons, Originalgröße

Um die Details besser zu erkennen und zu interpretieren ist eine weitere Darstellung zu sehen. Es ist eine Vergrößerung des Fundaments und zeigt die Hintergründe ohne der Initialen und die Elemente einzeln aus denen die Icons bestehen.

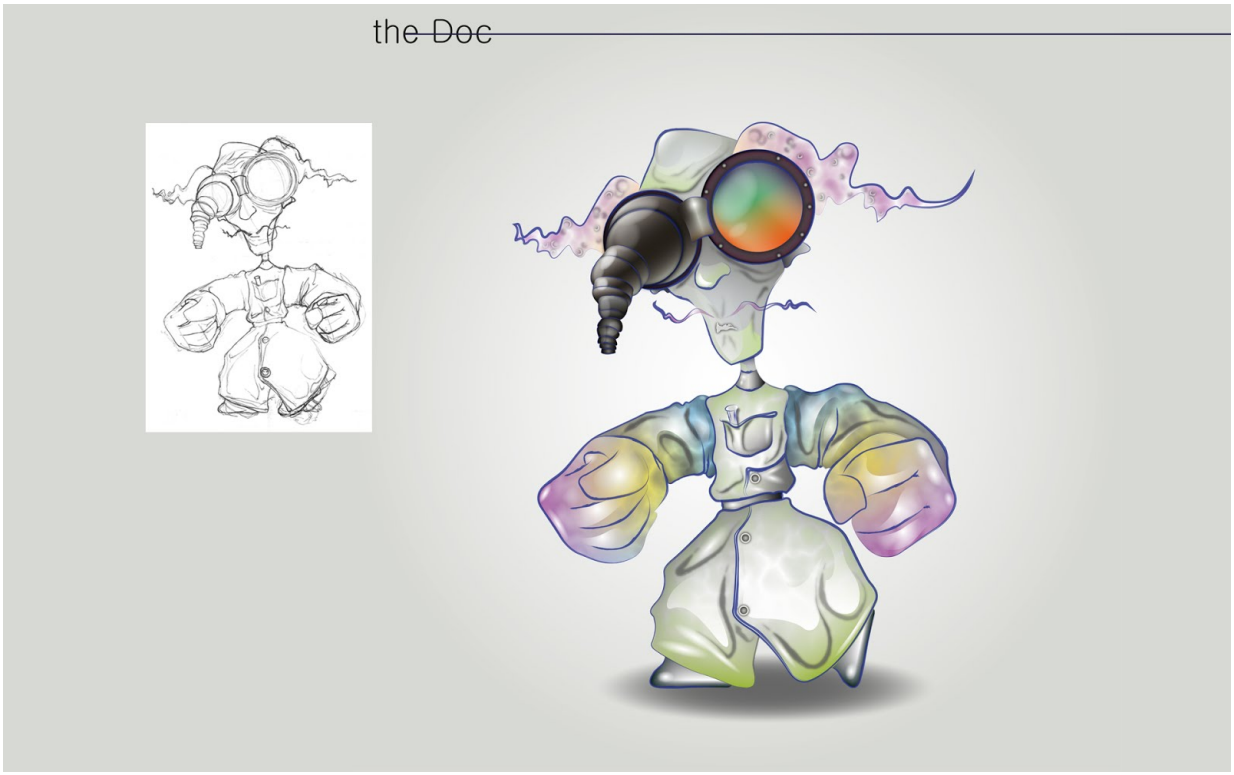


Die Hauptfigur

Wie jede gute Geschichte einen eigenen Hauptcharakter besitzt ist es auch in unserer Wunschabteilung notwendig gewesen eine Hauptfigur zu erfinden. Ein solches Vorgehen ist meiner Meinung nach enorm wichtig, da es eine charmante Art und Weise ist, eine stabile Erinnerung an das Produkt zu erzeugen.



Von Anfang an hatte man sich für eine bestimmte Art der Figur entschieden. Das Muster der Vision ist beim Entwerfen kontinuierlich verfolgt worden und dem entsprechend sind drei Varianten für die endgültige Auswahl entstanden, aus denen man sich für eine entschieden hat.



Das Endprodukt der Hauptfigur ist mit Hilfe von Illustrator, einer Vektorbasieren Zeichenanwendung entstanden. Die Figur ist mit bestimmten Eigenschaften ausgestattet, die den Betrachter dazu lenken sollen sich auf diese Weise eine verrückte Kreatur vorzustellen. Diese könnten sein: die Schweißerbrille mit einer ausgefahrener Lupe, die Frisur und der Mustage, ein Reagenzglas in der Tasche, ein typischer Laborkittel und Arbeitshandschuhe. Da es sich um ein Kampfspiel handelt, ist die Figur in eine Kampfposition gesetzt worden, diese erkennt man an den geballten Fäusten und der versetzten Fußstellung. All das soll dem Betrachter vermitteln, dass diese Figur aktive Eigenschafften besitzt und nur darauf lauert, einen erneuten verrückten Gedankenfluss umzusetzen.

Die Stirn

Für die Stirnseite des Spiels sollten ein paar Vorschläge entstehen bezüglich der Farben, Kombinierung und Position. Der Name besteht aus vier Worten, die unterschiedlich lang sind. Es war mehr oder weniger eine Herausforderung die Kombination so auszuwählen, dass es am Ende nach einer guten Lösung ausschaut. Man hat versucht aus dem Namen eine Form zu kneten, so dass ein bestimmter Rahmen entsteht und es dem Hauptcharakter des Produktes nicht widerspricht. Nach einem zusammenstellen der Variante hat man die Ergebnisse präsentiert und sich für eines davon entschieden. Dieser Entwurf ist bewusst aus zwei Farben entstanden um das Team bei der Entscheidung nicht vom Thema abzulenken. Es soll lediglich dabei helfen die Form des Aussehens festzulegen, wie soll es auf der Hauptseite des Spiels repräsentiert werden.

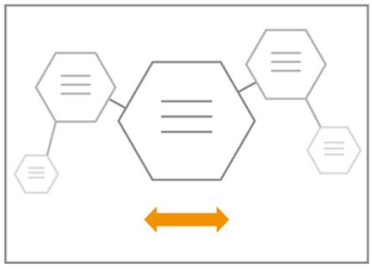
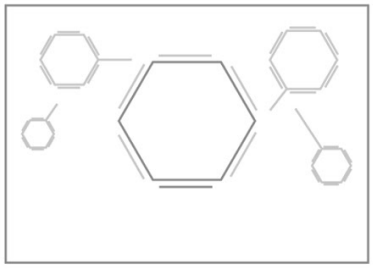
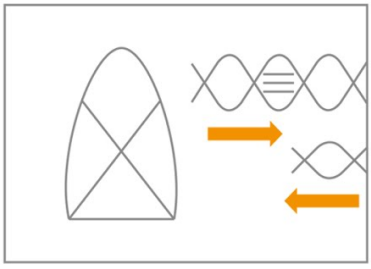
Die Kombination aus den Farben, eine recht verspielte Schriftart und die unterstützenden Gestaltungsmuster sind schließlich dazugekommen und stellen nun den Namen des Spiels in einem ganz neuem Licht zur Schau.
Die Komponente besteht aus einer Würfelartigen Schrift, einem innen enthaltenen Muster, der den Leitfaden der Gesamtgestaltung wiederspiegeln soll, Glanzeffekten, Schatten und plastischen Akzenten der jeweiligen Vorderseite. Im großen und ganzen ist das Team von der Lösung aus dem Bereich zufrieden gewesen.

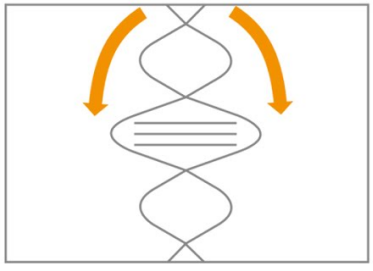
CRAZY DOGS EVOLUTION RUN

Anbei ist man ist man an einen Punkt angelangt, die entworfenen Komponenten in einem Gesamtbild sehen zu wollen. Dabei ist ein Entwurf entstanden, der es geschafft hat die Teammitglieder für die vorherige Entscheidung eine positive Zensur zu verleihen. Es ist viel offensichtlicher geworden, dass die Entscheidung zu Gunsten der Betroffenen gefallen war. Eine solche Darstellung soll dem Spieler einen Vorgeschmack bieten, die Stimmung wiederspiegeln und zu gleich eine Funktion für die Bedienelemente enthalten. Die Darstellung mit den jeweiligen Komponenten für die Stirnseite des Spiels entspricht den genauen Vorstellungen der restlichen Entwickler des Produktes.
Die Darstellung enthält ein Hintergrundbild, das eine Szene aus dem Spiel zeigen soll mit einer Spielfigur rechts in der Ecke sichtbar, die Hauptfigur, Optionsbuttons, der Name und zur Unterstützung dienende Gestaltungselemente auf der nächsten Ebene.

Die Bedienung

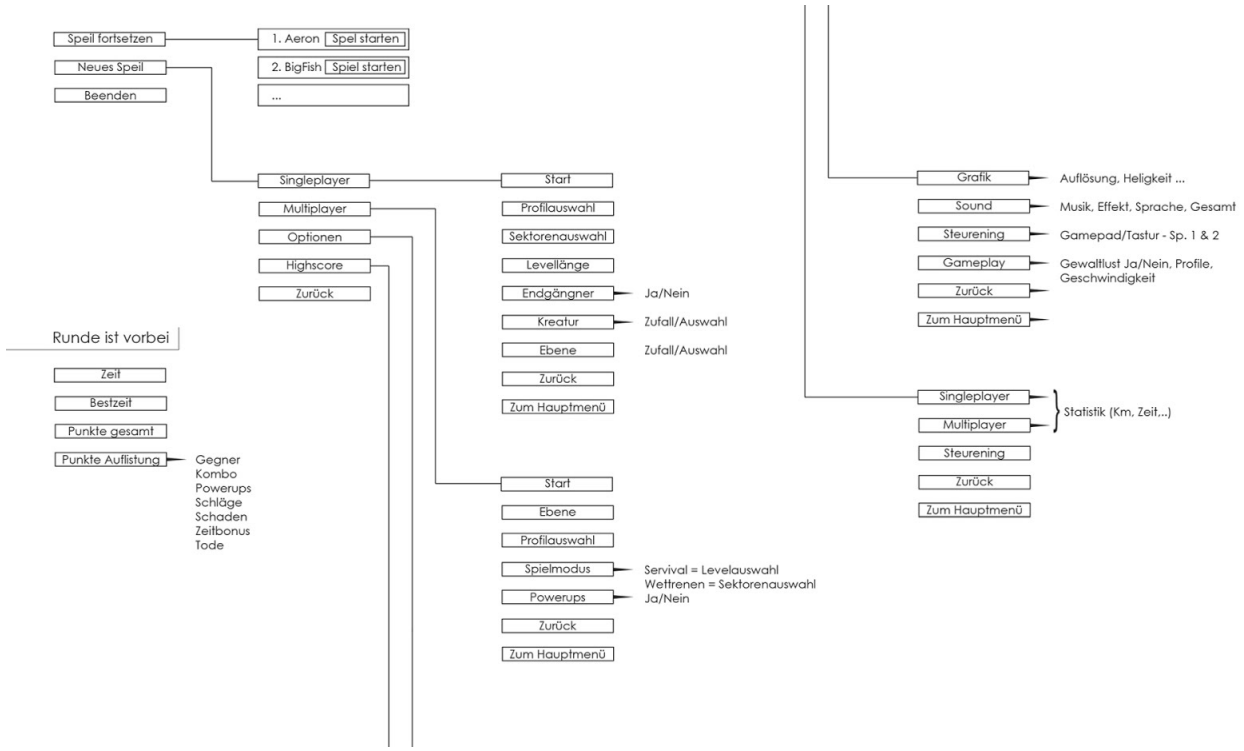
Der nächste Schritt ist es gewesen eine Möglichkeit zu finden wie man anschließend die Bedienelemente, die für die unterschiedlichen Optionen benötigt werden, im wesentlichen bedient. Es gab einige Vorstellungen und Wünsche diesbezüglich und man hat versucht diese nicht zu umgehen, sondern eine angemessene Mischung daraus zu machen. Es soll leicht und zu gleich unterhaltend wirken. Dabei soll die Aktion nicht zu viel Zeit kosten, da der Anwender nach längerer Anwendung der Bedienelemente mit dessen Bewegung bekannt ist und deshalb sehr schnell gelangweilt werden kann. Es war also eine Herausforderung einen Weg zu finden, der all diesen Bedingungen entspricht. Dabei sind einige Varianten entstanden.
Die Interaktion des ganzen soll wie folgt ausschauen: die Grundelemente sollen beweglich sein, sie müssen sich verschieben können, im Hintergrund verschwinden und sich wieder hervorheben. Auf den Elementen sollen dann die Optionen einzeln platziert werden. Leider ist es zu der Umsetzung aus Zeitlichen Gründen nicht gekommen und man hat sich für eine schlichte Variante entschieden.



Optionsarchitektur

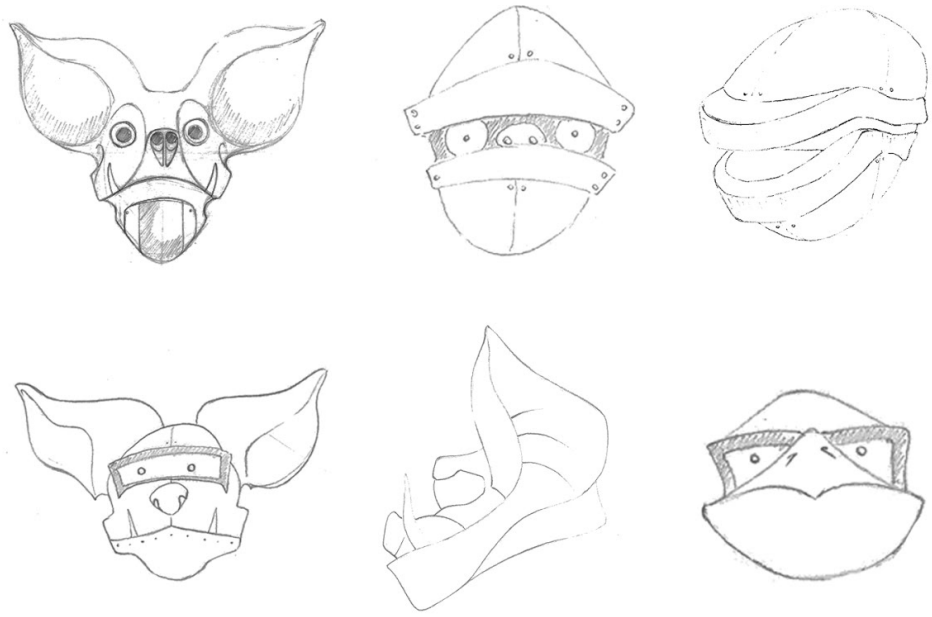
Es wurde überlegt eine komplette Informationsarchitektur aufzustellen um all die Einstellungsmöglichkeiten und Optionen, die man sich für die Umsetzung eventuell vorstellen könnte, für sich selbst näher zu bringen. Dabei ist folgendes zu beachten: es sind drei Auswahlebenen entstanden, je tiefer man hinein geht, desto mehr Auswahlmöglichkeiten werden dem Anwender geboten. Eine solche Hierarchie hat sich als negativ erwiesen und man hat direkt nach einer schlichteren Lösung gesucht. Die Tests haben Hinweise sichtbar gemacht und zwar, dass der Anwender keine ausgiebigen Einstellungen vornehmen möchte, bevor er überhaupt in die Spielwelt eintauchen kann. Es soll einen klaren Weg geben um mit dem Spielen direkt zu beginnen.



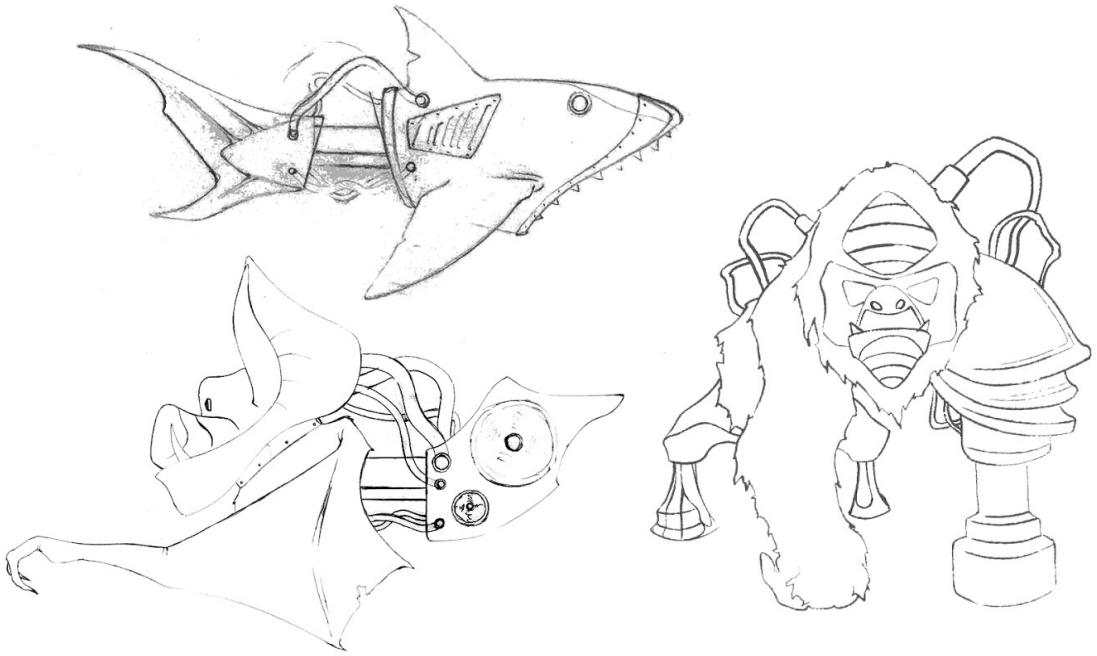
Das In-Game Design

Entwürfe der Spielobjekte

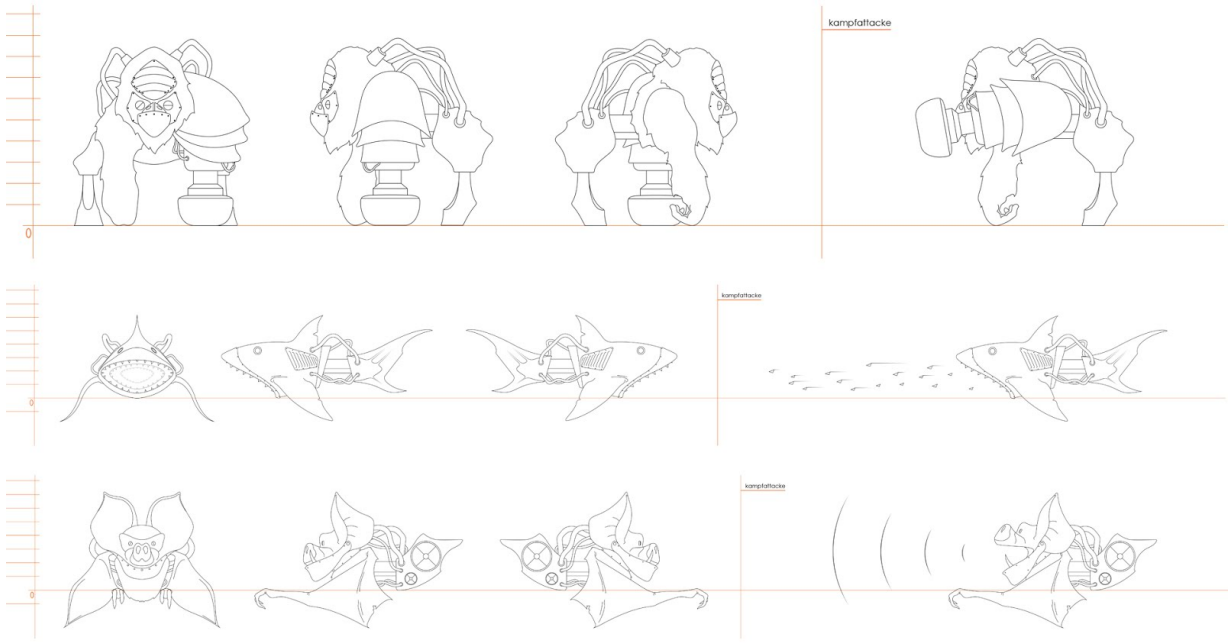
Da es sehr häufig zum Modellieren gekommen ist, und das Spiel sicherlich allerlei Objekte braucht um es plastischer gestalten zu können, waren diverse Skizzen notwendig, um die Figuren heraus zu kristallisieren, die den eigenen Wünschen und Vorstellungen entsprechen. Man hat also angefangen sich Gedanken zu machen wie die Striche gesetzt werden müssen, damit eine angemessene Lösung dabei herauskommt. Da es in der Hauptgeschichte um Kombinationen aus unterschiedlichen Lebewesen geht, war es recht einfach etwas zu entwerfen. Ein Aspekt war aber sicherlich nicht so einfach. Und zwar ging es darum einen Weg zu finden die Spielfiguren so zu entwerfen, dass sie sich im gesamten in der Mitte zwischen realistisch und verspielt befinden. Die Modelle für das Spiel sollen ein realistisches Wiedererkennungsmerkmal besitzen und dennoch sollen die Figurenpassend zur Geschichte angelegt und für den Anwender amüsant und eigenartig ausschauen. Dabei sind vorab folgende Entwürfe entstanden:



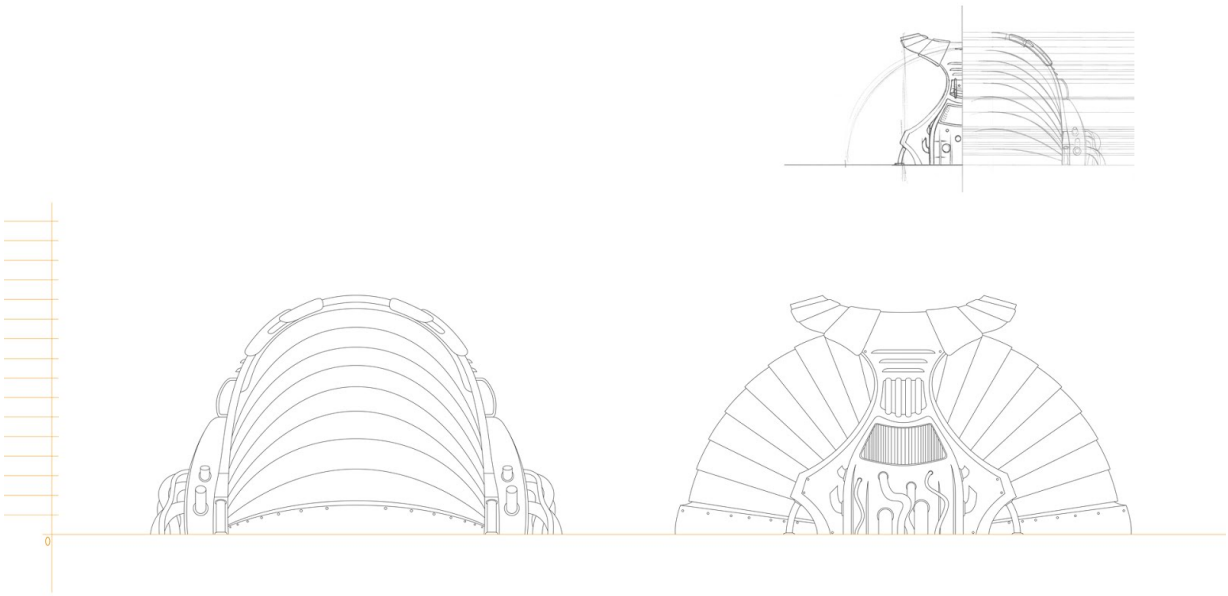
Der folgende Schritt war dazu notwendig, um die Entwürfe so vorzubereiten dass die letzten Feinschliffe passieren konnten. Man hat sich dabei für eine Lösung entschieden und diesen Stil bis zu der Abgabe verfolgt. Der Geschichte zu Folge sollen die Gegner, für die die Entwürfe im eigentlichen Sinn entstanden sind, eine Mischung aus Lebewesen und Maschine sein. Dabei sind folgende Entwürfe entstanden:



Nun sind die Entwürfe der endgültigen Aufbereitung bereit gewesen und man hat begonnen die Outlines für eine maßgetreue Umsetzung anzulegen. Mit Hilfe von Illustrator habe ich dies also übernommen. In der Darstellung sind drei Figuren sichtbar, Luft, Erde und Wasser. Für die jeweiligen Figuren sind nebenbei die Kampfattacken überlegt worden und dementsprechend daneben abgebildet, wie solche sich zu benehmen haben wenn der Fall eintritt.



Man hat sich gelegentlich daran beteiligt Ideen zu präsentieren, wie diverse andere Objekte, die im Spiel zu sehen sein sollen, auszusehen habe. Dabei hat man z.B. einen Entwurf für den Genmanipulator erstellt. Die Entwürfe sollen wie eine Schablone für die spätere Umsetzung dienen, deshalb ist es notwendig gewesen die Darstellung so anzulegen, dass man das Objekt aus mehreren Perspektiven sehen kann. Der Genmanipulator ist leider nicht umgesetzt worden, da es in der Umsetzung zu schwerwiegenden Fehlern kam und man hat sich eine schlichtere Lösung überlegt.

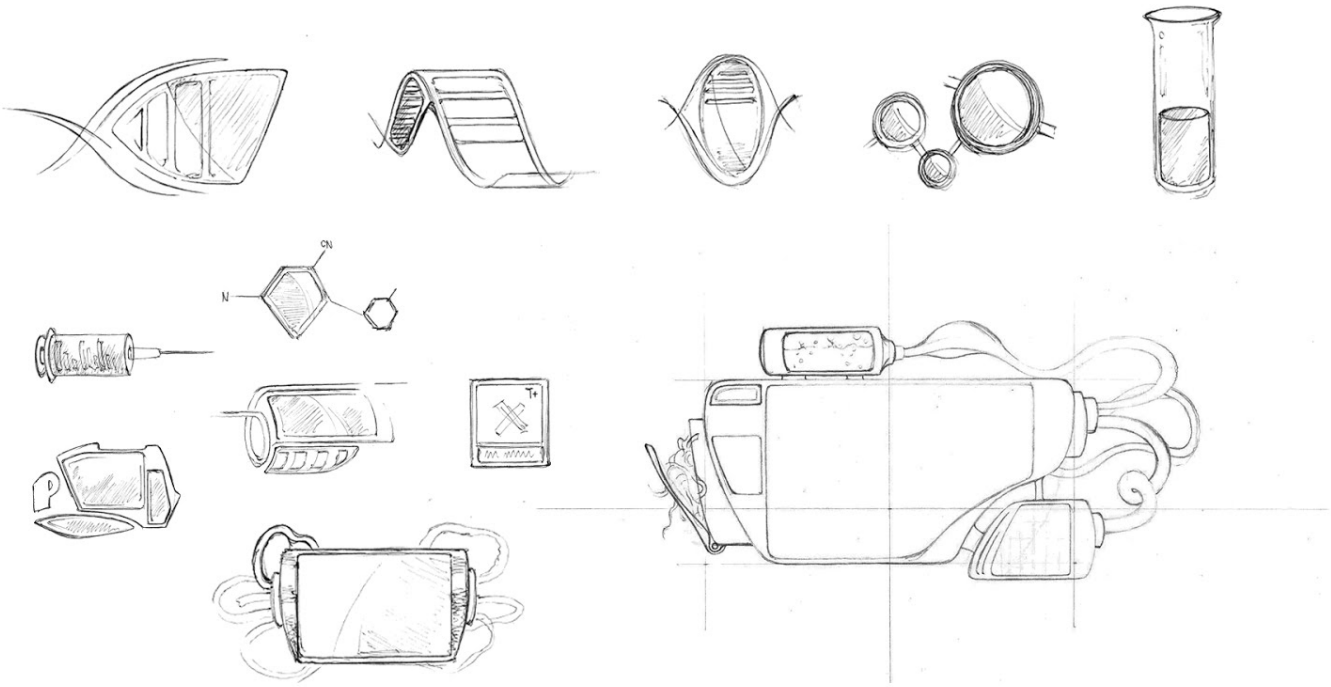


Mockups

Entwürfe

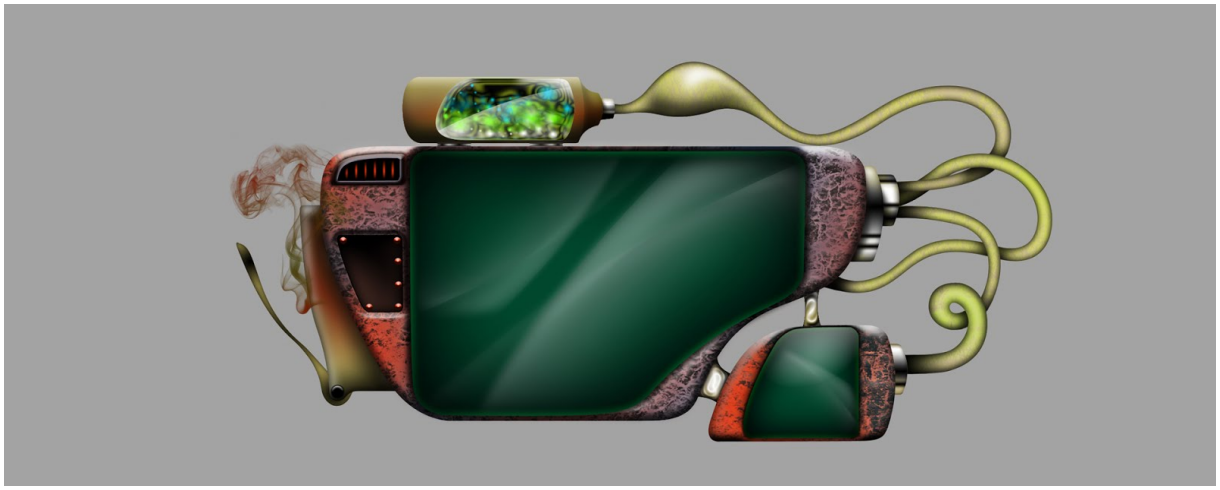
Mockups sind wichtige Bestandteile des Interfaces und haben einen hohen Wert für das Produkt, da solche Komponenten eine enorm wichtige Funktion besitzen. Sie dienen dem Anwender als eine Anzeige des Zustands ihrer Spielfiguren. Es ist also entscheidend, wie solche auszuschauen haben. Es ist wichtig die Komponenten so zu gestalten, dass sie insgesamt angenehm anzuschauen sind und jegliche Gestaltungselemente, die eben den Charakter solcher Komponenten letztendlich formen sollen, den Betrachter oder den Anwender nicht stören oder gar vom Interagieren ablenken. Es sollen diverse Komponenten auf dem Display sichtbar sein und jede davon soll mit einer eigenen Funktion ausgestattet sein, z.B. eine Zeit- und Punktanzeige.

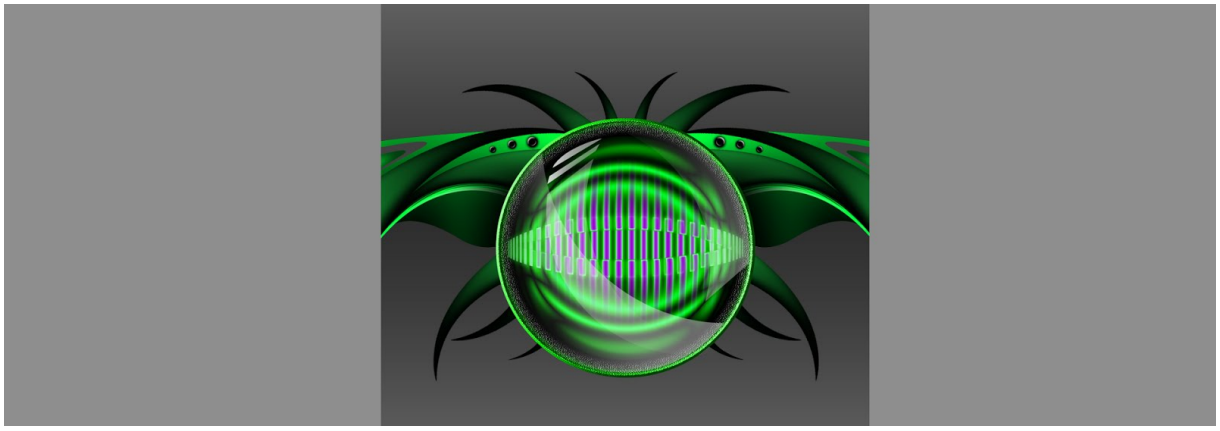
Dazu sind vorab diverse Skizzen entstanden um eine ungefähre Vorstellung davon zu erhalten, wie solche am Ende auszusehen haben. Beim Entwerfen war es wichtig die Komponenten so zu formen, dass sie parallel ein Merkmal erhalten, das sie direkt mit dem Spiel verbindet. Also einen Individuellen Look. Es war insgesamt eine Herausforderung, eine solche Mission zu übernehmen, da zu erst die Gestaltung von großer Bedeutung ist, eine wichtige Information auf den Displays zu sehen sein muss und zu gleich die Anzeigen eine kleine Größe haben. Die ersten Entwürfe dazu sehen folgendermaßen aus.



Entwicklung

Die folgenden Darstellungen sollen einen Einblick verleihen, wie als erstes ein solcher Prozess ab lief. Man hat sich regelrecht auf die Suche begeben, nach der perfekten Kombination von all den Eigenschaften, die undenkbar sind für solch eine Anzeige. Man hat Details eingefügt, die lediglich das Ausschauen unterstützen sollen, da es sonst das Zentrum der Komponente von großer Bedeutung sei, da all die Information folglich bei der Interaktion sichtbar sein wird. All die drei Varianten der Gestaltung haben zwar Potential, nur leider haben die Tests erwiesen, dass diese ist keine optimale Lösungen sind. Die Gründe dafür sind plausibel, es erfüllte zwar den Zweck der Unterhaltung aber nicht der Fokussierung der Information, die später auf dem Display zu sehen sein muss. Der Anwender wird also gerne weg sehen wollen, von dem was er eigentlich im Auge behalten soll. Dementsprechend hat man sich gegen die Entwürfe entschieden und hat angefangen zu reduzieren.

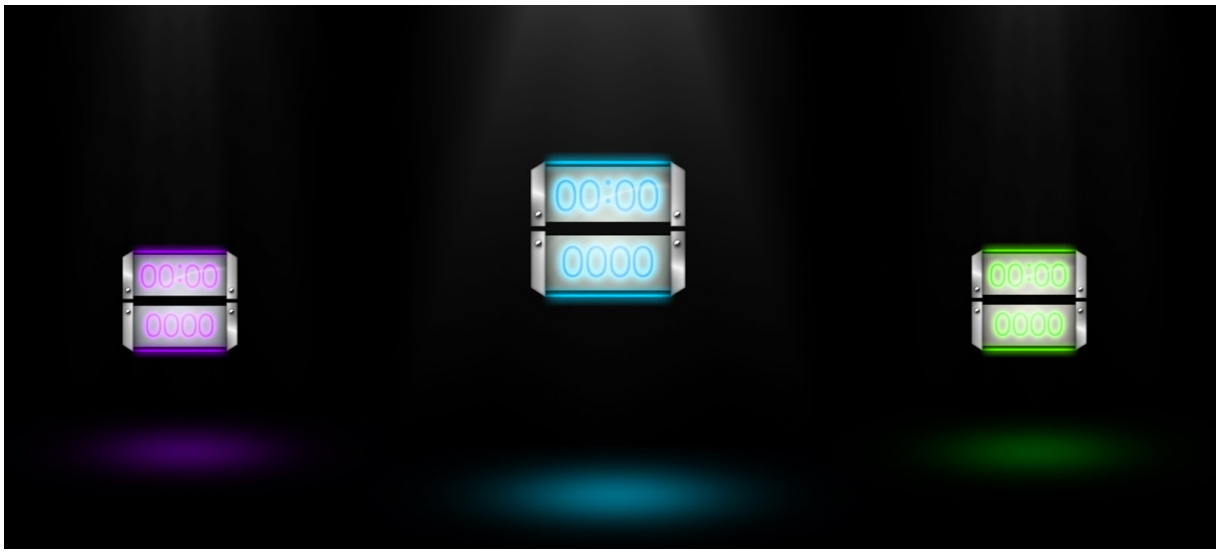




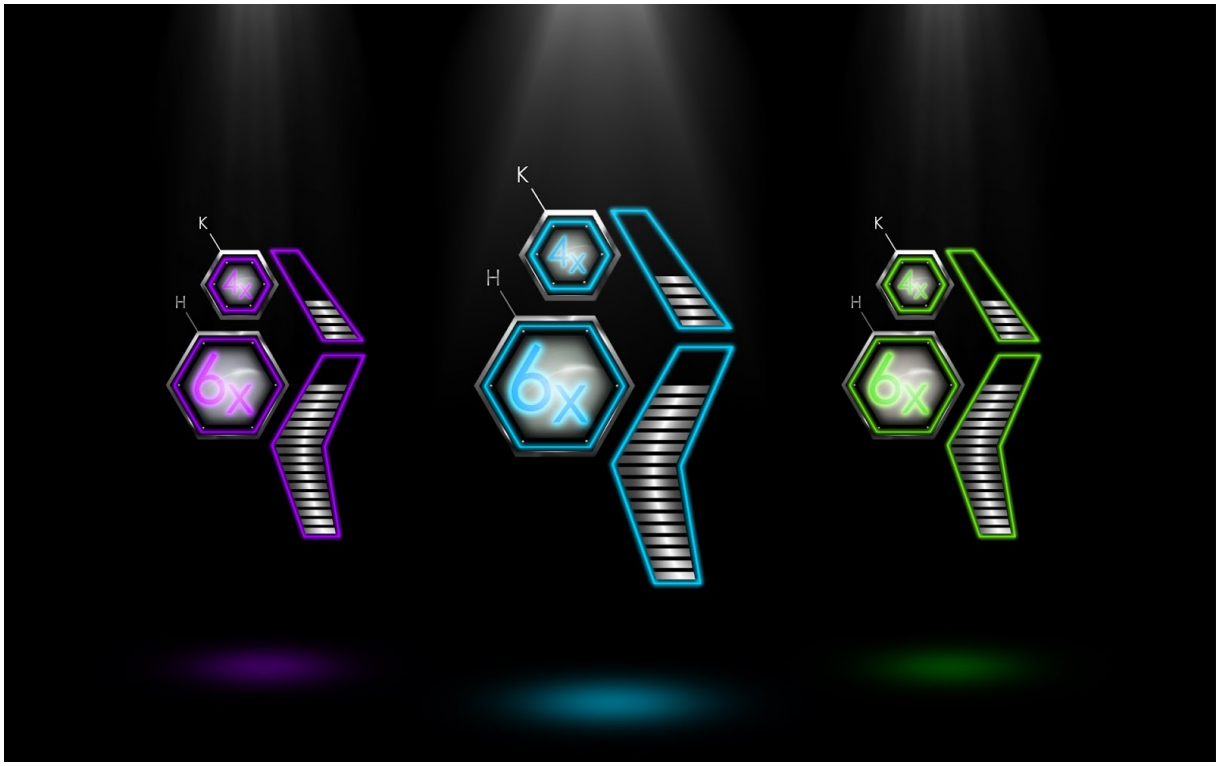
Das Endprodukt

Nach vielen Versuchen ist man endlich an einem Punkt angekommen, der deutlich näher am Ziel ist. Man hat eine Lösung gefunden die Komponenten so zu gestalten, dass sie beides enthalten, das Aussehen und eine Funktion.

Zeitanzeige



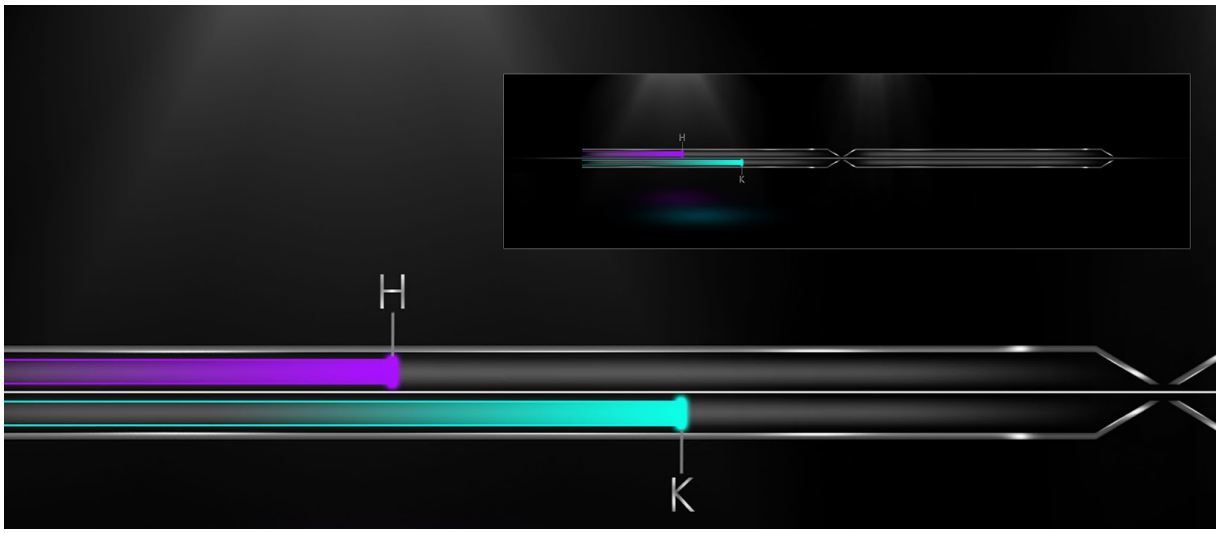
Hauptanzeige



Iconanzeige

Die Haupteigenschaften der Gestaltung der Komponenten sind, die Metallflächen und die Neonleuchtenden Rahmen. Die Hauptanzeige besteht aus vier Grundelementen: Lebensanzeige, Energieanzeige, Iconanzeige (Für Powerups) und der Multiplikator. Jedes davon hat eine eigene Funktion. Beim Entwerfen hat man versucht diese vier Anzeigen beisammen zu halten, da diese Priorität haben. Die Anzeigen sind in den meisten Fällen zentral platziert und ein solches Grundmuster wollte man auch nicht beschädigen. Folgend ist die Entfernungsanzeige zu sehen. Diese ist wie der Name auch schon sagt für die Darstellung des zurück gelegten Weges zuständig. Darunter befindet sich ein gleichgroßer Bereich der für die Punkte zuständig ist. Es ist ebenso wichtig für diese Art von Information die Anzeigen zu trennen. Die letzte Darstellung ist eine Abbildung der Icons. Sie sind in der Hauptanzeige sichtbar und sind dafür da um eine zusätzliche Information zu erteilen und zwar in dem Fall, wenn ein Spieler manipuliert wird. Die Bezeichnungen für die jeweiligen sind oben drüber zu sehen.

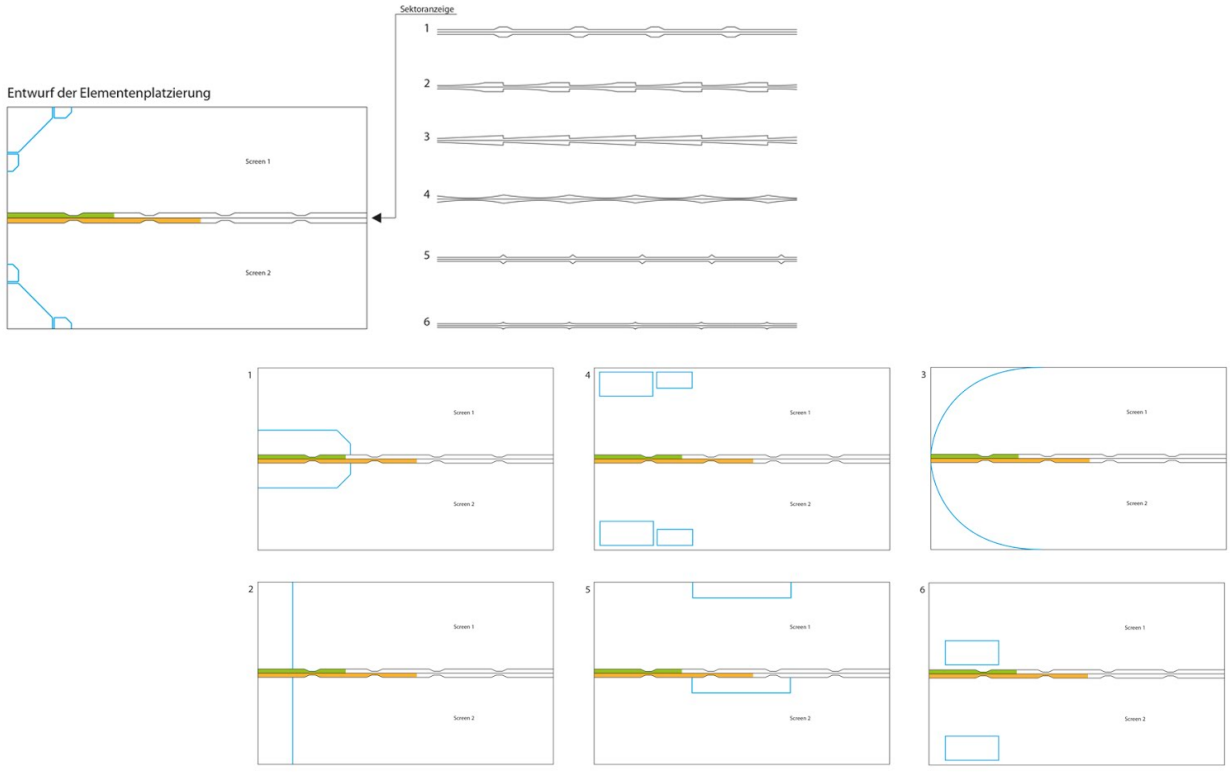
Sequenzanzeige



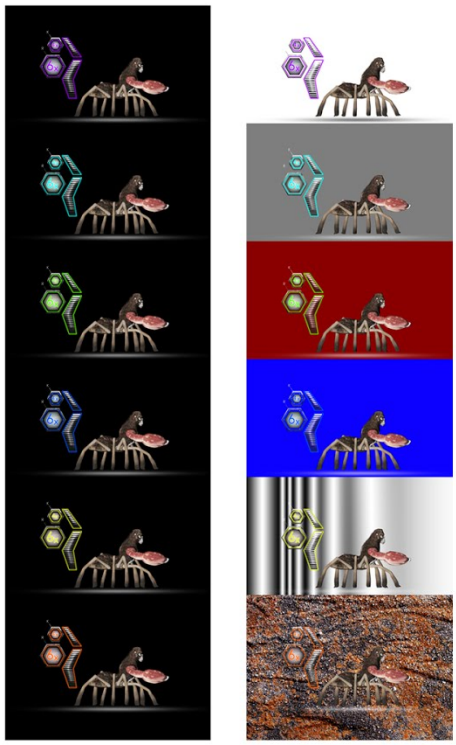
Die Sequenzanzeige hat den Zweck dem Anwender zu zeigen wo er sich in dem Moment befindet. Die Unterteilung der jeweiligen Spieler ist deutlich sichtbar. Läuft der Balken voll mit Farbe und ist der Spieler an dem Knick angekommen hat der Spieler einen Sektor hinter sich gebracht.

Platzierung der Mockups

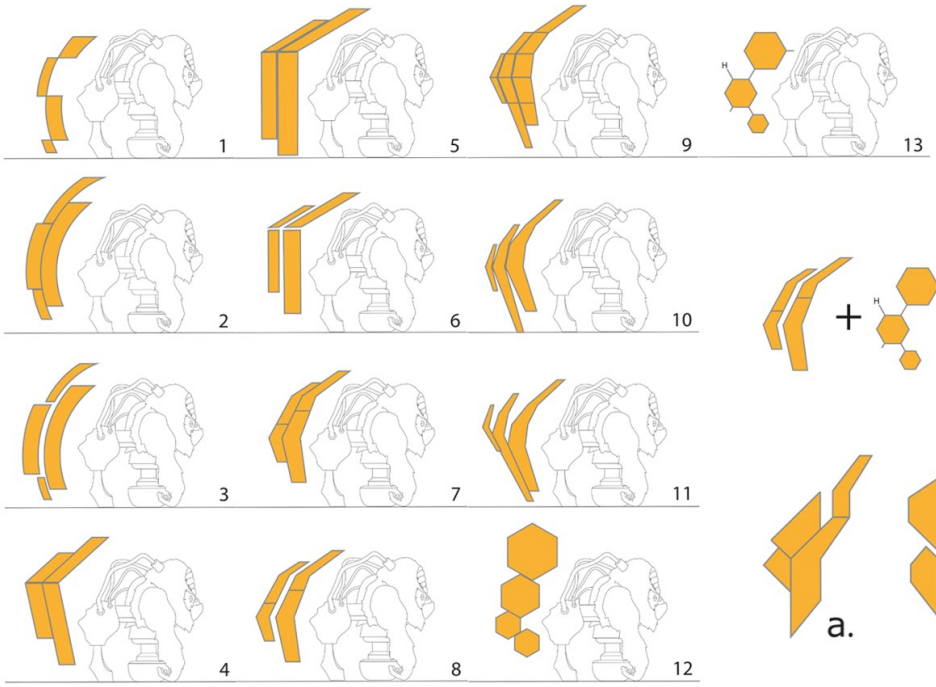
Genau so wichtig wie die Gestaltung der Interfacekomponente ist auch das Entwerfen der genauen Platzierung solcher Elemente und dies sollte gut überlegt sein, denn es macht, eigener Erfahrung zufolge, einen sehr großen Unterschied. Daher hat man einen Entwurf der Entscheidungsdebatte vorgelegt und man hat sich die Grundlagen des Problems gemeinsam angeschaut. Der Recherche zufolge habe man erfahren können, dass zwei übliche Möglichkeiten existieren die Interfacekomponenten zu platzieren. Die einen machen es statisch, in einem solchen Fall haben die Mockups zwar ebenso einen festen Platz, haben aber einen bestimmten Hintergrund wo sie letztendlich platziert werden. Im unserem Fall hat man sich gegen die Art entschieden und die Elemente freistehend auf der Spielebene platziert. Der Entwurf dafür sieht folgendermaßen aus:

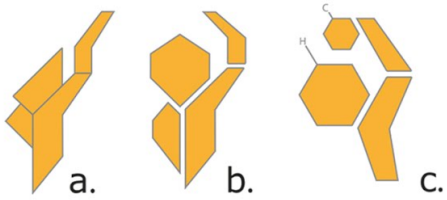


In der vorletzten Etappe ist man an die Positionierung der Hauptanzeige herangegangen. Die Hauptanzeige sollte nah an die Spielfigur gesetzt werden, um den Anwender nicht viel nach den Hauptobjekten suchen zu lassen. Dafür hat man diverse Möglichkeiten der Kombinierung der Anzeigen direkt an der Spielfigur testen können.



In der letzten Phase ist ein letzter Schritt notwendig und zwar die Komponenten in einem Szenario zu testen. Dafür hat man die Spielebene simuliert um den Testzweck mehr Validität zu verleihen. Man hat also eine Art Bühne zusammen gestellt um Platz zu gewinnen für die Komponenten und natürlich der Spielfigur. Man hat in einem Fall die komplette Sammlung von Interfaceelementen in das Szenario geholt und für die Platzierung das vordefinierte Muster verwendet. In einem anderen Fall hat man gleichzeitig mit der Wechselwirkung der Komponenten und der extremen Hintergründe herum probiert. Man wollte Bilanzen ziehen bezüglich des Zusammenspiels von zwei Hauptebenen und hat dafür unterschiedliche Hintergründe vorbereitet um deutlich sehen zu können welche Eigenschaften in welchen Fällen nicht wunschgemäß funktionieren.


1
5
9
13
2
6
10
3
7
11
4
8
12


a.
b.
c.

Testing

Stefan Hartwig

Im wesentlichen ist allen Entwicklern und Gestaltern bewusst und bekannt, dass für das Testen eine Alternative zu finden nicht leicht fallen werden, vor allem weil das Resultat einfach die wichtigsten Fakten entschlüsselt. Daher ist auch in dem beschriebenen Projekt sehr viel getestet worden. Es gab viele Möglichkeiten es spontan oder angekündigt durchführen zu können. Man hat entweder im Team oder privat getestet. Waren es drei unterschiedliche Entwürfe, die man dem Probanden zur Vorschau vorgelegt hatte oder einfach nur eine einzige Darstellung die bestimmte Reaktionen ausgelöst hat. Alle Methoden habe wunderbar funktioniert und begehrenswerte Tatsachen ans Licht gebracht.

Hans Ferchland

Anfangs waren ausgelassene Tests zur Funktionalität in der Programmierung leider wenig möglich. Zur Mitte des Projekts war das Spiel prototypisch genug fortgeschritten um zum einen die Mechanik des Gameplays zu Testen, als auch die Robustheit des Codes. Vorallem durch die Leveldesigner wurden Bugs entlarvt und dann behoben. Regelmäßiges debuggen, provozierten Fehlern und häufiges Probespielen durch die Teammitglieder hat Feedback innerhalb als auch außerhalb des Teams erzeugt, das in die Weiterentwicklung von Features und das beheben von (Schönheits-)Fehlern floss.

Ahmed Arous

Das Testen der Steuerung hat sehr viel Zeit in Anspruch genommen. Dies ist einerseits damit verbunden das ich Anfangs regelmäßig meine eigenen Module bauen musste um einen unverfälschten Test sicher zu stellen, denn die anderen Module waren meist selbst noch Baustellen und andererseits damit das Fehlerquellen nicht sofort ersichtlich waren. Wenn man plötzlich durch einen Collider schoss oder aus heiterem Himmel Saltos schlug, begab man sich in den Skripts auf Fehlersuche ohne unbedingt einen Anhaltspunkt zu haben. Man hat früh die Erfahrung gemacht vorsichtig mit Änderungen umzugehen und diese umgehend auf ihre vollständige Funktionalität zu überprüfen. Mein bisheriger Lieblingsfehler lies den Charakter über den Bildschirmrand hinaus wachsen und blieb einige Zeit unentdeckt bis sich ergeben hat, dass es tatsächlich die Kombination zweier Bugs aus verschiedenen Skripts waren, die diesen Riesenwuchs auslösten. Und selbst wenn die Skripts zumindest bewusst fehlerfrei liefen, musste im gewissen Maß natürlich auch das Feeling der Steuerung getestet und sich Feedback dazu eingeholt werden. Da es sich nicht um eine klassische Steuerung

handelt, war es sich nicht wirklich einfach sich an etwas Bekanntem zu orientieren, hier gab es keine Zauberformel. Hier hieß es Trial & Error, mit vielen öffentlichen Attributen wurde im Spiel während der Laufzeit an den Werten geschraubt und das Resultat beurteilt. Auch wenn die aktuelle Steuerung noch zu wünschen übrig lässt, hat sich zu Beginn des Projekts vergleichsweise schon einiges getan.
Ich denke einen Aufbau gefunden zu haben, dass einem sehr viel Kontrolle über die Charaktere verleiht und es nur eine Frage der Zeit ist hier ein zufriedenstellendes Ergebnis zu erreichen.

Marcel Müller

Das Testen war der Level war leider erst ab ungefähr der Hälfte der Projektphase möglich. Zu diesem Zeitpunkt war das Spiel so weit fortgeschritten, dass man die gebauten Level auch tatsächlich spielen konnte.
Vornehmlich getestet habe ich natürlich auf die Spielbarkeit und optische Fehlerfreiheit der von mir gebauten Midule. Natürlich findet man beim Spielen aber auch immer wieder fehler, deren Ursache nicht immer im Level selbst liegt. Bugs wurden der verantwortlichen Person meistens zügig mündlich mitgeteilt. An der Korrektur dieser Fehler wurde in der Regel so schnell wie möglich gearbeitet.
Das Testen der Level war eine iterative Aufgabe. Die Überarbeitung von Scripten, dem Sprungverhalten der Charaktere oder neue Shader machten das wiederholte Testen der Module nötig. Gelegentlich habe ich Freunde und Bekannte spielen lassen um Feedback über das Gameplay und den Levelaufbau von einer unbeteiligten Person zu erhalten.

Teamwork

Christoph Duda

Die Arbeit in diesem Team hat mir viel Spaß gemacht und lief ohne besondere Zwischenfälle in einem ordentlichen Arbeitstempo ab. Bereits bei der Projektplanung

und Erarbeitung des Konzepts, an dem wir anfangs zu dritt gearbeitet haben, konnte man erkennen, dass das Team motiviert war, seine Grenzen an diesem Projekt zu erkunden. Jeder hatte Herausforderungen vor sich und Aufgaben, die er zum ersten Mal lösen sollte. Kam man an einer Stelle nicht weiter, wurde das Problem im Team gelöst. Sehr hilfreich waren da die Wöchentlichen Skype Termine, die meist viel länger ausfielen, als die angepeilte eine Stunde. Für die wirklich schwerwiegenden Probleme und Angelegenheiten, die nicht über eine Telefonkonferenz gelöst werden konnten, gab es ebenfalls Wöchentlich eine Teamversammlung in der Hochschule. Jedes Teammitglied konnte seine stärken in diesem Projekt einsetzen und ich finde die vorliegende Demoversion zeigt den Ehrgeiz des Teams. Ich konnte in diesem Projekt eine ganze Menge lernen, da ich noch nie Charaktere oder Levelelemente für ein Spiel designt habe und nun die Möglichkeit hatte alle Schritte dieser Entwicklung zu durchlaufen und anzuwenden. Dieses Projekt war für mich auch eine willkommene Abwechslung zu den restlichen Semestermodulen, die sehr programmierlastig waren. Interessant war auch zu sehen, welchen Workflow die anderen Teammitglieder mit ihren bevorzugten Tools hatten. Im Projekt gab es auch spontane Planänderungen, die jedoch immer sehr sinnig waren und erst dann eingriffen, wenn ein Teammitglied mit seiner aktuellen Arbeit fertig wurde. Natürlich war es sehr schade, dass uns während des Projekts ein Teammitglied unangemeldet verlassen hat und wir eine lange Zeit vertröstet wurden, was zur Folge hatte, dass viele Aufgaben erst spät von anderen Teamkameraden übernommen werden konnten. Dennoch haben wir unsere Ziele in der vorgegebenen Zeit weitestgehend erreicht und ich würde gerne weiterhin oder nochmals mit diesem Team arbeiten.

Stefan Hartwig

Im großen und ganzen bin ich mit dem Projekt sehr zufrieden. Es gab vieles neues zu erlernen, neue Erfahrungen bezüglich der Zusammenarbeit zu machen und vieles mehr. Meiner Meinung nach hat vieles einwandfrei funktioniert und ist bestmöglich gelungen. Sei es die Termine zur Besprechung ordnungsgemäß einzuhalten, Vorschläge zu demonstrieren, über die Ergebnisse zu diskutieren oder sich gegenseitig unter die Arme zu greifen, all diese Möglichkeiten haben sich gegeben und ich persönlich habe nichts negatives auszusetzen. Man hat sich darum gekümmert solide Leute ins Boot zu holen, die alle was zum Projekt beitragen konnten und die in der Lage waren eigene Kompetenzen vorzuweisen und eigene Methodiken bezüglich allerlei Kooperationssituationen anzuwenden. Es war eine nutzbare Periode und ich finde es wichtig da mit machen zu dürfen.

Hans Ferchland

Vom Beginn bis zum Schluss hat das Projekt Spaß gemacht und Ergebnisse gezeigt. Jeder hat sein bestes gegeben, auch wenn nicht alles erreicht wurde, so ist das Ergebnis schon halbwegs rund, spielbar und spaßig. Durch die Enge Kommunikation

war trotz der Gruppengröße die Aufgabenverteilung und das Zusammentragen der Ergebnisse, wie auch das besprechen von Folgeschritten sehr gut und stets Zielführend! Einzig der Absprung eines Teammitglieds trübt den sonst reibungslosen Projektablauf. Da wir uns alle auch privat gut kannten, war die Teamatmosphäre ausgelassen und natürlich freundschaftlich.

Marcel Müller

Die Zusammenarbeit lief problemlos und harmonisch über die gesamte Dauer des Projekts. Zu Anfang der Arbeit wurden die verschiedenen Aufgaben innerhalb der Bereiche Programmierung und Grafik weiter auf die einzelnen Leute verteilt. Wegen der Trennung der Aufgaben gab es zu Anfang der Projektarbeit wenige Abhängigkeiten zwischen meiner Arbeit und der der anderen Grafiker oder Programmierer.

Nachdem die anderen Grafiker die Arbeit an den Charakteren abgeschlossen hatten, halfen sie bei der Erstellung der Level. Während
das Modellieren und Zusammensetzen der Objekte zu einem Modul noch
immer meine Aufgabe war, kümmerten sich die Anderen zum Beispiel um
Partikeleffekte, optische Filter oder Module mit anderen Themen.
Bei Abhängigkeiten unter den verschiedenen Arbeiten wurde der Bitte um
Hilfe immer schnell nachgekommen, Deadlines wurden im Großen und Ganzen
eingehalten. Bei Problemen oder Verzögerungen wurde während der wöchentlichen Statusbesprechung gemeinsam an einer Lösung gearbeitet und auch meist gefunden. Hilfreich waren sicherlich die wöchentlich angesetzten Meetings zum gemeinsamen Arbeiten.

Ahmed Arous

Schon in vorherigen Semestern wurde mit dem Gedanken gespielt gemeinsam ein solches Projekt anzugehen und rückblickend bin ich äußerst froh darüber, dass es uns im Rahmen dieses Semesters ermöglicht wurde. Da uns keine wirklichen Restriktionen gesetzt wurden und wir großen Wert darauf gelegt haben, dass jeder sich in den Bereichen einbringen konnte in denen er sich persönlich steigern wollte, war das gemeinsame Arbeiten sehr angenehm und überaus motiviert. Ich denke das Ergebnis kann sich sehen lassen und ich bin davon überzeugt, dass jeder beteiligte große Lernerfolge erzielen konnte. Auch wenn wir längst nicht alles erreicht haben was wir uns vorgenommen hatten, bin ich weiterhin von der gemeinsamen Arbeit und dem Spielkonzept überzeugt. Ich gehe davon aus, dass jeder in seinem eigenem Interesse weiter an dem Projekt arbeiten wird. Abschließend bleibt zu sagen, dass die gemeinsame Teamarbeit großen Spaß gemacht hat.

Daniel Glebinski

> Ich schließe mich den Meinungen meiner Kommilitonen an. Die Zusammenarbeit hat mir mit allen sehr viel Spaß gemacht. Ich hatte zuvor mit niemanden aus dieser Gruppe ein Projekt und war sehr erfreut, dass sie mich mit ins Boot geholt haben. Ich würde jederzeit wieder mit dieser Gruppe zusammenarbeiten.

Register

fbx-Dateiformat: FBX (Filmbox) is a proprietary file format (.fbx) developed by Kaydara and now owned by Autodesk. It is used to provide interoperability between digital content creation applications.[...] (*http://en.wikipedia.org/wiki/FBX*)

Gamedesign Document: Schriftstück, dass alle inhaltlichen und technischen Aspekte des Projektes und somit das Projektziel definiert.

Profiler: Tool zur Laufzeitanalyse von Software. Zeichnet u.A. Speicher, CPU/GPU Last und Script-Historie an

Rigging: "Das Rigging ist eine Arbeitstechnik im Bereich der 3D-Animation. Beim Rigging wird mithilfe einer entsprechenden Software ein so genanntes Skelett bzw. Rig aus Bones (Knochen) oder auch Joints (Gelenken) konstruiert, das festlegt, wie die einzelnen Teile eines Meshes (eines Polygonnetzes) bewegt werden können. Nicht selten orientiert man sich bei der Konstruktion an der Beschaffenheit eines tatsächlichen Skelettes, beispielsweise bildet man etwa einen echten Oberschenkelknochen oder ein echtes Kniegelenk nach." (*http://de.wikipedia.org/ wiki/Rigging_%28Animation%29*)

Weight Painting: Der Weight Paint-Modus dient zum Erstellen und Bearbeiten von Vertex-Gruppen. Dabei kann ein Vertex nicht nur Mitglied in einer oder mehreren Gruppen sein, sondern innerhalb einer Gruppe unterschiedliches Gewicht - also unterschiedlichen Einfluss auf das Ergebnis - haben. Vertexgruppen werden u.a. dazu eingesetzt, um die Zuordnung Mesh → Bone festzulegen, also welche Teile des Meshes von welchem Bone bewegt werden.
(*http://de.wikibooks.org/wiki/Blender_Dokumentation:_Referenz:_Weight_Paint_Mode*)

ZBrush: ZBrush ist ein Grafikprogramm der Firma Pixologic, das mit einer Hybridtechnologie aus 2D und 3D arbeitet.

Es ist sowohl als gewöhnliches Malprogramm für 2D-Grafiken als auch zur 3D-Modellierung und Texturierung verwendbar. Die erste Version erschien 1999. Grundlegend unterschiedlich zu anderen Grafikprogrammen speichert ZBrush in jedem Pixel nicht nur RGB- oder Alpha-, sondern auch Tiefeninformationen ab. Daher auch der Name: die Variable Z wird in Raumkoordinaten üblicherweise für die Tiefenkomponente verwendet. Diese speziellen Pixel werden nach dem Firmennamen als Pixols bezeichnet. Auf diese Weise kann man auf die virtuelle Leinwand nicht nur flache Farben malen, sondern auch Reliefs auf ihr erstellen. Der Prozess ähnelt damit dem plastischen Modellieren. Es lassen sich nun auf diese 3D-Daten Beleuchtungs- und Materialsimulations-Algorithmen anwenden wie bei polygonbasierten 3D-Modellen in konventionellen 3D-Programmen.
Ein Objekt in ZBrush kann in verschiedenen Subdivision-Levels bearbeitet werden. Wenn man z. B. ein Objekt mit 1000 Polygonen importiert, kann man es von ZBrush in 4.000, 16.000, 64.000, 256.000, 1.024.000, usw (Anzahl steigt Exponentiell!). Polygone unterteilen lassen (bei jedem Schritt wird jedes Polygon in vier weitere Polygone unterteilt), um es detailreicher bearbeiten zu können, wobei man auch nach der Unterteilung wieder in Levels mit weniger Polygonen wechseln kann [...] (*http://de.wikipedia.org/wiki/ZBrush abgerufen am 30.07.2014*)