Computergrafikpraktikum 11.-29.08.2014

Christoph Eichler, Oliver Vornberger
Universität Osnabrück

Das Ziel der Praktikumgsgruppe bestand darin, mit Hilfe der populären, frei zugänglichen Spiele-Engine Unity, ein Autorennspiel zu entwickeln. Dieses sollte sowohl eine Einzelspielerkomponente gegen mögliche KI-Gegner, als auch einen Mehrspielermodus beinhalten. Weiterhin wurden eine realistische Fahrphysik, detaillierte Strecken und Modelle, sowie Powerups und eine benutzerfreundliche Menüführung in den Fokus gestellt.

Inhaltsverzeichnis

1. Aufgabenstellung

Das Ziel der Praktikumgsgruppe bestand darin, mit Hilfe der populären, frei zugänglichen Spiele-Engine Unity, ein Autorennspiel zu entwickeln. Dieses sollte sowohl eine Einzelspielerkomponente gegen mögliche KI-Gegner, als auch einen Mehrspielermodus beinhalten. Weiterhin wurden eine realistische Fahrphysik, detaillierte Strecken und Modelle, sowie Powerups und eine benutzerfreundliche Menüführung in den Fokus gestellt.

2. Blender

Autoren Benedikt Schumacher, Sargini Rasathurai

Blender

Die Software Blender ist eine kostenlose Software zur Modellierung. Beziehen lässt sich die Software über http://www.blender.org/download/ für alle Plattformen als 32- und 64-Bit-Anwendung.

Es empfiehlt sich auch an Notebooks Blender immer in Verbindung mit einer richtigen Maus zu nutzen.

2.1 Einführung in Blender

Autoren: Benedikt Schumacher, Sargini Rasathurai

Die Bedienung der Software erscheint in den ersten Minuten gewöhnungsbedürftig und kompliziert, erweist sich aber im Ganzen als sehr durchdacht. So hat das Programm etliche Tastenkombinationen zu den zahlreichen Werkzeugen, die, so diese bekannt sind, die Arbeit enorm erleichtern und beschleunigen. Die Perspektive lässt sich mittels Maus durch das Halten des Mausrades oder durch drücken der Tasten 2, 6, 8, 4 des Nummernblocks um einen Punkt drehen. Auf den Achsen verschieben lässt sich die Kamera, wird zu dem Mausrad auch Shift gedrückt oder es wird Strg und eine der zitierten Tasten des Nummernblocks gedrückt. Zoomen lässt sich durch drehen des Mausrades und mittels der Tasten + und - des Nummernblocks.

Wichtig ist auch, dass Objekte nur durch Klicken mit der rechten Maustaste ausgewählt werden können. Sollen mehrere Objekte ausgewählt werden, muss zusätzlich die Shift-Taste gehalten werden.

In Blender gibt es zwei verschiedene Modi; zum einen den sog. *Object Mode*, in dem ganze Konstrukte bewegt werden können, zum anderen den *Edit Mode*, in dem ein spezifisches Objekt in allen Einzelheiten bearbeitet werden kann. Der Wechsel zwischen den Modi wird durch das Betätigen der Tabulator-Taste vollzogen. Mittels der z-Taste können in beiden Modi die Flächen in den Drahtmodellen ausgeblendet und wieder eingeblendet werden. Durch den Hotkey N erscheint eine weitere Spalte an Einstellungsmöglichkeiten für alle Objekte. Hier können unter anderem die Positionen von Punkten oder auch ganze Objekten verschoben, rotiert und skaliert werden.

Weiter lässt sich durch drücken von F12 die aktuelle Ansicht als gerenderte Szene betrachten. Damit die Objekte anschaulich gerendert werden muss zuvor eine beliebige Lichtquelle gesetzt und gegebenenfalls auf die Szene gerichtet werden. Das Rendering kann je nach Leistung des verwendeten Computers einige Zeit in Anspruch nehmen und das Navigieren durch die Szene stark einschränken.

Auf der linken Seite der Applikation lassen sich geometrische Grundformen per einfachem Klick in die Szene setzen. Hierzu gehören unter anderem eine Fläche, ein Würfel, eine Kugel eine Pyramide etc. Diese lassen sich anschließend im Edit Mode nach Belieben verformen und erweitern.

Auch bietet Blender die Möglichkeit die fertig modellierten Objekte in verschiedene Dateiformate zu exportieren. Für die spätere Weiterverarbeitung in diesem Praktikum mussten die Modelle in dem *.fbx-Format* exportiert werden. Es können aber auch Modelle in Blender importiert und weiterverarbeitet werden.

2.2 Mesh-Modellierung

Autoren: Benedikt Schumacher, Sargini Rasathurai

Meshes bestehen aus einzelnen Punkten (*Vertices*), die durch Kanten (*Edges*) verbunden sind. Die Kanten wiederum spannen Flächen(*Faces*) auf. Um ein 3D- Modell zu erzeugen, kann man eine Kombination der folgenden Techniken anwenden.

Poly-By-Poly Modeling

Bei dieser Technik setzt man einen Punkt nach dem anderen und erzeugt so die Flächen des 3D-Objekts. Sie wird oft angewandt, wenn man bereits ein Blueprint von dem zu erzeugenden Objekt verfügt. Die *Blueprints* werden dann als Hintergrundbilder für die frontal-orthogonale, seitlich-orthogonale bzw. für die Vogelperspektive eingestellt. Hat man nun einzelne markante Punkte der Vorlage (3 oder 4) ausgewählt, kann man mit der F-Taste aus diesen Punkten ein Face erstellen.

Box Modeling

Eine komplett andere Technik ist das *Box Modeling*. Hierbei nimmt man als Grundkörper einen Würfel, der durch verschiedene Werkzeuge wie *Knife*, *Subdivision Surface*, Skalierung/Transformation usw. derart verändert wird, dass die gewünschte Form entsteht. Die Werkzeuge können sowohl auf den ganzen Körper, als auch auf die einzelnen *Vertices, Edges* oder *Faces* angewandt werden.

Modellierung mithilfe von Extrude

Diese Technik eignet sich gut, wenn das zuerstellende Objekt (beispielsweise ein Leuchtturm) einem Grundkörper (Zylinder) ähnelt. Durch Auswählen und Extrudieren (Hotkey „E“) bestimmter Vertices/Edges/Faces können Veränderungen erzielt werden. Dabei werden zusätzliche *Vertices/Edges/Faces* aus einem Mesh in eine bestimmte Richtung „gezogen“. Es besteht automatisch eine Verbindung zu dem Grundkörper.

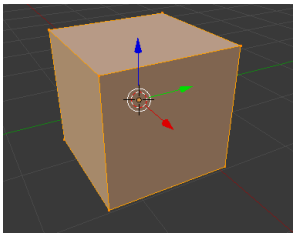
2.3 Blender Werkzeuge

Autor Nils Baumgartner

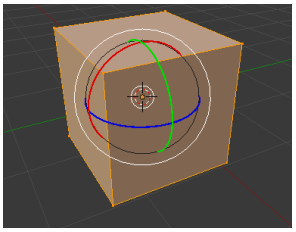
Viele Tools sind leicht benutzbar durch Tastenkombinationen und erleichtern dem Benutzer seine Modelle zu editieren. Zunächst lassen wir unser Augenmerk auf den einfachen Tools, welche die Translation, Rotation und Skalierung beinhalten.

Translation, Rotation und Skalierung

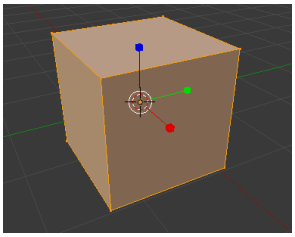
Sobald man eine Auswahl eines oder mehrerer Elemente hat und sich im „Edit Mode“ befindet, kann man diese bewegen („G“), drehen („R“) oder Skalieren („S“). Nun kann man zum feinen manipulieren ebenfalls angeben, um welche Achsen man seine Veränderung erzeugen möchte und einen Wert. Um eine Achse auszuwählen drückt man den Entsprechenden Achsen Anfangsbuchstaben auf der Tastatur oder man möchte alle Achsen außer einer bestimmten wählen so nutzt man die Kombination mit „Shift“ und der nicht zu verändern sollenden Achse. Wenn man ungern mit „Hotkeys“ arbeitet, so kann man auch in den Eigenschaften direkt Positionen angeben (mehr dazu später) oder man mit der Maus an den Manipulatoren ziehen und somit eine direkte Veränderung erzeugen.



Translate

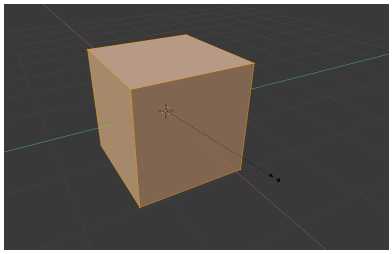


Rotate

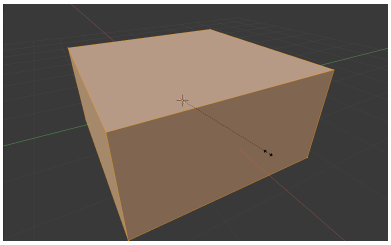


Skale

Als kleines Beispiel wollen wir diesen Würfel Skalieren nur entlang der X-Y Achse um das doppelte. Zunächst wählen wir das Skalier-Tool („S“), würden nun alle Achsen ansprechen wollen außer die Z-Achse mittels („Shift+Z“) und geben nun als Faktor zum verdoppeln „2“ an und bestätigen mit „Enter“.



'S' , 'Shift+Z'



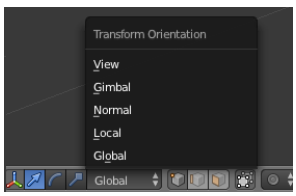
Result

Genaue Positionen von Punkten oder Flächen bestimmten, kann man wie in der „Einführung von Blender“ mittels des „Hotkeys“ N sich die Eigenschaften anzeigen lassen wo man für einzelne Punkte die genauen Koordination angeben kann und für eine Summe von Punkte oder Flächen lediglich den Median bestimmen kann.

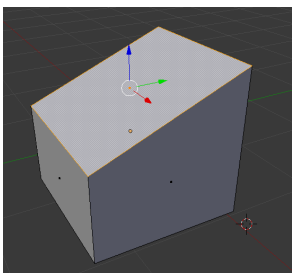
Orientierung

Für spezielle Manipulation (z.B. entlang der Normalen) bietet Blender die Einstellung mittels welcher Orientierung man eine diese durchführen möchte. Blender bietet zudem die Möglichkeit eigene Orientierungen zu erstellen. Dabei gibt es 5 bereits vorhandene Orientierungen:

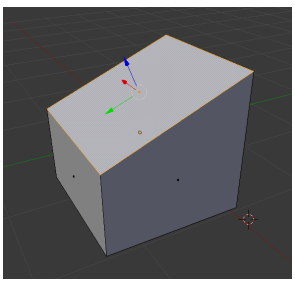
- „Global“
- „Local“
- „Normal“
- „Glimbal“
- „View“



Orientierung



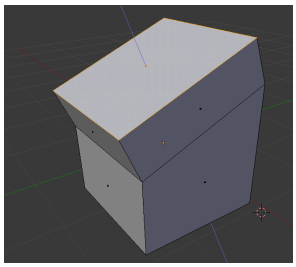
'Global'



'Normal'

Extrudieren

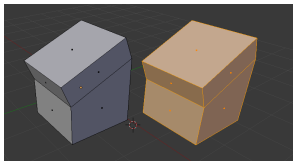
Eines der wohl nützlichsten Werkzeuge in Blender ist das Polygon Netzwerkzeug „Extrude“. Es ermöglicht Quader aus Rechtecken und Zylinder aus Kreisen zu erstellen. Generell ermöglicht das Werkzeug vorhandene Punkte, Kanten oder Flächen zu duplizieren und direkt mit dem Original zu verbinden. Hierbei ist das Werkzeug leicht zu benutzen, bietet aber dennoch große Leistungsfähigkeit und große Individualisierung beim extrudieren. Hierbei wird im Normalfall entlang der gemeinsamen Normalen (bei ausgewählten Flächen) extrudiert. Dies lässt sich aber auch auf bestimmte Achsen begrenzen und sogar auf Orientierungs-Achsen. Das Extrudieren lässt sich mittels des „Hotkeys“ E leicht aufrufen, danach folgen optionale Parameter wie bereits bei der Translation,Rotation und Skalierung.



'Extrude'

Duplizieren

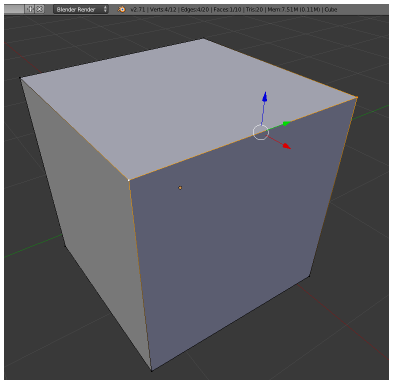
Ein weiteres Werkzeug, welches zwar klein ist aber dennoch genannt werden sollte ist das Duplizieren („Hotkey“ D), womit ausgewählte Komponenten dupliziert werden.



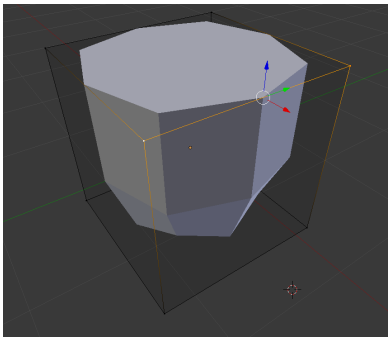
Dupliziertes Objekt

“Remove Doubles“ - Redundanz Entfernen

Mit-einher der Werkzeuge Extrudieren und Duplizieren, kommen beim modellieren menschliche Missgeschicke zustande, indem extrudierte oder duplizierte Punkte direkt über einander liegen. Dies bereitet in der späteren Modellierung und dem „Rendering“ Probleme, da dadurch unschöne Seiteneffekte auftreten. Hierbei bietet Blender die Möglichkeit doppelte Punkte zu entfernen und die angehörigen Flächen und Kanten beizubehalten.



Würfel mit 12 Eckpunkten (2 Ecken doppelt)

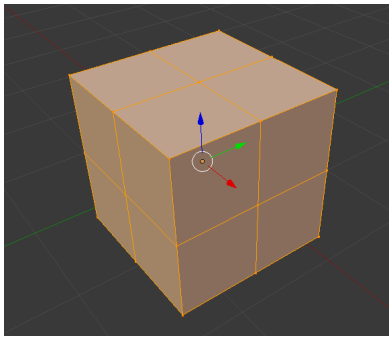


Seiteneffekt bei 'Subdivision Surface'

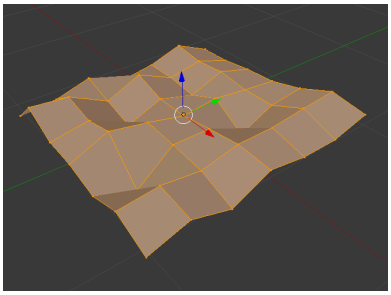
Unterteilen - „Subdivide“

Dieses Werkzeug unterteilt Kanten und Flächen in die Hälfte oder mehr (welches Einstellbar ist) wodurch weitere Kanten bzw. Flächen entstehen. Es ist leicht zu nutzen mit dem "Hotkey" W.

Dieses Werkzeug ist das Allgemeinere Werkzeug des „Modifier“ „Sudivision Surface“. Beide unterteilen zwar die ausgewählten Komponenten in weitere, jedoch lässt sich letzteres permanent aktiv schalten, wodurch die Änderung zum Schluss angewandt werden und rundet bzw. gleicht die entstandenen neuen Komponenten den vorherigen an. Bei „Subdivision Surface“ werden die Parameter jedoch passend gewählt, sodass das gesamte Objekt geglättet wird. Als Optionen bzw. Parameter lassen sich hierbei Schnitt Typen festlegen, als auch Anzahl der Schnitte und ob diese noch Randomisiert werden sollen, wodurch das Objekt in einigen Situationen echter wirkt (bzw. Steine,Terrain,...).



Würfel mit 'Subdivide'

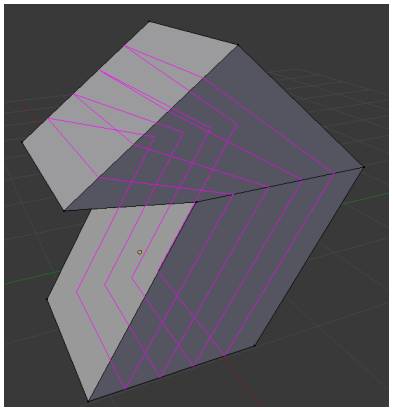


Ebene Randomisiert und unterteilt ('Cuts'=5 , 'Fractal'=1.0, 'Along-Normals'=1.0

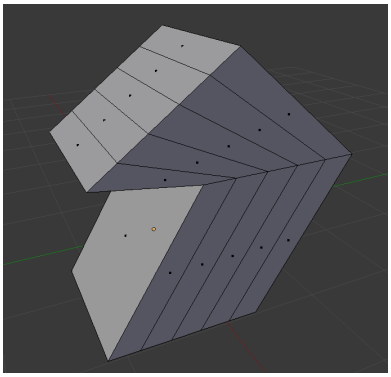
Mehr zu „Subdivision Surface“ später.

Schleifen Unterteilung - „Loop Cut“

Der „Loop Cut“ unterteilt eine Objekt entlang einer signifikanten Seite. Dieses Werkzeug bot großen Komfort bei der Modellierung eines Reifens, da hiermit leicht die Flächen auf dem Reifen erzeugt wurden, und wird mit dem "Hotkey" 'CTRL+R' benutzt. Der „Loop Cut“ bietet dem normalen „Subdivide“ die Möglichkeit umlaufende Schnitte zu erzeugen.



Umlaufender Schnitt eines komplexen Körpers

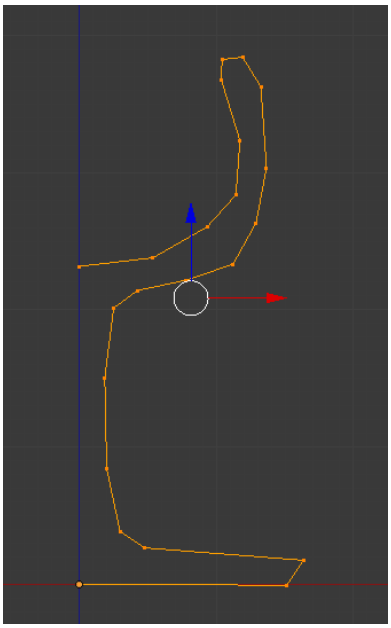


Resultat des Schnitts

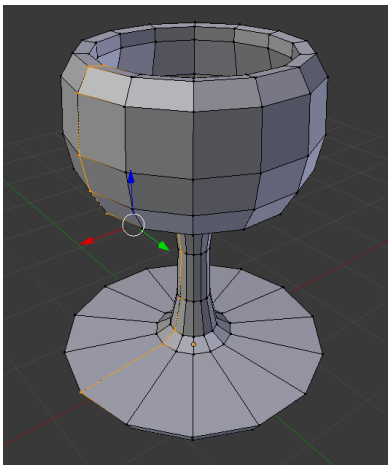
Rotationskörper - „Spin“

Bei diesem Werkzeug lassen sich Rotationskörper gut modellieren, bei denen der Spiegel „Modifier“ nicht gut anwendbar ist. Beim „Spin“ wird das Objekt um eine Achse gedreht. Optional können eingestellt werden:

- “Steps“ - Schritt-zahl
- “Angle“ - Winkel
- „Center“ - Drehpunkt
- “Axis“ - Drehachse



Weinglas im Profil



Rotiertes Profil wird zum Weinglas

Benutzte Einstellung:

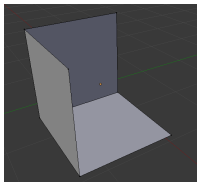
- Steps : 14
- Angle : 360
- Center : 0.0|0.0|0.0

- Axis : 0.0|0.0|1.0

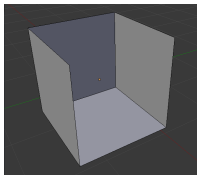
Komponenten Löschen

Um ungewollte Punkte zu beheben kann man Komponenten löschen oder mit anderen zusammenführen. Entscheidet man sich für ein Löschen eines Punktes, Kante, oder Fläche und nutzt den Löschen “Hotkey“ (X oder Del), so kann man auswählen was hiervon gelöscht werden soll:

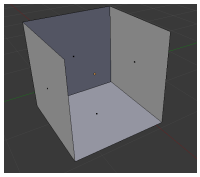
- Eckpunkte („Vetices“)
  - Hierbei werden insbesondere auch alle Kanten und Flächen gelöscht welche sich an dem Punkt befinden
- Kanten („Edges“)
  - Hierbei werden auch alle angrenzenden Flächen gelöscht
- Flächen („Faces“)
  - Entfernt alle Flächen und dazwischenliegende Punkte und Kanten, sofern diese von mehren ausgewählten Flächen eingeschlossen sind, oder wenn diese nicht von einer anderen Fläche „gehalten“ werden
- Nur Kanten und Flächen („Only Edges & Faces“)
  - Hier wird zunächst nur die Fläche entfernt, und danach alle übrig bleibenden Kanten, wodurch auch deren angrenzenden Flächen gelöscht werden
- Nur Flächen („Only Faces“)
  - Es werden wirklich nur die Flächen entfernt und alle Kanten und Punkte bleiben erhalten



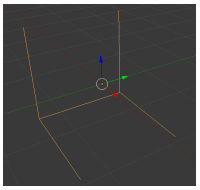
Resultat nach Löschen eines Vertex



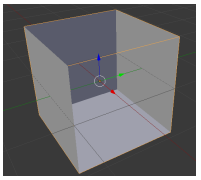
Resultat nach Löschen einer Kante



Resultat nach Löschen zweier angrenzenden Flächen



Resultat nach Löschen zweier angrenzenden Flächen



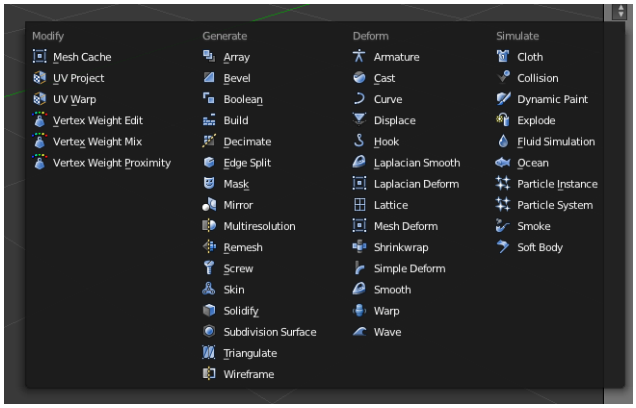
Resultat nach Löschen zweier angrenzenden Flächen

Es Folgen noch weitere Entfernmetdhoden, welche aber in dieser Dokumentation den einzuhalten Rahmen übersteigen würden und nur für spezielle Anwendungen Sinn haben.

2.4 Blender Modifier

Autor Nils Baumgartner

Blender bringt eine Vielzahl von Modifizierungen für Objekte mit neben den bisher genannten Werkzeugen, von der Generierung eines Spiegelbilds bis zur Simulation von Explosionen und Flüssigkeiten

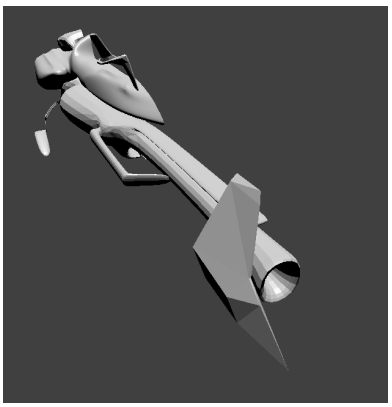


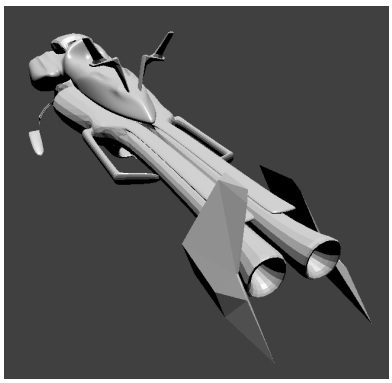
Liste aller 'Modifier'

Viele dieser Modifizierungen waren für unsere Aufgabe der Modellierung von Fahrzeugen und Umgebungsobjekten nicht relevant und hätten auch den Zeitlichen Rahmen des Praktikums überstiegen. Daher wird auch hier der Fokus auf die wesentlich relevanten Modifizierer („Modifier“) gerichtet.

Spiegeln - „Mirror“

Ein sehr wichtiger Modifizierer, welcher so gut wie bei allen größeren Modellen angewandt werden sollte, welche symmetrisch sind. Das Spiegeln kann an allen drei Hauptachsen angewandt werden, wodurch sich beim späteren „UV-Mapping“ die Arbeit die zugehörigen Flächen zu identifizieren lohnt. Diese Anwendung zeigte sich als besonders hilfreich bei der Modellierung der Fahrzeuge und allen anderen Umgebungsobjekte. Durch das spiegeln wird ein Großteil der Arbeit abgenommen und die Adäquatheit garantiert.

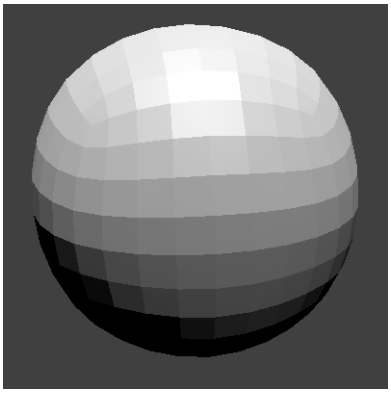

Eine modellierte Hälfte


Nach hinzufügen des 'Mirror Modifier'

Unterteilungsfläche - “Subdivision Surface“

Der "Modifier" „Subdivion Surface“ ist sehr mächtig, welcher in der Modellierung der Fahrzeuge häufig angewandt wurde. Hierbei werden wie beim Werkzeug „Subdivide“ die Komponenten in feinere Komponenten unterteilt. Diese Unterteilung bleibt aktiv, jedoch wird diese Unterteilung nicht direkt angewandt in dem Sinne, dass man die einzelnen Komponenten ansprechen kann. Dies bietet Vorteil, dass man eine Vorschau bekommt wie das Objekt nach dem anwenden aussehen wird, als auch beim UV-Mapping.

Würfel mit Iteration 1



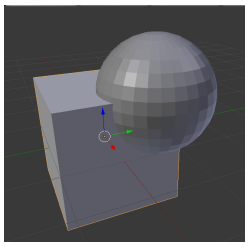
Würfel mit Iteration 3

Jedoch wird dies auf das gesamte Objekt angewandt, womit das modellieren von harte Kanten ein Problem wird. Um härtere Übergänge zu erzeugen kann man mehrere Flächen dazwischen legen, oder die Flächen seiger (kantiger) stellt. Zur Lösung dieses Problems helfen viele weitere Werkzeuge unter anderem auch das „Bever“ welches an dieser Stelle lediglich genannt werden sollte.

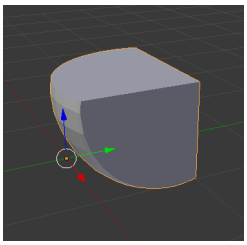
Verknüpfungen - “Boolean“

Bei dem „Modifier“ „Boolean“ werden zwei Objekte miteinander „verglichen“. Hierbei kann man das Augenmerk auf verschiedene Ziele richten, welche wiederum zu verschiedenen Ergebnissen führen.

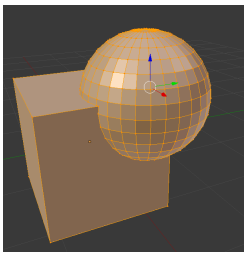
- “Intersect“ - Schnittmenge
  - Lässt von auf dem angewandten Objekt lediglich die Menge zurück welche in beiden Objekten vorhanden ist. (A && B)
- “Union“ - Vereinigung
  - Dieser Wert fügt an dem angewandten Objekt das Zielobjekt hinzu. Jedoch werden die Überschneidungen des Zielobjektes nicht mit übernommen, lediglich die Schnittpunkte. (A || B)
- “Difference“ - Unterschied
  - Hierbei wird das Zielobjekt aus dem angewandten Objekt ausgeschnitten. (A / B)



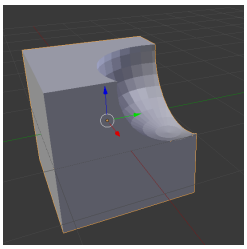
Ausgangslage



'Intersect'



'Union'



'Difference'

2.5 UV-Texturing

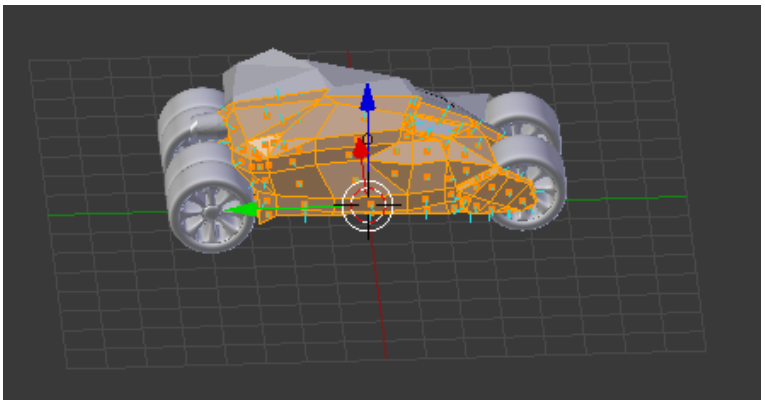
Autor Alexander Tessmer

UV-Texturing

Mit dem UV-Texturing ist es möglich ein 3D Objekt mit einem 2D Bild zu texturieren. Hierbei beschreibt ein UV-Koordinatensystem die Punkte der Textur. Jeder Punkt des Polygonobjektes bekommt dabei eine eindeutige Texturposition. Es wird eine UV-Map erstellt, sodass ein erstelltes 3D Objekt in Form eines Polygonnetzes anhand dieser UV-Map texturiert werden kann. Es können mehrere UV-Maps für jedes 3D Objekt erstellt werden. In diesem Fall kann man das 3D Objekt auch mit mehreren Texturen versehen. Dafür muss allerdings das Polygonnetz noch in Untergruppen zerteilt werden, da sonst ein *Export* nicht möglich ist. Die UV-Koordinaten werden für die *Faces* festgelegt, nicht für die *Vertices*. Ein *Vertice* der in zwei *Faces* vorhanden ist, kann daher geteilt werden und so unterschiedliche Koordinaten besitzen. Dadurch kann man *Faces* beliebig auf der Textur positionieren.

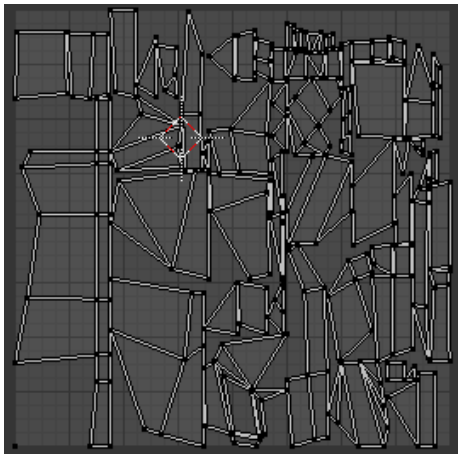
Blender Modelle müssen mit UV-Mapping texturiert sein, damit sie in Unity benutzt werden können.

Für das UV-Texturing braucht man eine UV-Map. Diese entsteht durch den *Unwrap* von einem *Mesh*



Mesh für den *Unwrap*

Der *Unwrap* soll dafür sorgen, dass das 3D Model auf 2D abgebildet werden kann. Dafür wird das Polygonnetz aufgefaltet und gegebenenfalls bei zu großen Winkeln abgeschnitten. Dies fällt besonders chaotisch aus, wenn das Model komplett vorhanden ist und nicht nur eine Hälfte die gespiegelt wird.



UV Map in Blender

Das Ergebnis ist nun wild verteilt. Soll man aber eine Textur mit unterschiedlichen Materialien anwenden, muss man nun noch herausfinden welches *Face* man gerade vor sich hat, um dieses dann richtig zu platzieren. Hierbei muss man leider feststellen, dass man die *Faces* nur ganz auswählen kann und nicht nur eine Ecke oder Kante. Sollte das *Face* also gespiegelt oder gedreht sein, ist es je nach Form des *Face* schwer dieses zu erkennen, solange man noch nicht gerendert hat. Glücklicherweise bleiben auch einige *Faces* zusammen und daher entstehen meist komplexere Formen, an denen man die Drehung oder Spiegelung erkennt. Dies sorgt dafür, dass man teilweise genau so viel Zeit aufwenden muss, wie für die Modellierung selbst. Man kann auch

absichtlich *Faces* überlappen lassen. So sorgt man dafür, dass auf beiden *Faces* exact die gleiche Textur erscheint. Bei Vorderseiten und Rückseiten ist dies praktisch, da man so zum Beispiel bei Holz die gleiche Maserung Vorne und Hinten erhält.

3. Animation

Autor: Ilhan Gören

Um dem Spiel etwas "Leben" einzuhauchen, wurden an einigen Stellen Spielcharaktere eingefügt, die mit Blender modelliert wurden. Zusätzlich wurden die Figuren animiert, um so dem Spiel mehr Authentizität zu verleihen. Die Animation von Objekten erwies sich doch etwas schwieriger als vorher angenommen. Durch zahlreiche Videos im Internet und einschlägiger Literatur konnte man sich sehr gut in die Thematik einarbeiten. In meiner Dokumentation führe ich die notwendigen Schritte auf, die wichtig für die Animation von Objekten ist. Dabei benutze ich bei meiner Ausarbeitung stets die Terminologie, die in Blender verwendet wird und versuche immer eine angemessene deutsche Übersetzung anzugeben.

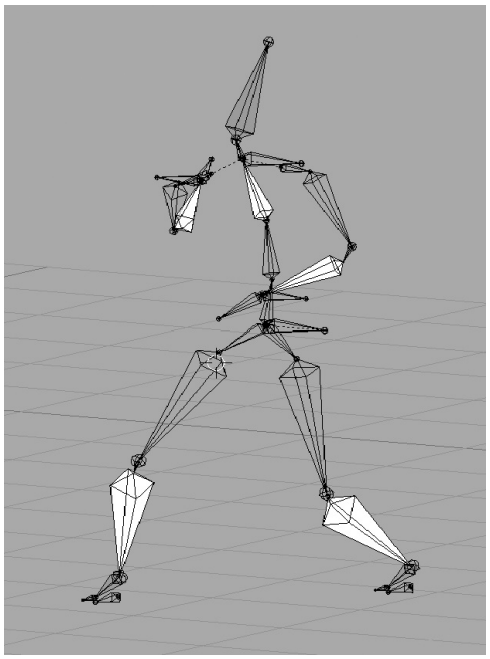
3.1 Rigging

Autor: Ilhan Gören

Rigging bezeichnet den Prozess ein Objekt so mit Funktionalität auszustatten das es nachher leicht animierbar ist. Es kommt aus dem (Puppen)Theater wo jede Puppe ein Gestell (Rig) hat an dem es aufgehängt ist. Das bewegen der einzelnen Schnüre bewegt dann die Puppe. So ähnlich funktioniert das auch bei Animationen, nur das keine Schnüre und Gestelle gebraucht werden, stattdessen werden Knochen (bones) verwendet die man in die Objekte an die gewünschten Positionen setzt, mit dem Model verbindet und dann so bewegen kann wie man will. Deswegen gibt es neben dem Bearbeitungsmodus für Skelette auch einen Posiermodus, in dem das Skelett und damit die Figur in eine bestimmte Pose gebracht werden kann, ohne die ursprüngliche Position der Knochen und des Models zu verlieren.

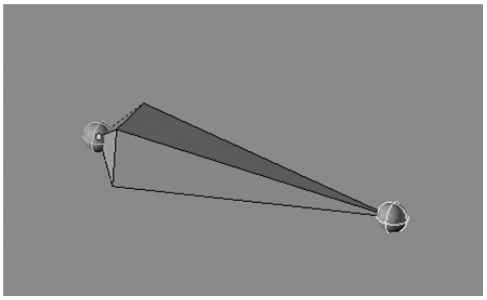
Hinzufügen eines Armatures

Armature-Objekte sind skelettartige Strukturen, die aus verbundenen Knochen (sogenannten Bones) bestehen. Die Knochen in einem Armaure können dazu verwendet werden, um andere Objekte zu verformen und ist für die Charakter Animation unumgänglich. Wie oben beschrieben, ähneln Armatures oft einem Skelett (siehe folgendes Bild).



Armature in Blender.

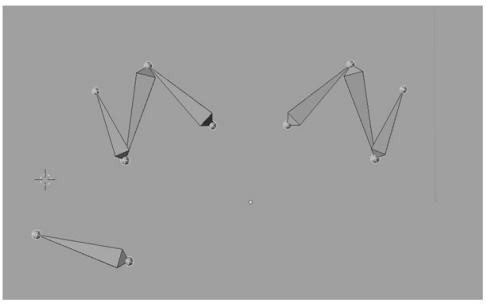
Um ein Armature hinzuzufügen, geht man in das Space-Menü und wählt Add->Armature. Ein Armature mit einem einzelnen Knochen wird dort in die Szene eingefügt, wo sich der 3d-Cursor befindet. Ein Bone ähnelt dabei einem Oktahedron mit einem dicken und einem dünnem Ende. An jedem Ende befindet sich eine Kugel. Dabei steht die Kugel am dicken Ende für die Wurzel (in Blender root genannt) und die Kugel am dünnen Ende steht für die Spitze (in Blender tip genannt). Diese können jeweils separat angewählt werden.



Bone in Blender.

Richtig interessant wird es natürlich erst, wenn man Bones miteinander verbindet, sodass eine Hierarchie von Knochen ensteht, die sich gegenseitig beeinflussen können. Um ein Bone einem Armature hinzuzufügen nutzt man im Edit Mode die spacebar toolbox und wählt Add->Bone. Das Bone wird an die Position des 3d-Cursors hinzugefügt. Knochen können im Edit-Mode mittels der Extrude-Funktion (in Blender mittels der E-Taste) hinzugefügt werden. Der Teil des Knochens, von der aus extrudiert wird, bestimmt das Verhalten und die Beziehung des neu extrudierten Knochens. Extruding von der Spitze: Mittels der rechten Maustaste wählt man die Spitze des Knochen und drückt dabei die E-Taste, um zu extrudieren. Der neue Knochen wird automatisch ein Kind des Knochen, von der aus extrudiert wurde und wird automatisch mit dem ursprünglichen Knochen verbunden Extrudieren von der Wurzel: Mittels der rechten Maustaste wählt man die Wurzel des Knochen und drückt dabei die E-Taste, um zu extrudieren. Knochen. die von einer Wurzel extrudiert wird, sind keine Kindknochen mehr und werden nicht mit dem ursprünglichen Knochen verbunden. Es entspricht somit dem Hinzufügen eines neuen Knochens.

So lässt sich schrittweise ein Skelett erstellen, das später dem Objekt hinzugefügt wird.



Nach dem Rigging schließt sich das Skinning an, das im folgenden aufgeführt wird.

3.2 Skinning

Autor: Ilhan Gören

Armaturen und Knochen sind ziemlich interessant, aber richtig interessant wird es erst, wenn sich Bewegungen am Skelett sich unmittelbar auf das Mesh auswirken. Im Pose-Modus kann man einzelne Knochen auswählen und Transformationen wie Rotation, Drehung und Skalierung ausführen. Man stellt jedoch fest, dass sich diese Transformationen nicht auf das Mesh auswirken. Was zu tun ist, ist das Binden der Vertices vom Mesh zu bestimmten Knochen innerhalb des Mesh. Dieser Bindungsvorgang wird allgemein als Skinning bezeichnet. In Blender existieren dazu zwei Wege: Envelopes und Vertex Groups. Es handelt sich hierbei um sogenannte Armature-Modifier. Im folgenden wird bloß das Prinzip des Envelopes aufgeführt, da dies noch sehr einfach von statten geht. Die Methode mit Vertex Groups ist etwas komplexer. Dazu gibt es im Internet zahlreiche Videos, das schrittweise anhand eines Beispiels die Methode erklärt

Skinning mit Envelopes

Envelopes ist der schnellste und einfachste Weg, die Vertices vom Mesh mit dem Armature zu verbinden. Diese Vorgehensweise habe ich auch bei meinem Objekt genutzt. Da es gleichzeitig sehr viele Objekte in einer Szene geben kann, gibt es in Blender eine Gruppierungsfunktion mit der ausgewählte, z.B. zusammenhängende, Objekte in Gruppen zusammengefasst werden können oder es gibt Fälle wo zur Strukturierung klare Hierachien benötigt werden. Hierfür bietet Blender die Möglichkeit, Objekte in eine "Eltern-Kind-Beziehung" zu setzen (parenting). Das Verschieben, Rotieren oder die Größenänderung des übergeordneten Objekts (parent) verändert dann auch die untergeordneten Objekte (children).

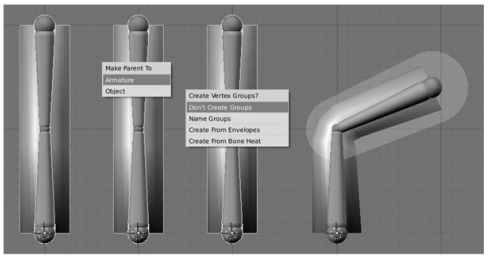
Folgende Schritte wurden unternommen, um das Mesh durch das Skelett zu kontrollieren:

1. Im Objektmodus oder Pose-Modus wählt man das Mesh mit der rechten Maustaste aus und fügt mittels der Tastenkombination Shift + Rechtsklick das Armature hinzu

2. Durch drücken der Tastenkombination Strg + P wird das armature als parent gesetzt. Eine Auswahl von Optionen erscheint: man wählt innerhalb dieser Optionen armature. Nachdem man nun armature ausgewählt hat erscheint ein weiteres Fenster mit dem Titel "Create Vertex Groups?". Für dieses Beispiel wählt man "Don't Create Groups".

3. Nach diesen Schritten sollte man nun im Pose-Modus das Skelett bewegen können und mit ihm auch das Mesh

Das ist der grundlegende Arbeitsablauf für Envelopes. Das folgende Bild veranschaulicht dies nochmal



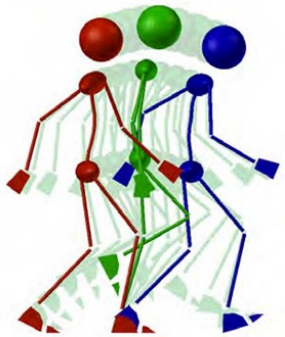
3.3 Animation in Blender

Autor: Ilhan Gören

Nachdem man ein Grundwissen über Modellierungstechniken besitzt, zeige ich im Folgenden, wie man sein fertiges Objekt zum Animieren bringt und es so zum "Leben" erweckt. Wir starten mit der Animation einfacher Objekte. Dieses Kapitel dient dazu, sich mit den wichtigsten Werkzeugen in Blender vertraut zu machen, die zum Animieren wichtig sind

Keyframing mit der Timeline

Um mit der Animation anzufangen, sollte man sich mit Keyframes (=Schlüsselbilder) vertraut machen. Zu Zeiten von Daumenkinos mussten Animatoren jedes Frame einer Szene einzeln zeichnen. Dies erfordert natürlich sehr viel Zeit für komplexe Szenen. Diese Aufgabe übernimmt heute der Computer. Man markiert lediglich die wichtigsten Keyframes und überlässt die Arbeit dem Computer, der zwischen den Keyframes interpoliert, um so den Eindruck einer flüssigen Bewegung zu vermitteln. Das folgende Bild gibt die Idee dahinter sehr gut wieder. Die farbigen Figuren stehen hier für die Keyframes, die vorher definiert wurden.



Verdeutlichen kann man sich dies anhand eines einfachen Beispiels:

1. In der Default-Szene lösche den Cube und füge den Affen ein

2. rotiere den Affen entlang der x-Achse um 90 Grad (drücke R, drücke X, tippe 90 ein und bestätige mit Enter)

3. Unter dem 3D-View befindet sich ein Timeline-Fenster (siehe nachfolgendes Bild). Die vertikale grüne Linie befindet sich hierbei auf Position 1, dass den ersten Frame darstellt. Durch Linksklicken innerhalb der Timeline kann man die Position der Linie ändern. Die Animation sollte stets mit Frame 1 beginnen

4. Nun wähle mit Rechtsklick den Affen aus und drücke im 3D-View die I-Taste, um ein Keyframe anzulegen. Eine Liste mit den Hauptauswahlpunkten Location, Rotation, Scaling, LocRot, und LocRotScale erscheint (siehe nachfolgendes Bild). Wähle LocRot um die aktuelle Position (Loc) des Objektes und die Rotationsrichtung (Rot) festzulegen

5. Um den Affen nun zu bewegen, setze den Slider der Timeline auf Frame 50. Blenders Standardeinstellung ist 25 FPS(Frames per second). 50 Frames entsprechen somit 2 Sekunden innerhalb eines Films

6. Nun bewege den Affen nach vorne entlang der Blickrichtung

7. Drücke erneut die I-Taste und wähle LocRot, um ein erneutes Schlüsselbild zu erzeugen

Jedes Keyframe das erzeugt wurde, ist auf der Timeline mit einer vertikalen gelben Linie gekennzeichnet. Wenn man nun als EndFrame 50 eingibt und die Tastenkombination Alt + A drückt, kann man sich die Animation in einer Schleife anschauen. Zum Stoppen reicht ein Drücken auf die Escape-Taste.

Automatisches Keyframing

In Blender besteht die Möglichkeit der automatischen Erstellung von Keyframes. Unterhalb der Timeline befindet sich ein Record-Button. Durch Drücken auf diesen Button aktiviert man das automatische Keyframing. Das bedeutet, dass jede Bewegung, die man an sein Objekt ausführt, automatisch ein Keyframe angelegt wird, sodass man nicht mehr manuell mit der I-Taste ein Keyframe erzeugen muss. Es sollte bloß sichergestellt werden, dass für jede neue Bewegung der Slider in der Timeline auf ein neues Frame zeigt

3.4 Vorwärts-und inverse Kinematik

Autor: Ilhan Gören

Animiert man ein Objekt mit der Hilfe eines Skeletts, wird zwischen zwei verschiedenen Animationsmethoden unterschieden: der Vorwärts-Kinematik (FK) auch direkte Kinematik genannt, und der inversen Kinematik (IK). Eine Mischform zwischen diesen beiden Methoden ist die automatische, inverse Kinematik.

Vorwärts-Kinematik

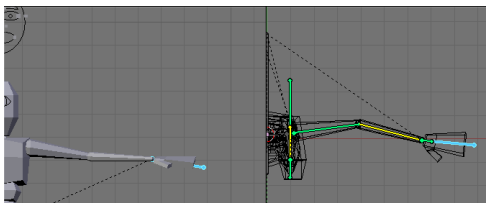
Bei der Vorwärts-Kinematik bestimmt der Parent eines Objektes die Bewegung seines Childs. Das ist in Blender der Normalfall. Probleme treten dann auf, wenn man in einer Skelett-Animation eine Kette von verbundenen Knochen hat, und das Ende der Kette einen bestimmten Punkt im Raum erreichen soll. Nehmen wir als Beispiel die Bewegung einer Hand. Es ist leicht anzugeben zu welchem Zeitpunkt sich die Hand an welcher Stelle befinden soll. Es ist aber schwer, wenn man zunächst den Oberarm, dann den Unterarm und zum Schluss die Hand selbst bewegen muss.

Inverse Kinematik

Bei der inversen Kinematik steuert das Ende einer Bone-Kette die Bewegung (siehe nachfolgendes Bild). In dem Beispiel wird nur das Zielobjekt des äußersten Bones bewegt, die übrigen Bones folgen nach. Man nennt den letzten Bone der Kette den IK Solver, da sich die Bewegung der Bones als Lösung einer mathematischen Gleichung ergibt. Der IK Solver versucht ein Ziel (Target) zu erreichen. Die Länge der Kette - also wie viele Bones der Bewegung folgen sollen - lässt sich einstellen. So will man in der Regel wohl nicht, dass sich der Rumpf des Characters bewegt, wenn sich die Hand bewegt. Die Stellung der übrigen Bones ist durch die Bewegung nicht festgelegt, es gibt eine Menge Möglichkeiten, die zu einer bestimmten Handposition führen. Man kann daher die Freiheitsgrade der folgenden Bones beschränken (Degrees of Freedom = DoF). Trotz geschickter Kombination von Freiheitsgraden und komplizierten Skelettaufbauten kommt man manchmal mit einer Kombination von IK und FK schneller zum Ziel. Gerade Armbewegungen werden häufig mit FK animiert. Dann hilft Automatic IK.

Auto IK

Auto IK ist die Abkürzung für "Automatische inverse Kinematik". Mit Auto IK ist jeweils der Bone, den Sie gerade bewegen, der IK-Solver. Die Nützlichkeit dieses Features ist allerdings begrenzt, es funktioniert nur gut, wenn es keinen weiteren Solver in der Kette gibt, und wenn die Kette nicht mit dem gesamten Rig verbunden ist.



Eine winkende Figur. Animiert mit IK.

3.5 Quellen

Quellenangaben:

Brito, Allan: Blender 3D 2.49 Architecture, Buildings, and Scenery. Packt Publishing. 2010

Chronister, James: Blender Basics. 2006

Fisher, Gordon C.: Blender 3D Basics Beginner's Guide.Packt Publishing. 2012

Flavell, Lance: Beginning Blender. Apress. 2010

Hess, Roland: The Essential Blender. No Starch Press. 2007

http://de.wikibooks.org/wiki/Blender_Dokumentation

Powell, Aaron W.: Blender 2.5 Lighting and Rendering. Packt Publishing. 2010

Simonds, Ben: Blender Master Class. No Starch Press. 2013

Williamson, Jonathan: Character Development in Blender 2.5. Course Technology. 2012

4. Leveldesign

Das Team für Leveldesign bestand aus Alexander Altemöller, Michael Kaufmann und Dennis Altenhoff.

Die Aufgabe der Gruppe bestand in der Erstellung von einigen Landschaften mit einer Rennstrecke.

4.1 Einführung in Unity

Autor: Michael Kaufmann

Was ist *Unity*?

Unity ist eine umfangreiche Entwicklungsumgebung um hochwertige 2D und 3D Spiele zu produzieren.

Die Benutzeroberfläche

Unitys Benutzeroberfläche

Unity ist grob eingeteilt in 4 unterschiedliche Teilbereiche.

- Die Szene, hier entsteht das Spiel.
- Die Hierarchie, hier werden alle *GameObjects* die man in der Szene gesetzt hat aufgezeigt.
- Der Inspektor, hier werden Details über ausgewählte *GameObjects* angezeigt.
- Das Projekt, Hier werden alle importierten *Assets* und *Prefabs* aufgelistet.

Prefabs sind quasie abgespeicherte Schablonen, welche man an jedem beliebigen Ort in der Szene drucken kann. Ein *Asset* hingegen ist eher etwas wie der Überordner für ein *Prefab*. Quasie eine Auswahl an Farben und Formen mit denen die Schablone drucken soll. *Assets* können in dem in Unity integrierten *Asset-Store* erstanden werden. Manche werden kostenlos angeboten, andere kosten hunderte Euro.

Unity vereinfacht viele ansonsten komplizierte Aktionen. Zum Beispiel beim Transformieren, Rotieren oder Skalieren muss man nicht umständlich Matrizen miteinander multiplizieren. Sowas kann in *Unity* unkompliziert per Mausklick erledigt werden.

4.2 Wüstenstrecke

Autor: Dennis Altenhoff

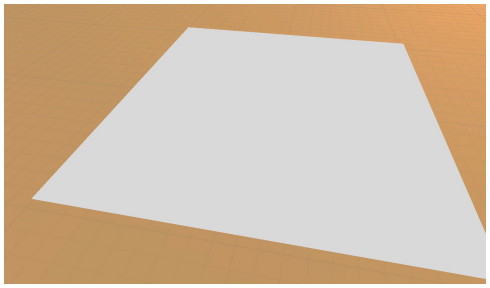
Die Idee

Das Ziel der Wüstenstrecke war es, eine möglichst authentische Landschaft zu gestalten.

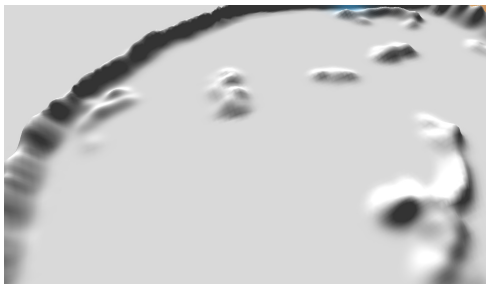
Das bedeutete, dass eine sandige Landschaft zu erstellen war, die hauptsächlich eben, aber stellenweise auch hügelig ist. Zudem war eine Begrenzung aus Hügeln notwendig, damit die Spieler nicht vom Terrain fallen können und eine vertretbare Rundenlänge zustande kam.

Die Umsetzung

Zu allererst wurde ein *"Terrain-GameObject"* eingefügt. Dieses wurde mit einem entsprechenden *Terrain-Tool* so bearbeitet, dass einige Hügel entstanden. Zudem wurde eine von Unity bereitgestellte *Skybox* hinzugefügt.

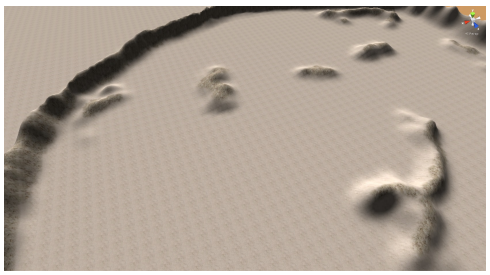


Das Terrain in Ursprungsform.



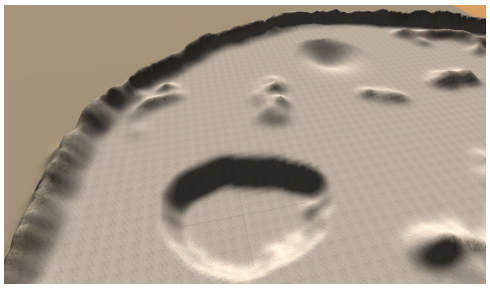
Das Terrain mit Hügeln und Begrenzung.

Anschließend wurde der ebene Boden mit einer Sand-Textur, die Hügel mit einer Sandstein-Textur versehen.

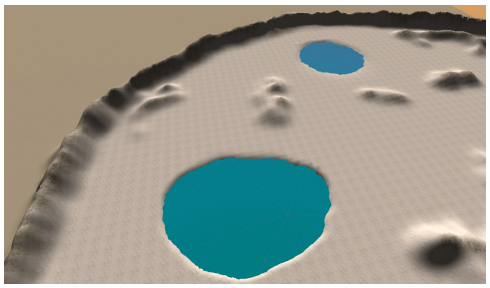


Das Terrain mit Textur.

Um die Umgebung noch authentischer und lebendiger zu gestalten wurde das Gebiet um Oasengebiete, einzelne von mehr Gras umgebene Wasserflächen, erweitert. Hierfür wurden Vertiefungen im Terrain erzeugt und ein *"Wasser-GameObject"* hinzugefügt.

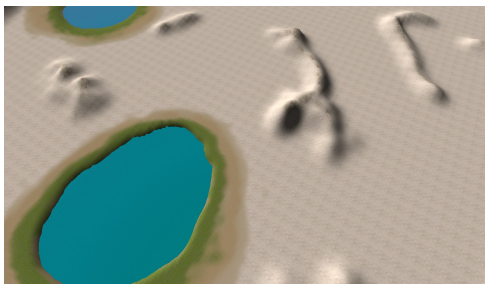


Eine Vertiefung für den See.



Der See mit eingefügtem Wasser-GameObject.

Im Folgenden wurde die Umgebung mit verschiedenen Texturen versehen, die mit steigender Entfernung zum Wasser immer weniger Gras aufwiesen und möglichst weich in eine Sand-Textur übergingen.

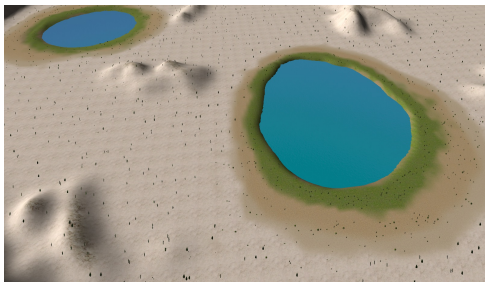


Die fertige Oase (ohne Vegetation).

Danach folgte das Erstellen der Vegetation. Hier wurde darauf geachtet, dass, um möglichst große Authentizität zu erreichen, Kakteen verwendet werden und viele verschiedene Arten auf der Karte vorhanden sind.

Die verschiedenen Arten von Kakteen wurden möglichst authentisch auf dem Gebiet verteilt, sodass überall eine angemessene Dichte vorhanden war. Dies geschah über ein in Unity integriertes *Terrain-Tool*, mit dem man wie mit einem Pinsel mehrere Bäume auf einmal auf der Karte platzieren kann. Da diese Funktion aber immer nur eine einzelne Art und zu viele Kakteen auf einmal platzierte, war es notwendig, einen Teil der erstellten Kakteen der benutzten Art mithilfe eines anderen, speziell zur Entfernung von Bäumen vorgesehenen, *Terrain-Tools* wieder zu entfernen und weitere Kakteen anderer Arten hinzufügen und das Vorgehen solange zu wiederholen, bis eine möglichst realistische Umwelt geschaffen wurde. Ebenfalls wurde darauf geachtet, dass die Kakteen, die näher am Wasser stehen als andere, mehr Blüten tragen.

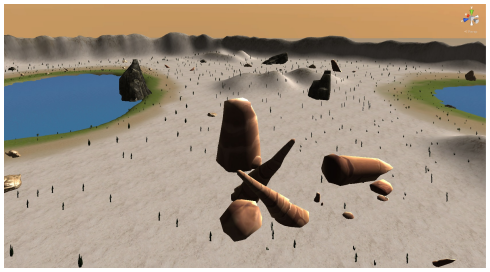
Vegetations-Beispiel.



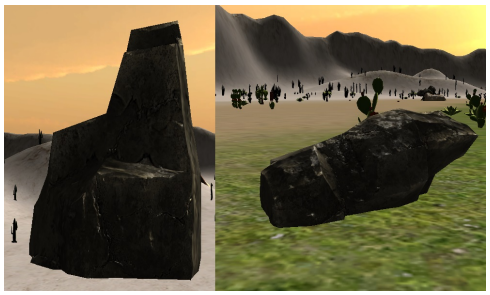
Vegetations-Beispiel.

Vegetations-Beispiel.

Nachdem nun die Vegetation platziert wurde, wurde das Gebiet um verschiedene Steine ergänzt. Da aber die Anzahl an Stein-Modellen begrenzt war, musste mit Hilfe von Transformationen der einzelnen Objekte der Eindruck erweckt werden, dass es sich, trotz der Tatsache, dass es dieselben Grundmodelle waren, um verschiedene Modelle handelt.

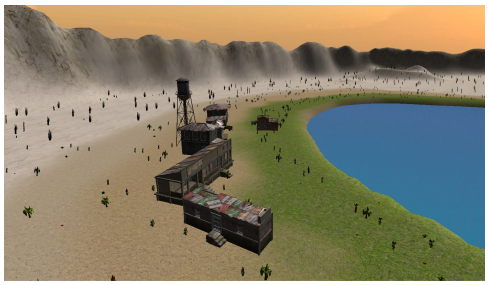


Einige Steine.



Zwei eigentlich gleiche Steine im Vergleich.

Zuletzt wurde mit Hilfe des *Asset-Stores* eine Stadt eingefügt, die sich nahe einer Oase "angesiedelt" hat. Damit wurde die Landschaft noch lebendiger.



Eine Siedlung nahe einer der Oasen.

Einfügen der Straße

Nachdem die Umgebung vorhanden war, wurde mithilfe eines *Package* aus dem *Asset-Store* (*EasyRoads3D Free*) eine Straße hinzugefügt und mit von der *Blender*-Gruppe erstellten Rennstrecken-Elementen versehen.

Reifenstapel und Startampel aus Blender in Unity.

Leitplanken aus Blender in Unity.

Letzte Details

Zum Schluss wurden Partikel-Systeme hinzugefügt, die für Staubwehen auf der Strecke sorgen, um so die Authentizität der Strecke weiter zu steigern. Ein Beispiel dafür ist, bei genauerem Hinsehen, in diesem Video vorhanden:

4.3 Bergstrecke

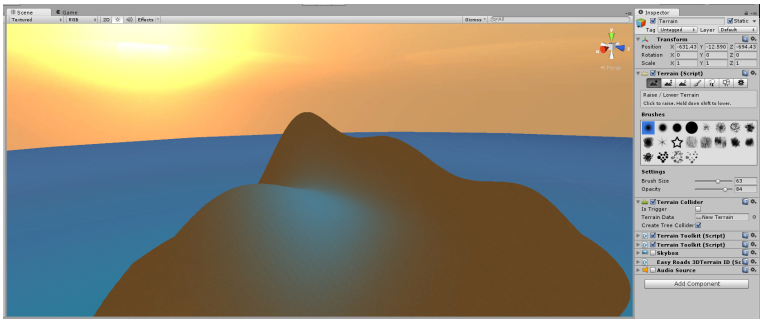
Autor: Alexander Altemöller

Ein Ausschnitt aus der Bergstrecke

Bei der Modellierung dieser Strecke hatten wir das Ziel, eine möglichst realistische aber auch gleichzeitig sehr abwechslungsreiche Berglandschaft zu modellieren.

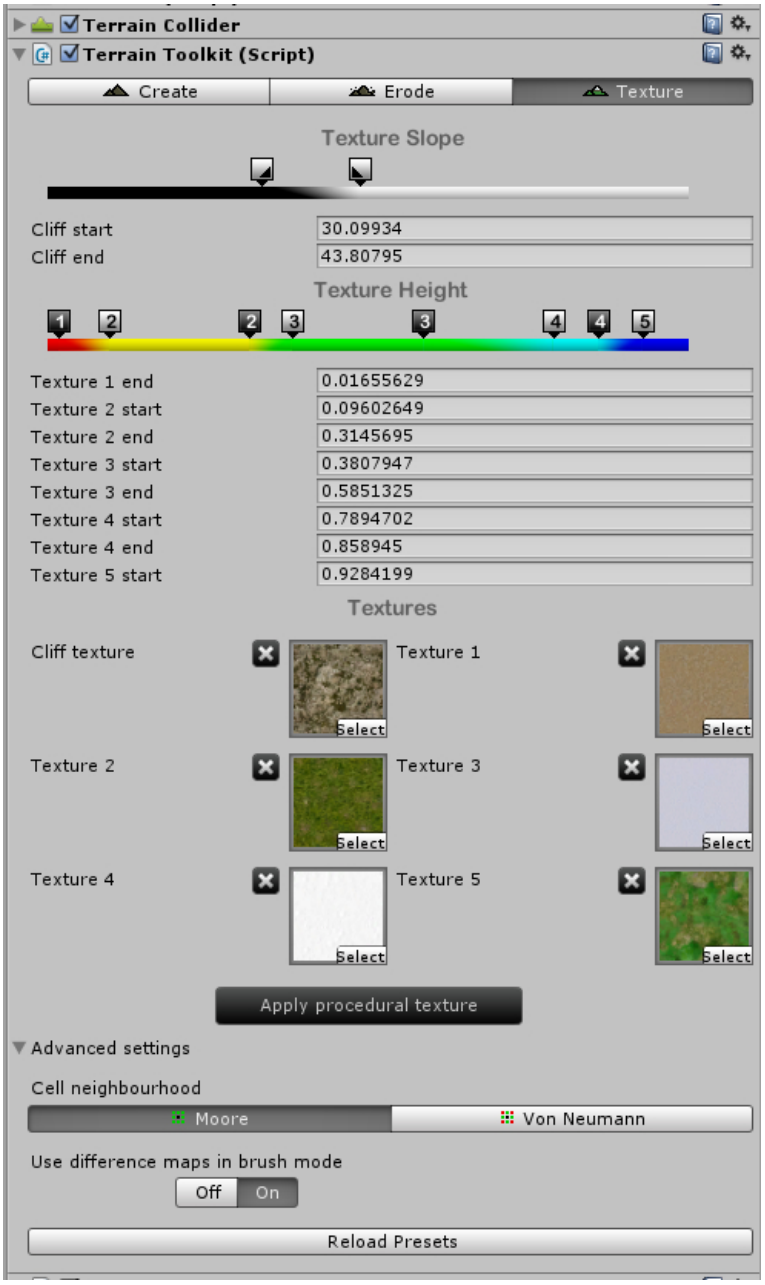
Erzeugen und Texturieren des Gebirges

Dafür wurde zunächst mit dem in Unity3D integrierten *Terrain-Tool* eine Gebirgslandschaft erschaffen. Dies geschieht sehr intuitiv und ist auch für Laien nach einer kurzen Einarbeitungszeit mittels eines „*Brushes*“ leicht möglich.



untexturiertes Terrain

Als nächstes müssen nun passende Texturen auf das fertige Terrain gelegt werden. Unity3D liefert von sich aus einige niedrig aufgelöste Standardtexturen mit, deutlich schönere und auch hochaufgelöste lassen sich im *Asset-Store* finden. Desweiteren bietet Unity3D mittels eines importierbaren Tools die Möglichkeit automatisch Höhen und Tiefen zu erkennen. Sobald man Unity3D beispielsweise eine Textur für die oberste Höhe übergibt (beispielsweise eine Textur für Berggipfel) und diese Textur automatisch anwenden lässt, wird nun jeder Berggipfel mit dieser Textur versehen. Dasselbe funktioniert vollkommen analog für Absenkungen, sodass man beispielsweise eine Sandtextur für den Übergang zum Meer wählen kann.



Das importiert Terrain-Toolkit

Von diesem sehr nützlichem Tool haben wir jedoch bei dieser Strecke keinen Gebrauch gemacht, da ansonsten jeder Berggipfel, jedes Tal, jeder Hügel und jeder Meeresübergang absolut identisch texturiert worden wären, was vielleicht beim ersten Blick nicht stark auffällt, aber natürlich im Vergleich mit der Realität absolut unrealistisch ist, da es immer leichte Differenzen in der Natur gibt. Diese Differenzen so gut wie möglich zu simulieren, um ein möglichst realistisches Spielerlebnis zu bilden, war eines unserer primären Ziele, weswegen wir bei den Texturen auch zu dem Brush-Tool gegriffen haben, und unser Terrain „per Hand“ Stück für Stück texturiert haben.

Objektplatzierungen

Nachdem wir also nun unsere fertig texturierte Berglandschaft vor uns haben, können wir einzelne Objekte platzieren. Hierzu gehört beispielsweise Wasser, Bäume, Felsen oder Straßenlaternen.

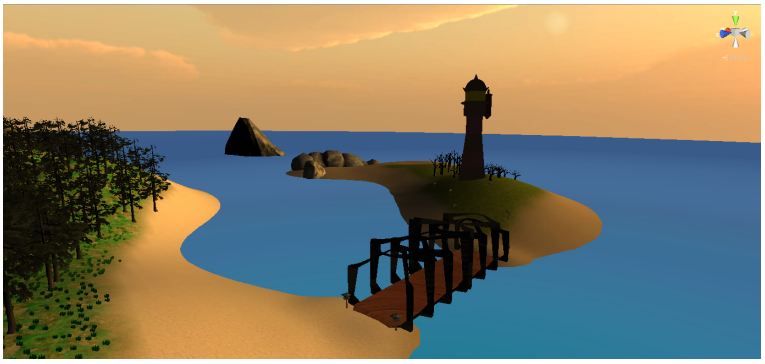
Im Bezug auf das platzieren von Bäumen und Gräsern liefert Unity3D auch die Funktion mit, mehrere Bäume gleichzeitig mit Hilfe des Brush-Tools zu platzieren. Dies geschieht, indem man als entsprechendes Baumobjekt (bzw. Grasobjekt) das passende Prefab auswählt, mit welchem das Brush-Tool verwendet werden soll. Hierbei ist darauf zu achten, dass man für die Baumobjekte (und generell für jedes andere Objekt) vorher entsprechende *Collider* definiert, sodass auch eine Kollision mit den Bäumen stattfinden kann, und der Spieler nicht einfach hindurch fahren kann. Auf die Funktionalität der Collider wird in diesem Teil der Dokumentation nicht näher eingegangen und wir verweisen hier auf die Fahrphysik, welche sich sehr intensiv mit diesem Thema auseinander gesetzt hat.

Für die Platzierung von Straßenlaternen empfiehlt es sich, im Vorraus ein Prefab zu erstellen, in welchem die Beleuchtungsinformationen für jede Laterne abgespeichert werden. Unter Umständen kann eine Straßenlaterne mehrere Lichtquellen erhalten, damit die Beleuchtung realistischer simuliert wird. Speichert man nun das Laternenobjekt zusammen mit den beispielsweise vier Lichtquellen in einem Prefab ab, müssen nicht für jede Laterne individuell vier Lichtquellen positioniert werden, da die Lichtquellen zusammen mit den Positionsinformationen im Prefab abgespeichert sind.

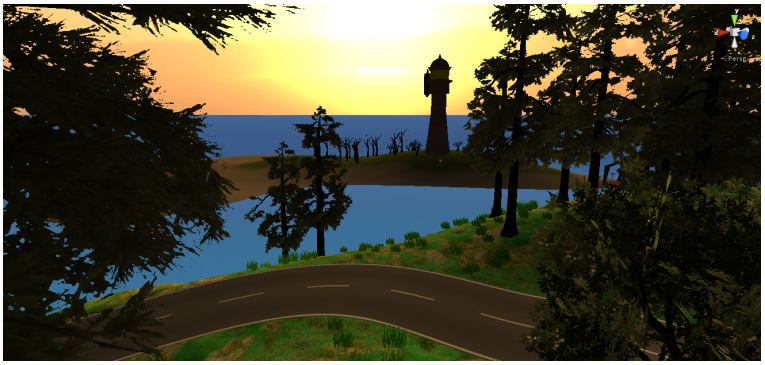
Mehrere platzierte Prefabs der Laternen in der Szene.
Die vier Lichtquellen pro Laterne mussten nun nichtmehr manuell positioniert werden.

Der letzte Feinschliff

Desweiteren war es uns bei der Geländegestaltung auch wichtig Orte einzubauen, welche nicht zur eigentlichen Strecke gehören, aber trotzdem befahrbar sind. Denn welche Insel besteht schon nur aus einer Rennstrecke und ist ansonsten komplett unbesiedelt? Aus diesem Grund platzierten wir beispielsweise noch eine kleine Insel mit einem Leuchtturm, welche über eine Brücke erreichbar ist, neben der Hauptinsel.



Die Insel abseits der eigentlichen Strecke



Ein weiterer Blick auf die Insel mit dem Leuchtturm

Nach dem fertigen Modellieren der Szene empfiehlt sich Lightmapping, um sowohl optischen als auch Performanceproblemen vorzubeugen.

4.4 Vulkanstrecke

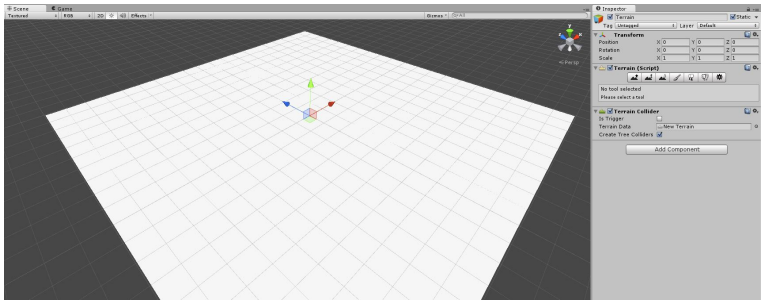
Autor: Michael Kaufmann

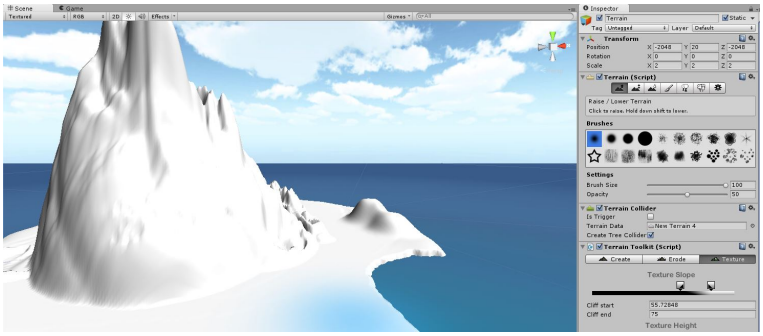
Die Vulkanstrecke

Ausschnitt aus der Vulkanstrecke

Beim Modellieren der Vulkanstrecke war es uns wichtig eine Strecke mit tropischer Thematik, hohem Detailreichtum und einer Lernkurve zu schaffe welche risikoreiche Fahrmanöver belohnt.

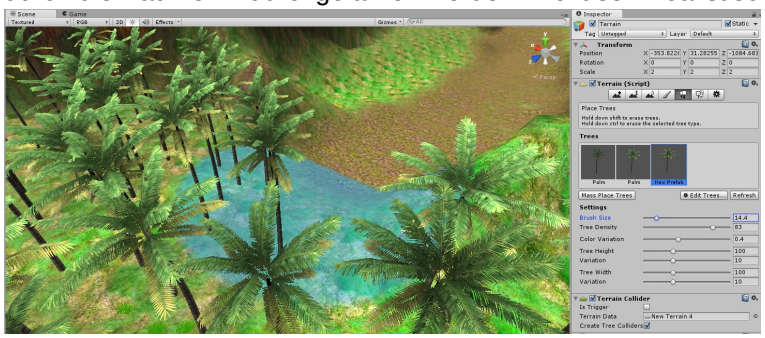
Erzeugen der Grundtextur

Zu Erst wird ein *Terrain* Objekt erstellt. Dies ist ein in Unity3d integriertes *Game Object* was unkompliziert und schnell erstellt werden kann. Ein *Terrain* Objekt ist nur zu einer Seite hin sichtbar. Ein standart Unity3d *Terrain* Objekt 

Dieses wird nun mit Hilfe der in Unity3d integrierten *Tools* so formverändert, dass ein Vulkan in der Mitte entsteht. Dies geschieht mit Hilfe der *Brush*-Funktion welche unkompliziert per Mausklick das *Terrain* an der ausgewählten Stelle anhebt oder auch absenkt. Wird nun ein Wasser Objekt erstellt, werden die abgesenkten Stellen mit diesem Wasser gefüllt und schon sieht das ehemals quadratische *Terrain* wie eine rundliche Insel aus. Die Grundtextur ist nun fertig 

Im nächsten Schritt wird das *Terrain* mit Hilfe des *Terrain Toolkit*, einem Skript aus dem *Asset Store* eingefärbt. Dieses kann an das *Terrain* angehängt werden und dann jenem mit Hilfe der intuitiven Benutzeroberfläche die gewünschte Grundtextur verpassen. Benutzeroberfläche des *Terrain Toolkit*

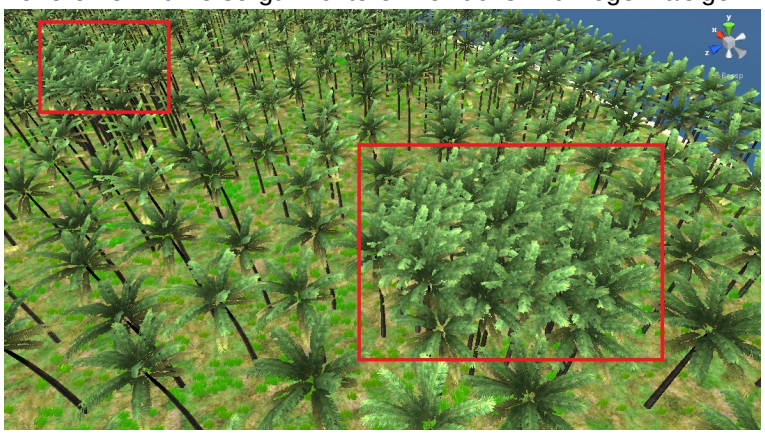
Wie bei der Entstehung des Vulkans und den anderen Formveränderungen der Insel, wurde nun wieder zur *Brush*-Funktion gegriffen um dem an sonsten sehr sich wiederholenden und unrealistischen Texturen des *Terrains* mehr Abwechslung und Vielfalt zu geben, und somit ein realistischeres Aussehen zu erwirken. Des weiteren wurde mit Hilfe der *Brush*-Funktion Palmen und Grasbewuchs auf das Terrain gesetzt um die Umgebung zu beleben. Hierzu muss nur vorher die gewünschte Baum oder Gras Textur, und der gewünschte Füllegrad, die so genannte *Opacity* ausgewählt, und dann auf die gewünschte Stelle des *Terrains* "gemalt" werden. Wichtig, der Ausgewählten Baumtextur vorher *Collider*, also "Bewegungsblocker" geben, sonst kann am Ende durch die Bäume hindurchgefahren werden. Nicht sehr realistisch. Palmen und eine neue Bodentextur



Umgebungsobjekte und Details hinzufügen

Nachdem das "Grundgerüst" nun steht, wird sich mit den Details befasst. Ein normirter Baumabstand von 1,5 Meter zu jedem anderen Baum fällt sofort als unrealistisch auf, und ist demnach unerwünscht. Desshalb wird wieder zur *Brush*funktion greifen, und den Wald an einigen Stellen ausdünnen oder auch dichter machen schafft dort Abhilfe und sorgt für unregelmäßigen Bewuchs. Auserdem verhindern wohl platzierte dichte Stellen in der Vegetation ein zu einfaches Abkürzen duch den Wald, wodurch dem Spieler mehr abverlangt wird und für eine

höhere Lernkurve sorgt. Dichterer Bewuchs in unregelmäßigen Abständen



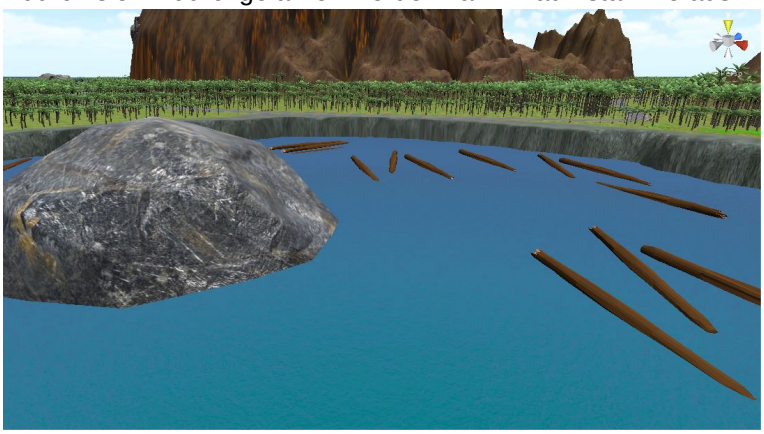
Die Straße

Im nächsten Schritt wurde sich nun mit der Strecke an sich befasst. Für die Fahrbahn sorgt hier wieder ein *Asset* aus dem *Asset-Store*, die *free* Version von *EasyRoads3D*, ist wie der Name schon vermuten lässt eine kostenlose Testversion eines *Tools* um realistisch wirkende Straßen auf ein beliebiges *Terrain* zu setzen. Trotz das nicht alle Funktionen in der Testversion zur Verfügung stehen ist *EasyRoads3D* ein leicht zu bedienendes Wekzeug um eine realistische Straße durch den Wald zu ziehen. Teil der Strecke welche mit *EasyRoads3D* modelliert wurde

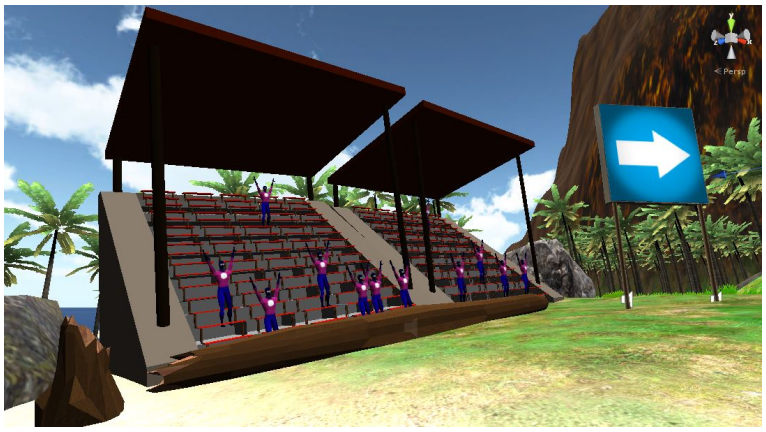
Abseits der Straße

Da wir nun wissen woher die Straße verläuft können wir uns im nächsten Schritt mit der Streckendekoration befassen, und der bisher trotz der Vegetation noch sehr "steril" wirkenden Strecke Leben einhauchen. Ein paar kleinere Objekte können in *Unity* selber modelliert werden. Für den Rest werden mit Hilfe der *Blender* Gruppe und des *Asset Stores* Streckenobjekte wie zum Beispiel die Ampel am Start, Steine oder auch die Schiffe importiert und mittels *Drag and Drop* in die Szene eingefügt. Hierbei sollte darauf geachtet werden, dass den Objekten aus Blender vorher noch *Collider* angehängt werden, da sonst wie auch bei den Bäumen das Problem

besteht dass später an sonsten durch sie hindurchgefahren werden kann. Baumstämme aus *Blender* und ein Meteorit aus dem *Asset-Store*



Für diesen Tunnel wurden Baumstämme aus *Blender*, Steine aus dem *Asset Store* und in *Unity* selber modellierte Laternen verwendet

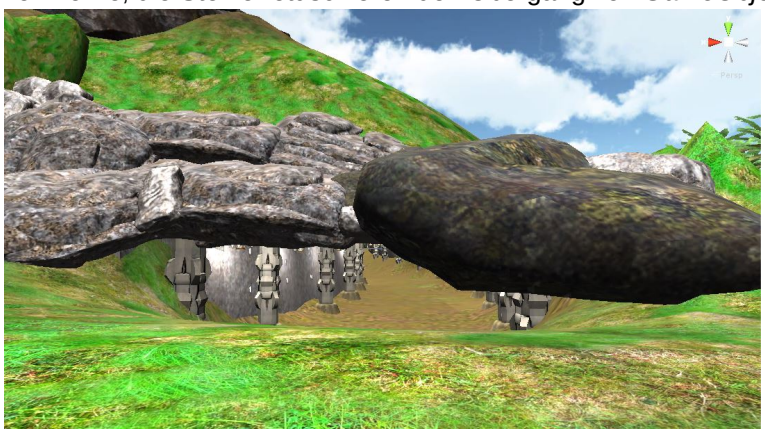


Tribüne und Zuschauer aus *Blender*, das Schild in *Unity* modelliert

Die Abkürzung durch die Erde

Zu guter Letzt war uns noch eine "richtige" und "unterirdische" Abkürzung wichtig. So etwas ist komplizierter als es scheint, denn eine "Bohr-Funktion" gibt es in Unity nicht. Unsere Lösung ist es, mittels *GameObjects* auf eine abgesenkte Stelle im *Terrain*, Wände und Decke des unterirdischen Ganges zu modellieren, und dann um das ganze nach außen hin zu verschleiern ein zweites *Terrain* Objekt über das Gebilde zu "spannen", und an den Enden dafür zu soren das es farblich, reibungslos in das erste *Terrain* Objekt über geht. Da man allerdings immernoch einen "Schnitt" sieht, wurde das ganze noch etwas mit Palmen retuschiert. Die Abkürzung

von Vorne, die Steine retuschieren den Übergang von *GameObject* zu Zweitem *Terrain*



Die Abkürzung von der zur Fahrbahn zeigenden Seite

4.5 Regenbogenstrecke

Autor: Dennis Altenhoff

Die Idee

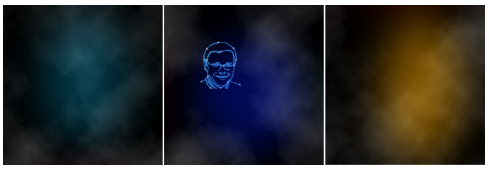
Die Idee der Regenbogenstrecke war es, entgegen der anderen Strecken, eine weniger realistische, sondern der Fantasie entspringende Strecke zu entwerfen. Es wurde sich an einer Strecke des Rennspiels *"Mario-Kart"* orientiert, die ebenso wie die unsere, den Namen *Rainbow-Road* trägt (im Deutschen *Regenbogen-Boulevard*). Ein Beispiel für das Original ist folgendes Bild:

Die Original *Rainbow-Road*. (Quelle[1])

1 http://www.diariodocentrodomundo.com.br/dois-amigos-na-rainbow-road/

Die Umsetzung

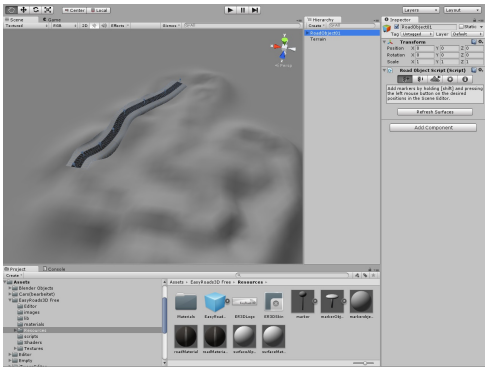
Auch in der Umsetzung unterschied sich die Regenbogenstrecke grundsätzlich von den anderen Strecken. Zum einen, weil keine Umgebung geschaffen werden musste, zum anderen, weil keine von Unity vorgegebene *Skybox* verwendet wurde. Hierfür wurde von einem der Gruppenmitglieder in mühevoller Arbeit eine neue *Skybox* erstellt, die an die Strecke angehangen wurde. Dafür mussten sechs Bilder erstellt werden, die als Seiten der *Skybox* dienten.



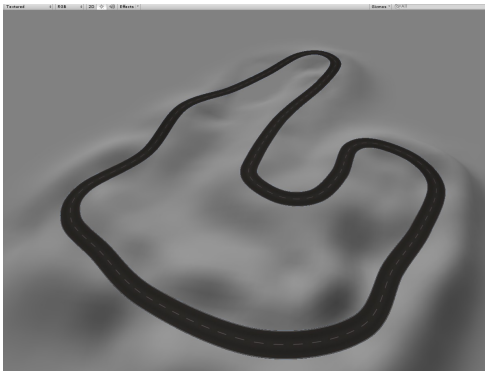
Einzelne Skybox-Bestandteile (Beispiele). (Hier zu sehen: Oben, Rechts, Unten)

Der erste Schritt zur Streckenerstellung war dennoch, wie bei den anderen Strecken, die Erstellung eines *Terrains*. Denn das heruntergeladene *Tool* zur Straßenerzeugung benötigte dieses als Unterlage und passte sich dessen Höhenverhältnissen an. Das bedeutete, dass durch *Terrain-Tools* Hügel erzeugt werden mussten, die den späteren Streckenverlauf darstellen sollten.

Dann wurde das darunterliegende *Terrain* wieder entfernt, wobei das fertige Straßen-Objekt bestehen blieb und auch seine Position und den Straßenverlauf beibehielt. Anschließend wurde die Straße mit einer selbsterstellten Regenbogen-Textur versehen. Des Weiteren wurde ein Skript angehangen, dass dafür sorgte, dass die Straße beidseitig sichtbar wurde, da dass beim heruntergeladenen Tool nicht vorgesehen war. (Quelle des Codes.[2])

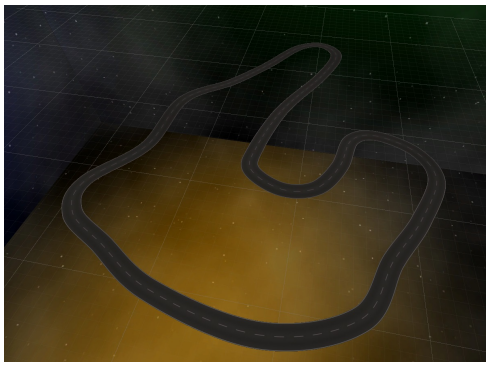


Erstellung der Strecke.

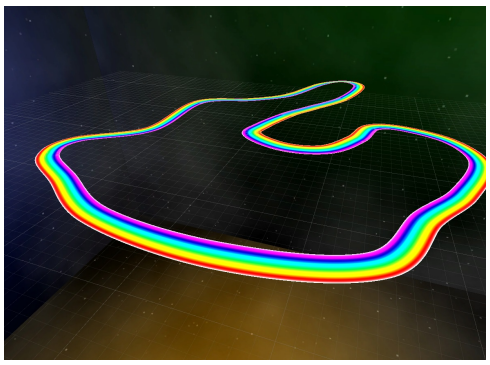


Die fertige Strecke mit der vorgegebenen Textur vor der Löschung des Terrains.

2 http://answers.unity3d.com/questions/280741/how-make-visible-the-back-face-of-a-mesh.html



Die fertige Strecke mit der vorgegebenen Textur ohne Terrain.



Die fertige Strecke mit der Regenbogen-Textur.

Nun wurde, um dem Vorbild näher zu kommen, die Strecke um Rampen ergänzt. Diese wurden durch die kurzfristig erhöhte Geschwindigkeit zu einem erschwerenden Faktor, der die ohnehin schwer befahrbare Strecke zusätzlich erschwerte. Man konnte aufgrund dieser nun noch leichter in darauffolgenden Kurven von der Strecke fallen.

Die Umsetzung der Rampen geschah durch eine Änderung am Fahrverhaltens der Autos.

Es wurde ein *Box-Collider* erstellt, der zu einem *Trigger* umgewandelt wurde, der die Rampen umgab. Also einem Bereich, in dem ein bestimmtes Ereignis ausgelöst werden kann, wenn ein Spieler selbigen betritt oder verlässt. Dies ist anders als beim *Collider*, der als eine unsichtbare Wand dient und bei dem nur bei Kollision mit einem anderen Objekt ein Ereignis ausgelöst werden kann. Der entsprechende *Code* dazu lautet:

```
/**
  *Geschrieben in C#
  */
                void OnTriggerEnter (Collider other){
                        if(other.gameObject.tag == "Rampe" && !rampenBoost){
                                vBackup=v;
                                maxSpeed+=55;
                        }
                }

                void OnTriggerStay (Collider other){
                        if (other.gameObject.tag == "Rampe" && !rampenBoost) {
                                bl.motorTorque = (float)(maxSpeed);
                                br.motorTorque = (float)(maxSpeed);
                                fl.motorTorque = (float)(maxSpeed);
                                fr.motorTorque = (float)(maxSpeed);
                                v=maxSpeed;
                                rampenBoost=true;
                        }
                }

                void OnCollisionEnter (Collision other){
                        if (other.gameObject.tag == "Straße" && rampenBoost) {
                                rampenBoost=false;
                                maxSpeed=this.speedbackup;
```

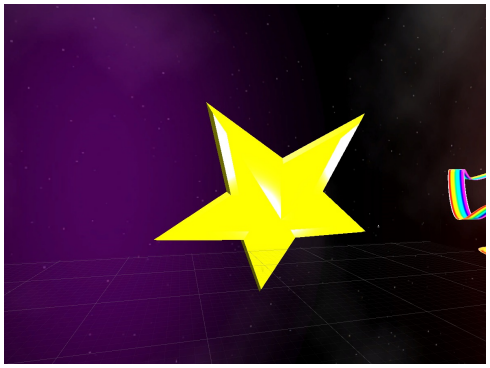
```
                    v=vBackup;
                }
            }
```

Diese Ergänzung führt dazu, dass der Spieler, wenn er die Rampe, die von dem *Trigger* umgeben wird, betritt, beschleunigt wird und mit der darauffolgenden Kollision mit der Straße, die das Ende des Sprungs bedeutet, seine Geschwindigkeit wieder angeglichen bekommt.

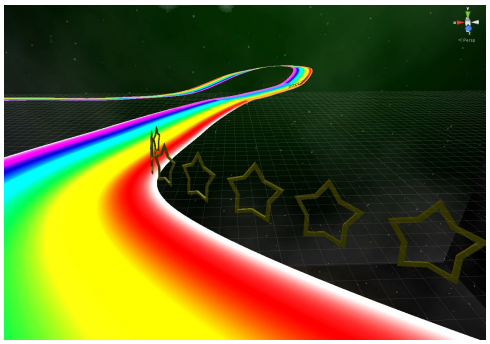
Eine Rampe in Aktion:

Nachdem die Strecke an sich fast fertiggestellt war, wurden, um die Umgebung der Strecke noch interessanter zu gestalten, in *Blender* erzeugte Sterne hinzugefügt und derart beleuchtet, dass sie den Eindruck erweckten sie würden selbst leuchten.

Im Folgenden wurden dann an Teilen der Strecke in Blender erzeugte Leitplanken hinzugefügt, die sich ebenfalls am Original orientierten. Zudem wurden zwei weitere, kleinere Strecken mit ähnlicher Textur eingefügt und rotiert, um als Umgebung zu dienen. Dies führte dazu, dass nun auch die Umgebung interessanter und ähnlicher zum Original wurde. (Ein entsprechendes Bild zur kompletten Umgebung findet sich am Ende des Kapitels zur Regenbogenstrecke)



Einer der Sterne.



Leitplanken der Regenbogenstrecke.

Zudem wurde in diesem Schritt eine äußerst schwierige, dafür umso lohnendere, Abkürzung hinzugefügt, die eine Sprungschanze darstellt, mit deren Hilfe man einen größeren Streckenabschnitt umgehen kann, dafür aber auch eine hohe Chance hat, von der Strecke hinunterzustürzen und so Zeit einzubüßen.

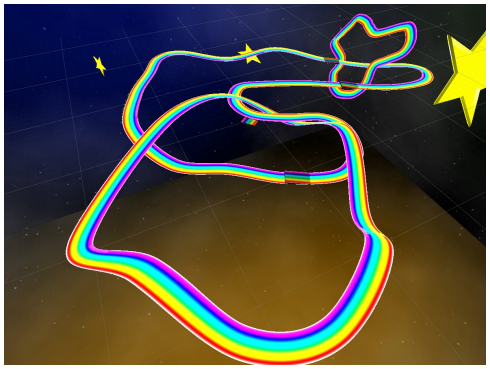
Die Abkürzung:

Da es, wie Tests zeigten, jedoch vorkommen konnte, das Unity bei einer Kollision zweier *Collider* nicht immer schnell genug für eine Ausbremsung sorgt, musste das Fahrverhalten der Autos angepasst werden. Diese Ergänzung führt zu einem Rückstoß-Effekt bei Kollision mit allen Objekten, die als Bande gekennzeichnet wurden:

```
/**
  *Geschrieben in C#
  */
                void OnCollisionEnter (Collision other){
                        if (other.gameObject.tag == "Bande") {
                                Vector3 dir = (transform.position -
other.gameObject.transform.position).normalized;
                                rigidbody.velocity = new
Vector3(dir.x*velBack,dir.y*velBack,dir.z*velBack);
                        }
                }
```

Wie genau ein *rigidbody* funktioniert, sollte im Kapitel Fahrphysik erklärt werden.

So ergab sich schließlich folgende Strecke:



Die fertige Regenbogenstrecke im Überblick.

4.6 Lightmapping

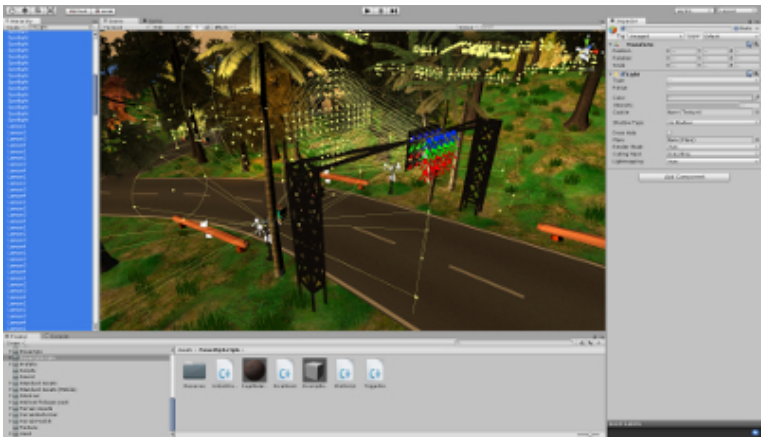
Autor: Alexander Altemöller

Es gab zwei grundlegende Probleme, welche wir durch das Thema *Lightmapping* lösen konnten.

Erstes Problem: Enormer Performanceverlust bedingt durch hohe Lichtquellenanzahl

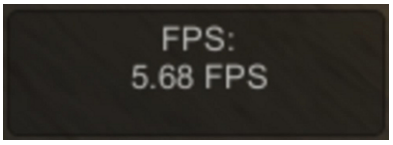
Nach der fertigen Modellierung einer kompletten Szene, kann es (je nach Komplexität der Szene) zu starken Performanceproblemen kommen. Ein Hauptgrund für diese Performanceprobleme sind unter anderem die Verwendung von mehreren hundert *GameObjects*, von denen ein Großteil Lichtquellen repräsentieren.

So kann es zum Beispiel vorkommen, dass in einer Szene am Ende des Modellierungsvorganges um die 200-300 Lichtquellen vorhanden sind.



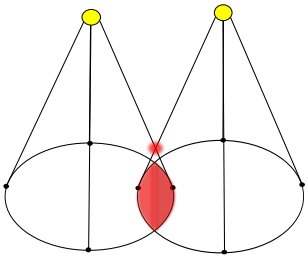
Licht in Unity

Jede dieser Lichtquellen wird in Unity3D zur Laufzeit des Programmes und in Echtzeit berechnet, und für jede Kameraposition aktualisiert. Hieraus folgt, dass die kleinste Bewegung der Kamera zu vielen neuen Berechnungen führt und somit das Programm nicht sehr performant und auch instabil läuft. Deutlich erkennbar ist dies an den *Frames per Second* (FPS). Während man für ein hinreichend gutes Spielerlebnis um die 25 bis 30 FPS benötigt, kann der Fall eintreten, dass die FPS unter 10 und im Schlimmsten Falle sogar unter 5 fallen, wodurch alle Bewegungen in der Szene vom menschlichen Auge als sehr kantig und abgehackt wahrgenommen werden.



Zweites Problem: Flackernde Lichter

Ein weiteres Problem mit Lichtquellen in Unity3D ist die Überschneidung von Lichtstrahlen, welche von mehreren Lichtquellen ausgehen.



Zwei sich überschneidende Spotlights. Der kritische Bereich ist rot markiert.
Falls ein solcher Bereich existiert, flackern alle betroffenen Lichter.

Zu der Laufzeit des Programmes entsteht in diesem Falle ein hässlicher „Flackereffekt“, welcher behoben werden muss.

Funktionsweise einer Lightmap

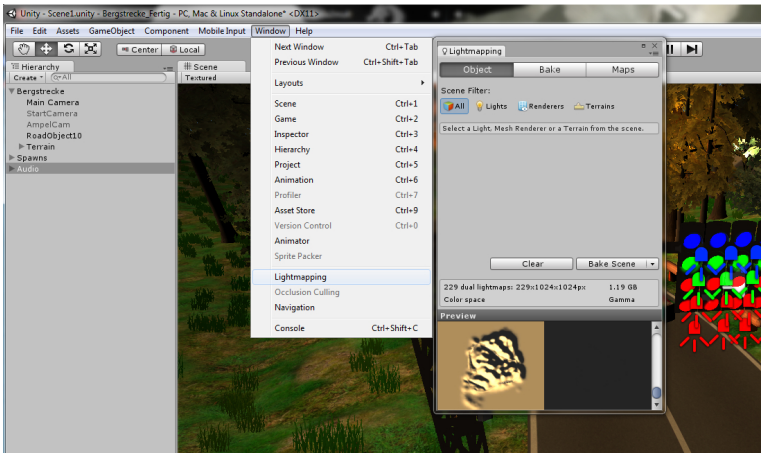
Die Lösung für diese beiden Probleme, ist das Erstellen einer sogenannten *Lightmap*.

Für eine Lightmap wird zu einem frei wählbaren Zeitpunkt die Beleuchtung der Szene statisch berechnet. Hierbei werden die Lichtinformationen der beleuchteten Szene direkt auf der Textur gespeichert, wodurch die alte, unbeleuchtete Textur "überschrieben" wird. Eine Lightmap kann man prinzipiell auf die Texturen von allen gewünschten Objekten anwenden, wobei natürlich die Anwendung auf die Bodentextur in den meisten Fällen am sinnvollsten ist.

Die angewandte Lightmap auf der Szene

Erstellen einer Lightmap in Unity3D

Lightmapping ist in Unity3D integriert und intuitiv verwendbar.



Die integrierte Lightmap Funktion von Unity3D

Das Lightmapping Menü lässt sich über den Reiter *Window -> Lightmapping* aufrufen. Hier stellt Unity3D dem Anwender gleich mehrere Einstellungsmöglichkeiten für die gewünschte Lightmap zur Verfügung.

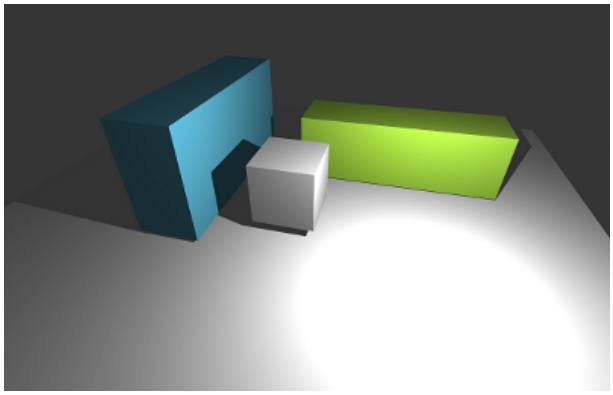
Als Erstes muss der User im Tab „*Object*“ die Lichtquellen auswählen, mittels der die Lightmap erzeugt werden soll. Standardmäßig sind hier alle Lichter angewählt, es gibt jedoch auch die Möglichkeit nur einzelne Lichter, einzelne *Renderer* oder einzelne Terrains auszuwählen, falls mehrere in einer Szene vorhanden sind. Von den letzten beiden Möglichkeiten haben wir keinen Gebrauch gemacht, weshalb in diesem Teil Dokumentation nicht näher auf die Renderer eingegangen wird. Desweiteren ist eine *Lightmap-Preview* aus der Vogelperspektive einsehbar, auf welcher der User noch vor dem Erstellen der Lightmap ungefähr einschätzen kann, ob er die richtigen Einstellungen getroffen hat, damit die Lightmap nach seinen Bedürfnissen erfolgreich berechnet wird.

Im Reiter „*Bake*“ sind die eigentlichen Einstellung für das Erstellen der Lightmap zu treffen.

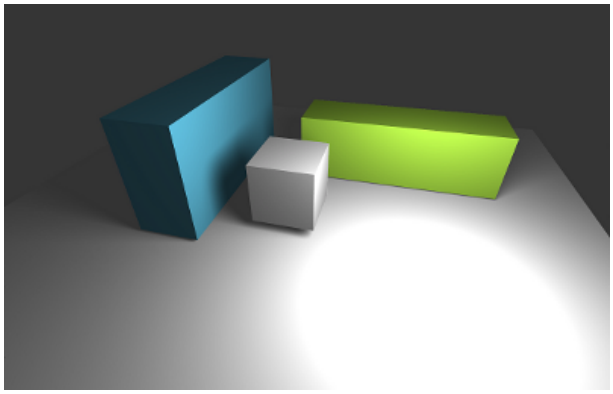
Hier lässt sich beispielsweise der Modus bestimmen, in welchem die Lightmap erstellt werden soll. Dazu hat der User die Wahl zwischen *Single Lightmaps*, welche harte Schatten haben, und *Dual Lightmaps*, welche eher realistischere, weichere Schatten besitzen. Falls man im Besitz der kostenpflichtigen Unity3D-Pro Version ist, lassen sich hier auch die nahezu perfekten *Directional Lightmaps* berechnen, bei welchen das Licht einer Lichtquelle (zum Beispiel das Licht der Sonne) sehr realistisch berechnet wird. Desweiteren lassen sich hier kleine Einstellungen wie beispielsweise der ambiente Lichtanteil und die Qualität der Lightmap einstellen. Eine höhere Qualität der Lightmap bedeutet natürlich gleichzeitig eine längere Berechnungszeit bei der Erstellung.

Im letzten Reiter „*Maps*“ lassen sich Einstellungen treffen, welche das Verhalten von mehreren Lightmaps untereinander bestimmen, falls in der Szene mehr als eine Lightmap verwendet wird. Hiervon haben wir in unserem Projekt keinen Gebrauch gemacht, weshalb hierauf in dieser Dokumentation nicht näher eingegangen wird.

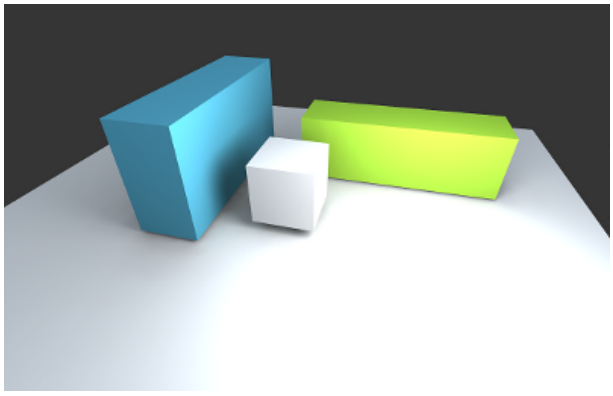
Mit einem Knopfdruck auf „*Bake Scene*“ wird die Lightmap nun statisch berechnet. Dieser Vorgang kann je nach Einstellungen einige Stunden Zeit beanspruchen.



Lightmapping mit harten Schatten



Lightmapping mit weichen Schatten

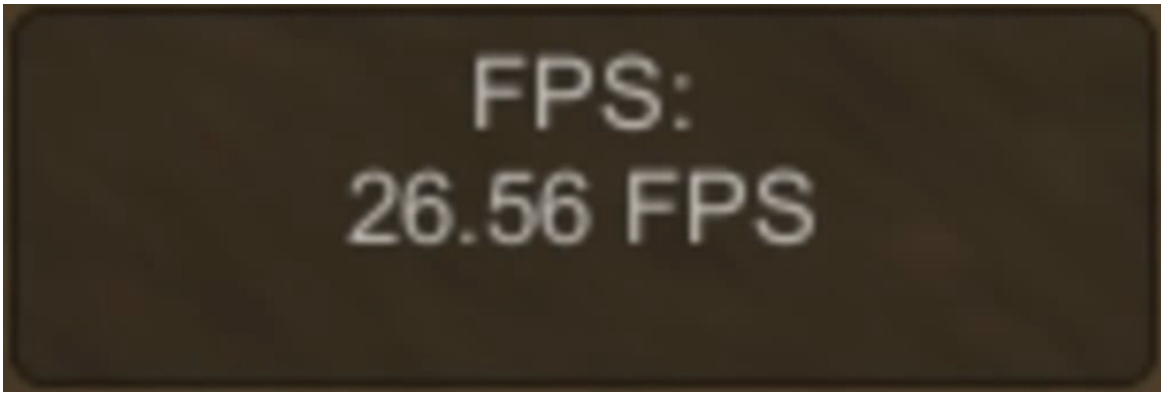


Lightmapping der Unity-Pro Version

Diese drei Bilder stammen aus der Unity3D-Dokumentation und sind zu finden unter http://docs.unity3d.com/Manual/Lightmapping.html .

Das Ergebnis

Nach dem erfolgreichen Erstellen einer Lightmap können nun in der Theorie alle Lichtquellen, welche die Lightmap mitberechnet hat, aus der Szene entfernt werden, wodurch eine drastische Performanceverbesserung erzielt wird.



Das Problem der flackernden Lichter ist ebenfalls gelöst, da sich nun keine Lichtstrahlen der Lichtquellen mehr schneiden, weil nun keine „echten“ Lichtquellen in der Szene existieren.

5. Grafische Benutzeroberfläche

Autor : Sebastian Dieckmann, Frederic Marchand

Eine grafische Benutzeroberfläche ist eine Benutzerschnittstelle des PC's, die Aktionen über graphische Symbole ermöglicht. Dabei gibt es zwei Möglichkeiten, wie man diese Symbole ansteuert, zum einen durch Klicken mit der Maus, zum anderen über Fingerbewegungen auf dem *Touchscreen*. Beide Mittel werden von uns in der Menüführung verwendet, wobei ersteres natürlich in der PC-Version und zweiteres in der Android-Version seine Verwendung findet.
Damit Knöpfe, Textfelder usw. aber auch korrekt angezeigt werden und benutzbar sind, müssen sie innerhalb der *OnGUI* Methode verwendet werden. Diese Funktion wird, ähnlich wie die *Update* Methode, jeden *Frame* ausgeführt und somit aktualisiert. Jeder *GUI* Komponente kann man eine Vielzahl von Argumenten übergeben, beim Knopf wäre das zum Beispiel der Name, der auf ihm steht, oder eine *Textur*, also sozusagen ein Bild und bei einem Textfeld kann man einen Text mitgeben, der am Anfang schon erscheinen soll, und/oder die maximale Länge der Zeichen, die man eintippen darf. Ein weiterer Bestandteil ist die *Scrollview*, also ein Scrollfenster, das beliebig gefüllt werden kann, wobei wir uns auf das Einsetzen von Knöpfen beschränkt haben. Dieses Element der Darstellung muss, genauso wie eine Gruppe (siehe unten), mit *Begin* angefangen, und mit *End* beendet werden. Desweiteren sind Größenangaben möglich, die, in Kombination mit der Anzahl der Komponenten in dieser *Box*, darüber entscheidet, ob der *Scrollbalken* überhaupt angezeigt wird.
Je nachdem, ob man GUILayout oder GUI verwendet, muss auch die Position der Komponenten durch ein Rechteck mitgegeben werden.
Ein erzeugter *Button* liefert einen *boolean*, also wahr oder falsch zurück, ob er gerade angeklickt wurde oder nicht. Diese Funktionalität ermöglicht ein einfaches Ausführen von *"ist der Knopf gedrückt, führe das folgende aus"* (siehe Beispiel unten). Eine wahrscheinlich bessere Methode mehrere Knöpfe auf einmal zu erzeugen, ist das sogenannte *selectionGrid*. Dazu erzeugt man zunächst ein *Stringarray* dessen *Strings* der Namen auf den Buttons und die Länge des *Arrays* natürlich der Anzahl entspricht. Dann erstellt man am besten einen *switch - case* über das *int* das dem *selectionGrid* zu gewiesen wurde, da dieses eine Zahl zwischen null und der Länge des *Stringarrays* zurückliefert.

Beispiel für einen Knopf

```
void OnGUI ()
{
        if (GUI.Button (new Rect (150, 240, 80, 32), "Exit")) {
                Application.Quit ();
        }
}
```

GUILayout

Mit *GUILayout* kann man verschiedene Komponente in einem automatisiertem Layout erzeugen, wobei man zunächst eine *Area* erstellen kann, die der Begrenzung der verschieden Bestandteile zusammengenommen entspricht. Es ist möglich so eine *Area* frei auf dem Bildschirm zu platzieren und den Bereich den diese einnimmt festzulegen. Daneben hat man die Möglichkeit eine Vertikale oder Horizontale Gruppenanordnung zu erzeugen, die die einzelnen Elemente nacheinander in der gegebenen Richtung anordnet. Wichtig hierbei ist, dass zum einen keine Verschachtelung von gleichen Gruppen möglich ist, und zum anderen die angefange Gruppe beendet werden muss.
Erzeugt man keine dieser Gruppen, wird in der oberen linken Ecke angefangen.

GUI

Eine manuelle Positionierung der Komponenten kann man durch die GUI's realisieren, wobei es auch hier wieder die Möglichkeit gibt mehrere Bestandteile zusammen zu fassen, indem eine Gruppe erstellt wird, die auch die gleichen Eigenschaften wie eine *GUILayout* - Gruppe besitzt außer der automatischen Anordnung innerhalb der Gruppe. Erzeugt man nun zum Beispiel einen *Button* mit einem Rechteck für die Position und ist dieser Knopf innerhalb von *BeginGroup* und *EndGroup*, werden die x und y Werte addiert. Hat man zum Beispiel

eine Gruppe angefangen mit den Koordinaten 200/200 und einen *Button* mit 50/150, ist die Position des Knopfes 250/350.

6. Menü

Hauptmenü

Autor : Sebastian Dieckmann

In dem Hauptnemü des Spiels, sieht man zunächst einmal vier Knöpfe, wobei *Einzelspieler* und *Multiplayer* zu der gleichen Kartenauswahl führen, da wir leider nicht mehr genügend Zeit hatten, den Einzelspieler in der PC-Version einzubauen. Möchte man trotzdem Einzelspieler spielen, kann man dies in der Web-Version und in der Android-Version machen.
Durch einen Klick auf *Exit* wird das Spiel beendet.

Das Hauptmenü

Optionsmenü

Autor : Frederic Marchand

Im Optionsmenü haben wir den Usern die Möglichkeit gegeben innerhalb der Anwendung die Auflösung und Qualität zu ändern. Um die Auflösung verändern zu können muss der User auf dem dazugehörigen Knopf drücken, wodurch eine Scrollview auftaucht in der einige Auflösungen aufgelistet sind und eine Checkbox mit deren Hilfe der Fenstermodus ein und aus gestellt werden kann. Jede dieser Auflösungen wird durch ein Knopf repräsentiert und kann durch drauf klicken und anschließendes bestätigen auf den Übernehmen-Knopf übernommen werden.
Bei den Qualitätsoptionen finden die User eine ähnliche Oberfläche, wobei durch die knöpfe innerhalb der Scrollview diesmal Qualitätsstufen dargestellt werden und die Checkbox fehlt.
Des Weiteren ist jede dieser Oberflächen mit einem „Zurück“ Knopf versehen, der den User eine Oberfläche zurück befördert.
Wir haben dieses Menü eingebaut, damit die User nicht jedes Mal die Anwendung neu starten müssen um diese Optionen zu ändern, sondern während der ausgeführten Anwendung die Möglichkeit haben darauf Einfluss zu nehmen.

Kartenauswahl

Autor : Sebastian Dieckmann

Nachdem man Multiplayer ausgewählt hat, gelangt man in das Menü zu Kartenauswahl, in welchem man vier Knöpfe mit jeweils einer eigenen *Textur* für jede einzelne Karte erkennen kann.
Hat man eine Karte ausgewählt und will sie sich anschauen, drückt man auf den *Anschauen* Knopf und es wird die Kamerafahrt dieser Karte aufgerufen. Durch Klicken auf den *Auswahl* Knopf gelangt man zur Autoauswahl und mit dem *Zurück Button* zum Hauptmenü.

Die Kartenauswahl

iTween

Autor : Sebastian Dieckmann

iTween Visual Editor ist ein kostenloses *Asset* aus dem *Asset Store*, das Pfad Animationen in Unity sehr vereinfacht und mit jeder Version von Unity funktioniert.
Obwohl es sehr viele Einsatzmöglichkeiten und Beispiele für diese Erweiterung gibt, wird hier nur auf das von uns Benutzte eingegangen, nämlich das Ansprechen der *iTween.MoveTo* Methode.
Damit diese Funktion korrekt ausgeführt wird, benötigt diese die richtigen Paramater, nämlich zum einen das zu bewegende *Game Object* und zum anderen *iTween.Hash*. In dieser Hashtabelle kann man dem Pfad verschiedene Optionen mitgeben, davon muss eines entweder eine Position oder ein Pfad sein, sonst würde das Objekt sich nicht bewegen.
"Orienttopath" ist eine weitere Auswahlmöglichkeit, die dafür sorgt, dass sich das Objekt am Pfad orientiert. In Kombination mit *"lookahead"* wird eine Zahl in Prozent festgelegt, wie stark diese Orientierung ist. Mit *"looktime"* wird angegeben wie lange das Objekt braucht, um sich den, mit *"looktarget"* übergebenen, Blickpunkt anzuschauen. Die Zeit, die die Animation brauchen soll, übergibt man einfach mit *"time"* als Zahl in Sekunden und möchte man eine automatische Schleife mit dem Pfad erzeugen, wird der *"looptype" "loop"* festgelegt.

Dazu kommt das Ausführen von Methoden bei verschiedenen Punkten des Pfades, nämlich beim Start, bei jedem Schritt, und am Ende.

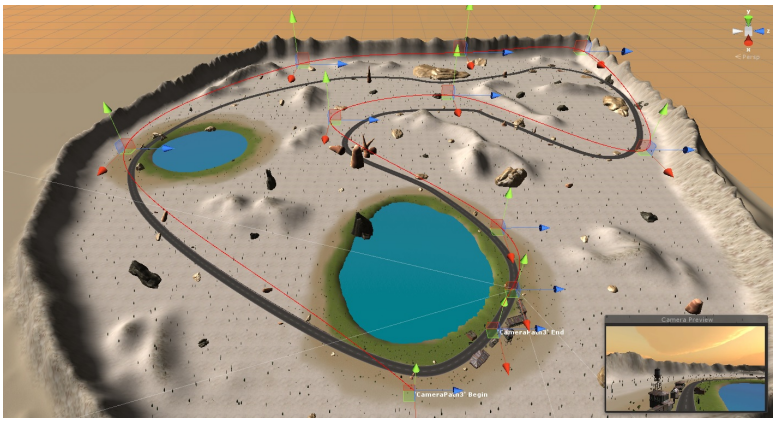
Beispiel für das Ausführen eines iTweens

```
void Start ()
{
       iTween.MoveTo (this.gameObject, iTween.Hash ("path", iTweenPath.GetPath ("CameraPath3"),
 "islocal",
        true, "orienttopath", true, "lookahead", 0.1, "time", 25, "easetype",
iTween.EaseType.easeInOutSine));
}
```

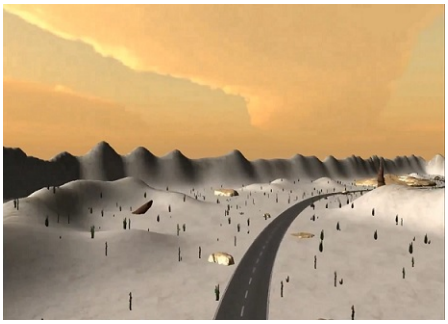
Kamerafahrt

Autor : Sebastian Dieckmann

Das erste Beispiel, in dem iTween benutzt wird, ist die Kamerafahrt.
Je nach Streckengröße müssen verschieden viele Punkte gesetzt werden, die nach belieben in x, y, und z Richtung verschoben werden können, bei der Wüstenstrecke sind es zum Beispiel zehn. Zwischen diesen Punkten wird eine Kurve angezeigt, höchstwahrscheinlich eine Bezier-Kurve. Damit die Kamera auch ungefähr in Richtung Straße schaut, geben wir der Methode, die für das *Fahren* zuständig ist, mit, dass sich die Kamera an dem Pfad orientieren soll.



Beispiel eines Pfades

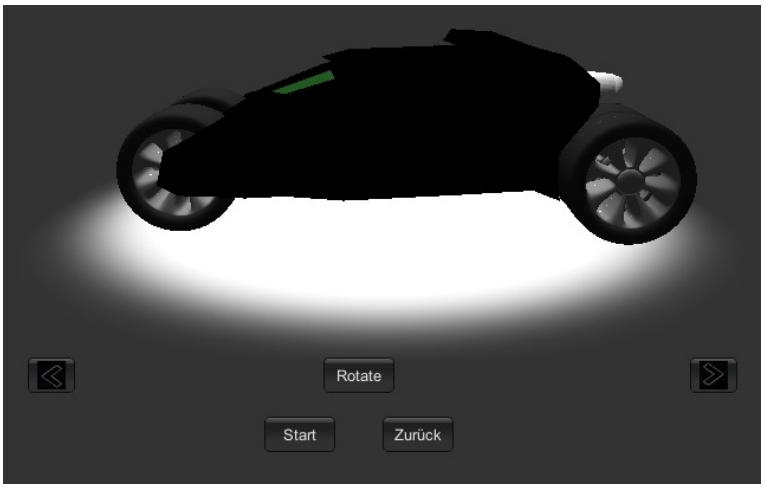


Beispiel einer Kamerafahrt

Autoauswahl

Autor : Sebastian Dieckmann

Grundlegender Gedanke hinter der Autoauswahl war, dass man schon etwas Anspruchsvolleres erstellt, als dass man einfach nur aus Bilder oder ähnlichem auswählen kann. Dazu werden erst einmal alle Autos in die Szene geladen und deaktiviert, damit man sie im Hintergrund nicht sieht.
Dann wird das erste Auto an den Anfang eines, für jedes Auto individuellen, iTween Pfades transformiert und aktiviert. Während das Auto nun entlang des Pfades *fährt*, wird durch die Update Methode eine Reifendrehung simuliert, die aufhört, sobald das Auto die Mitte erreicht hat. Hat das Auto die Mitte erreicht, kann man durch einen Klick auf den *Rotate* Knopf scheinbar das Auto zum drehen bringen, in Wirklichkeit aber bewegt sich die Kamera entlang eines Pfades mit Blick zum Auto.
Um das nächste Auto angezeigt zu bekommen, drückt man auf eine der beiden Pfeilknöpfe, dabei wird das vorher angezeigte Auto herausgefahren und das Neue kommt aus der anderen Richtung herangefahren.
Da dies in beide Richtungen möglich ist, arbeitet man am besten mit 2 Zählvariablen, in diesem Fall *preindex* und *index*, sonst kann es zu Unübersichtlichkeiten führen. *Index* ist die aktuelle Autonummer, die später als Argument übergeben wird.
Hat man nach ausreichender Überlegung das richtige Auto ausgewählt, wird mit einem Klick auf den *Start* Knopf die Anwendung der vorher ausgewählten Karte gestartet und die Hauptanwendung beendet.



Die Autoauswahl

7. Fahrphysik

In diesem Kapitel wird erklärt, was wir alles beachtet haben, um eine spielfreundliche und realistische Fahrphysik zu erschaffen.

7.1 Grundlegendes

Autoren: Felix Sandfort, Patrick Pfeiffer

Hier finden sich wichitge Utensilien von Unity erklärt, die von uns genutzt wurden, um die Fahrphysik zu erstellen.

7.1.1 Rigidbody

Autoren: Felix Sandfort, Patrick Pfeiffer

Der *Rigidbody* ist eine Komponente, die man einem *Gameobject* anhängen kann. Durch den *Rigidbody* erhält das entsprechende *Gameobject* physikalische Eigenschaften innerhalb des Spieles. So erhält es beispielweise eine Masse und einen Luftwiederstand, die beide variabel von außen einstellbar sind. Zudem ist einstellbar, ob eine Gravitation auf das *Gameobject* einwirkt oder nicht. *Gameobject*, die die Komponente *Rigidbody* haben, bewegen sich ihre *Child*-Objekte ebenfalls mit, wenn das erstgenannte *Gameobject* physikalisch beeinflusst wird.

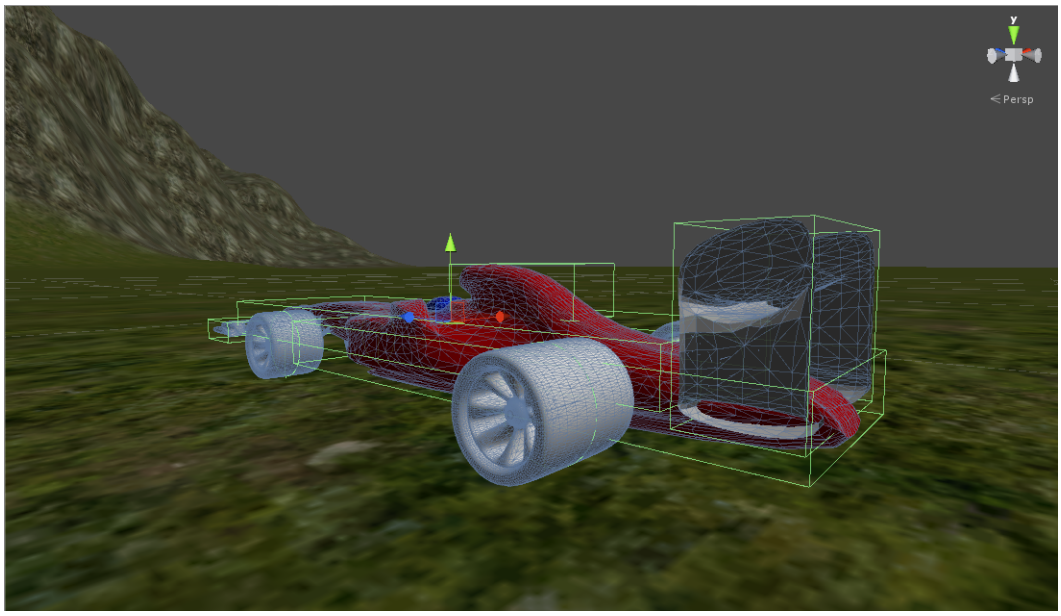
Rigidbodies können als *„Is Kinematic“* markiert werden. Wenn dies der Fall ist, werden sie nicht mehr durch physikalische Kräfte beeinflusst werden. Kollision existiert zwar noch, jedoch reagiert der *„Is Kinematic“-Rigidbody* etwa nicht physikalisch durch einen Rückstoß. *Rigibodies*, die als *„Is Kinematic“* markiert sind, können nur direkt, etwa durch ein Skript, verändert werden. Ob ein *Rigidbody* als *„Is Kinematic“* markiert ist, kann ebenfalls über ein Skript geändert werden.

7.1.2 Collider

Autoren: Felix Sandfort, Patrick Pfeiffer

Collider sind, wie der Name vermuten lässt, für die Kollisionsabfrage der Objekte innerhalb der Spielphysik zuständig und daher ein sehr wichtiger Bestandteil in dieser. Ohne *Collider* erfahren Objekte keine Kollision und ihr Körper geht durch andere Objekte hindurch.

Unity bietet verschiedene *Collider* zur Auswahl an. Zum einen gibt es die *Standardcollider*, diese sind die *Boxcollider* für quaderförmige *Collider*, *Spehrecollider* für kugelförmige *Collider* und *Capsulecollider* für kapselförmige *Collider*. Im Gegensatz dazu gibt es den *Meshcollider*, der sich der Form eines Objektes anpasst und so für eine genaue Kollisionsabfrage sorgt. Der Nachteil zu den *Standardcollidern* besteht darin, dass der *Meshcollider* rechenaufwändiger ist, weshalb dazu zu raten ist, ein Objekt mit möglichst passenden *Standardcollidern* auszustatten.



Neben den *Standardcollidern* und dem *Meshcollider* gibt es zudem noch den *Terraincollider*, der den *Collider* für das entsprechende Terrain darstellt.
Ein weiterer *Collider* ist der *Wheelcollider*. Dieser ist speziell für Reifen ausgelegt. Er achtet auf die Kollision mit dem Boden. *Wheelcollider* sind zudem sehr wichtig, damit ein Fahrzeug bewegt. Näheres dazu ist in dem entsprechenden Kapitel zu finden.

7.2 Die Fahrzeuge fahrbar machen

Autoren: Felix Sandfort, Patrick Pfeiffer

Hier sind die Punkte aufgelistet, die dafür sorgen, dass sich das Auto fortbewegen kann, womit diese Punkte innerhalb der Fahrphysik grundlegend sind.

7.2.1 Losfahren und Lenken

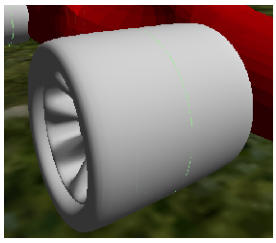
Autoren: Felix Sandfort, Patrick Pfeiffer

Da das grundlegendste Element der Fahrphysik darin besteht, dass sich das gewählte Fahrzeug durch einen Befehl entsprechend bewegt, bestand die erste Aufgebe darin, dass sich das Auto bewegen soll. Da die anderen Gruppen zu diesem Zeitpunkt ebenfalls gerade erst mit ihrer Arbeit begonnen haben, nutzten wir hierbei einen Wagen aus dem Asset Store von Unity und erstellen ein Terrain mit einer flachen Ebene und Bergen, um uns mit der Fahrphysik auseinandersetzen, die wir dann auf die später erhaltenden Wagen und Terrains anpassen können.

Weil wir noch keine Erfahrung mit Unity oder dem Erstellen einer Fahrphysik im Allgemeinen hatten, haben wir uns zuerst überlegt, wie man eine Bewegung herbeiführen könnte. Da die 4 Räder sowie der Chassis *Child*-Objekte des gesamten Autos, und somit Einzelteile, waren, bestand die erste Idee darin, direkt auf die Reifen zuzugreifen und diese so zu bewegen, dass sie sich wie bei einem echten Auto drehen. Da die Reifen daher nur begrenzte Bewegungsfreiheit haben sollten, haben wir *Joints* genutzt, die in Unity genutzt werden können, um zwei Objekte in bestimmter Weise aneinander zu befestigten. Jedoch bewegten sich die Reifen in jeder Jointbefestigung, die uns einfiel, entweder unrealistisch, so rotierten etwa die Reifen um den Mittelpunkt des Chassis, oder sie rührten sich gar nicht. Zudem hatten wir noch keine Möglichkeit die Bewegung der Reifen aktiv zu lenken, was in Anbetracht des gewünschten Ergebnisses nicht ausreichend war.

Aufgrund der Fehlversuche haben wir uns um mögliche Alternativideen bemüht, indem wir uns über die Möglichkeiten bei der Entwicklung mit Unity beschäftigt haben. Dabei stellte sich heraus, dass nicht die Reifen, sondern darum liegende *Wheelcollider* bewegt werden müssen. Diese *Wheelcollider* werden als von den Reifen

unabhängige Objekte erstellt, die aber auch *Child*-Objekte des Fahrzeuges sind. Die *Wheelcollider* werden hierbei an die Reifen des Fahrzeugs angepasst, sodass das Fahrverhalten dasselbe sein kann, wie wenn man mittels der Reifen fahren würde.



Mittels eines Skriptes werden dann die die *Wheelcollider* in die gewünschte Richtung bewegt. Hierfür wird dann der entsprechende Code in das Skript eingefügt.

```
bl.motorTorque = Input.GetAxis("Vertical")*v;
br.motorTorque = Input.GetAxis("Vertical")*v;
fl.motorTorque = Input.GetAxis("Vertical")*v;
fr.motorTorque = Input.GetAxis("Vertical")*v;
fl.steerAngle = Input.GetAxis("Horizontal")*dreh;
fr.steerAngle = Input.GetAxis("Horizontal")*dreh
```

Hierbei sind v und dreh frei einstellbare Variablen, damit man diese während des Testens des Fahrzeugs so verändern kann, dass das Fahrverhalten zufriedenstellend ist. V ist hierbei zunächst die Geschwindigkeit, wurde jedoch nach der Einführung der maximalen Geschwindigkeit zur Beschleunigung, während dreh für den Drehwinkel steht, den das Fahrzeug bei der Lenkung hat.

7.2.2 Schwerpunkt des Fahrzeuges

Autoren: Felix Sandfort, Patrick Pfeiffer

Nachdem es nun ein Skript gab, mit dem sich ein das Auto bewegen ließ, wurde dieses auf dem Testgelände ausprobiert. Dabei stellte sich schnell heraus, dass das Fahrzeug umkippt, wenn man zu stark lenkt oder im falschen Winkel über eine Steigung fährt.
Die erste Idee, wie man dieses Problem lösen könnte, bestand darin, die allgemeine Gravitation des Gebietes zu erhöhen, da die Fahrzeuge dadurch nicht mehr so schnell vom Boden abheben können und sich somit nicht mehr in der Luft ungünstig neigen können. Außerdem sollte so ein weiteres Problem behoben werden, dass darin bestand, dass das Fahrzeug bei einem Sprung von einer Rampe oder einer ähnlichen Anhöhe unrealistisch weit flog. Jedoch wurden beide Probleme durch diesen Lösungsansatz nicht behoben. Die Lösung zu letzteren Problem findet sich hier.

Nach weiterer Überlegungen konnten wir feststellen, dass die Ursache des Problems darin lag, dass das Fahrzeug seinen Schwerpunkt in seiner Mitte hatte, was für die meisten Fahrzeuge nicht der Fall ist, die diesen weiter unten haben. Daher bestand der zweite Lösungsansatz darin, den Schwerpunkt des Fahrzeuges weiter nach unten zu platzieren. Hierbei mussten wir den *Rigidbody* des Fahrzeuges ansprechen, da dieser dafür zuständig war, dass die Schwerkraft Einfluss auf das Fahrzeug hat. Wir ergänzten das Skript so, dass zu Beginn der Schwerpunkt des Spieles mithilfe von Variablen neu definiert wird. Der entsprechende Code sieht wie folgt aus:

```
rigidbody.centerOfMass = new Vector3
(rigidbody.centerOfMass.x+GewichtspunktX,rigidbody.centerOfMass.x+GewichtspunktY,GewichtspunktZ);
```

Hier wird jeweils eine Variable für den Schwerpunkt in x-Richtung, y-Richtung und z-Richtung bestimmt, der den Schwerpunkt jeweils in der jeweiligen Richtung ausgehend vom Mittelpunkt verschiebt. Hierbei wurden wieder Variablen genommen, um den passenden Schwerpunkt während der Fahrt zu bestimmen und korrigieren zu können. Zudem stellten sich die Variablen zu einem späteren Zeitpunkt als nützlich heraus, da jedes Auto einen anderen sinnvollen Schwerpunkt hat und dieser dadurch für jedes Fahrzeug angepasst werden kann.

7.3 Limitierungen

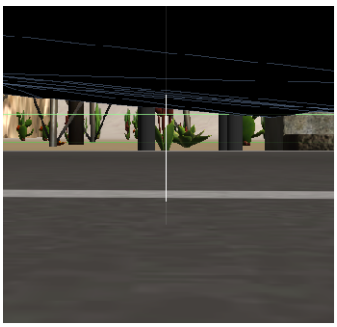
Autoren: Felix Sandfort, Patrick Pfeiffer

Da Variablen, die eine nicht einschätzbare Größe annehmen können, zu unerwünschten Nebenwirkungen führen können, sind Limitierungen erforderlich. Diese sind hier angeführt.

7.3.1 Raycasthit

Autoren: Felix Sandfort, Patrick Pfeiffer

Nachdem die Autos fahren konnten und eine vernünftige Lenkung hatten, bemerkten wir schnell, dass eine weitere Sache nicht ganz unseren Vorstellungen entsprach. Die Autos flogen, nachdem sie einen Berg hochfuhren, unrealistisch weit durch die Luft, bis sie wieder landeten. Anfangs wollten wir dies, genau wie das schnelle Umkippen der Fahrzeuge, durch die Gravitation regeln, welche wir dazu erhöht haben. Jedoch bemerkten wir, dass dies nicht die beste Lösung war, da man die allgemeine Gravitation nicht in den Fahrzeugskripts einstellen konnte, sondern hätte in den Maps eingestellt werden müssen. Also suchten wir nach einer anderen Lösung, welche sich in den Fahrzeugen einstellen lässt. Nach einiger Überlegnung hatten wir die Idee, einen *Raycast* zu nutzen, welcher die Distanz von dem Fahrzeug zu dem Boden abmisst und so prüft, ob der Boden nahe am Fahrzeug ist.



Für den Fall, dass der Boden nicht nahe am Fahrzeug ist, haben wir in jedem Frame eine nach unten wirkende Kraft auf das Fahrzeug hinzu addiert, welche dafür sorgte, dass das Fahrzeug sich dem Boden schneller nähert, wodurch die Sprünge nicht mehr so weit sind und dadurch realistischer ausfallen. Die maximale Distanz, die zwischen einem Fahrzeug und dem Boden liegen kann, damit diese noch als nahe interpretiert werden kann, ist als eine Variable im Skript aufzufinden, damit man den entsprechenden Wert während der Testfahrten anpassen kann und bei den verschiedenen Fahrzeugen unterschiedlich einstellen kann.

7.3.2 Maximale Geschwindigkeit

Autoren: Felix Sandfort, Patrick Pfeiffer

Ein weiteres Problem, dass während der Testfahrten aufgetreten ist, war die unbegrenzte Geschwindigkeit. Da dies für die meisten Fahrzeuge unrealistisch ist und ungewünschte Nebeneffekte hervorrufen kann, ist hier eine maximale Geschwindigkeit als Limitierung wünschenswert. Anfangs hatte wir uns überlegt, die Höchstgeschwindigkeit wie bei einem richtigen Wagen über die Kraft, welche die Räder ausüben und den Luftwiderstand des Autos, welcher bei Unity über den *Rigidbody* einstellbar ist, zu regeln. Jedoch wies diese Art einige Mängel auf, da dies schon bei kleinen Sprüngen zu starken Geschwindigkeitsverlusten führte, was nicht ganz im Sinne einer realistischen Fahrphysik liegt.

Nach weiteren Überlegungen haben wir uns dann dafür entschieden, in den Skripten der Fahrzeuge ein Attribut einzuführen, welches für die Limitierung der Höchstgeschwindigkeit zuständig ist. Dieses Attribut ist eine veränderbare Variable, damit sie von außen während der Testphase verändert werden kann, damit so die Maximalgeschwindigkeit besser angepasst werden kann und später die verschiedenen Fahrzeuge unterschiedliche Werte erhalten können. Wenn diese nicht überschritten wird, ist alles in Ordnung und das

Auto kann wie zuvor weiter beschleunigen. Wenn sie allerdings überschritten wird, wird von dem aktuellen Geschwindigkeitvektor des Fahrzeuges der normierte Geschwindigkeitsvektor in jedem Frame subtrahiert, bis die Geschwindigkeit wieder auf dem Wert ist, den wir als maximale Geschwindigkeit vorgesehen haben.

7.3.3 Maximale Steigung

Autoren: Felix Sandfort, Patrick Pfeiffer

Nachdem nun die unrealistischsten Probleme behoben waren, widmeten wir uns einem weiteren Problem, dass es zu lösen galt. Die Autos hatten mitlerweile zwar eine Höchstgeschwindigkeit, jedoch bestand weiterhin die Möglichkeit, extrem steile Abhänge hoch zu fahren, was eine realistische Physik nicht zulasseen würde. Somit überlegten wir uns auch hier eine Lösung. Anfangs war die Idee, dass dieses Problem sich ebenfalls mit dem Raycast lösen lassen könnte, jedoch ging der Raycast vom Ursprung des Fahrzeuges aus und war somit nicht geeignet, um damit auch die Steigung zu begrenzen. Somit haben wir uns überlegt, die Rotation des Fahrzeuges auszuwerten und somit zu verhindern, dass man unendliche Steigungen hochfahren kann, indem wir ab einem bestimmten Winkel, bei uns 30 und 60 Grad, zusätzliche Kraft auf das Auto wirken lassen, so wie wir es auch bei dem Problem mit den weitfliegenden Fahrzeugen getan haben. Jedoch ließ sich das Problem so nur teilweise lösen, da man immer noch seitlich an Hängen fahren konnte, was auch nicht so ganz unsere Absicht war. Somit überlegten wir, wie wir auch dieses Problem am besten lösen können. Die letztendliche Lösung war dann die Traktion der Reifen zu verringern, wenn man am Hang ist und somit für ein leichtes Runterrutschen zu sorgen.

7.4 Reifen an Wheelcollider anpassen

Autoren: Felix Sandfort, Patrick Pfeiffer

Da das Fahrzeug über die *Wheelcollider* und nicht über die Reifen bewegt wird, kann das Fahrzeug fahren, ohne das sich die Reifen bewegen. Dies ist allerdings nicht wünschenswert, da dies einen sehr unrealistischen Eindruck hinterlässt. Daher wollen wir die Reifen während der Fahrt drehen lassen. Hierbei ist es ein Ziel, dass sich die Reifen entsprechend der Geschwindigkeit des Fahrzeuges bewegen, so dass der Eindruck entsteht, dass die Reifen für die Fortbewegung verantwortlich sind. Hierfür erstellen wir eine Methode drehen(), die wir in die Update-Methode des Fahrzeugs-Skripts einfügen, damit sie bei jedem Update aufgerufen wird. Die Methode sieht wie folgt aus:

```
void drehen(){
        flwheel.Rotate (fl.rpm / 60 * 360 * Time.deltaTime,0,0);
        frwheel.Rotate (fr.rpm / 60 * 360 * Time.deltaTime,0,0);
        blwheel.Rotate (bl.rpm / 60 * 360 * Time.deltaTime,0,0);
        brwheel.Rotate (br.rpm / 60 * 360 * Time.deltaTime,0,0);
        }
```

Flwheel, blwheel, frwheel und brwheel sind hierbei die vier Reifen des Fahrzeuges. durch die Methode Rotate() dreht sich der jeweilige Reifen entsprechend der Werte auf den jeweiligen Achsen. Da sich die Reifen nur um die x-Achse drehen sollen, erhalten die Drehungen in y- und z-Achse den Wert 0. Die Drehung um die x-Achse wird bestimmt, indem wir zunächst die Drehung pro Minute des *Wheelcolliders* des dazugehörigen Reifens bestimmen. Hierfür nutzen wir das Attribut rpm des *Wheelcolliders*. Da wir jedoch die Drehung pro Sekunde benötigen, teilen wir den Wert durch 60. Außerdem multiplizieren wir den Wert mit 360, da wir den Winkel bestimmen wollen. Zum Schluss multiplizieren wir noch Time.deltatime zu dem Wert, damit dieser nicht pro Frame, sondern pro Sekunde berechnet wird.

Mittels dieser Methode drehen sich nun die Reifen passend zu der Geschwindigkeit des Fahrzeuges. Allerdings drehen sich die Reifen noch nicht nach links oder rechts, wenn man in die entsprechende Richtung lenkt. Dafür erstellen wir nun zusätzlich eine Methode lenken(), die wir ebenfalls in der Update-Methode aufrufen. Die Methode sieht wie folgt aus:

```
void lenken(){
```

```
                        float fly=fl.steerAngle*2-flwheel.localEulerAngles.z;
                        float fry=fr.steerAngle*2-frwheel.localEulerAngles.z;
                        flwheel.localEulerAngles=new
Vector3(flwheel.localEulerAngles.x,fly,flwheel.localEulerAngles.z);
                        frwheel.localEulerAngles=new
Vector3(frwheel.localEulerAngles.x,fry,frwheel.localEulerAngles.z);
                }
```

Durch diese Methode drehen sich die Vorderreifen um die Y-Achse entsprechend der dazugehörigen *Wheelcollider*. Hierfür setzen wir zuerst die Variablen fly und fry für die Rotation der Vorderreifen um die y-Achsen. Diese erhalten als Wert die Drehung der entsprechenden *Wheelcollider* mit 2 multipliziert. Die 2 wurde hinzu multipliziert, da Testfahrten gezeigt haben, dass die Reifendrehung ansonsten nicht sehbar ist. Anschließend muss noch der momentane Winkel in z-Richtung subtrahiert werden, damit unschöne Nebeneffekt vermieden werden, die aufgrund Drehung der Reifen, und somit des Koordinatensystems, auftreten. Hierzu wird localEulersAngles genutzt, da wir auf das lokale Koordinatensystem des Reifens zugreifen wollen. Zuletzt werden die neuen Werte für die y-Achse in die jeweiligen Reifen eingefügt.
Somit drehen sich auch die Reifen passend zur Fahrt.

7.4.1 Kippen von Zweirädern

Autoren Felix Sandfort, Alexander Tessmer

Da in Unity die Gravitationsphysik für Rigidbodys schon vorhanden ist, lag der Schluss nahe, diesen zu verstärken und so einen Kippeffekt für die Zweiräder zu erzeugen. Die Umsetzung erfolgte durch das Verändern der Höhe des Schwerpunktes vom Fahrzeug nach oben bzw. unten. Dies führt allerdings zu einem extrem wackeligem und unrealistischem Pendeln.

Die nächste Idee bestand aus dem direkten Setzen der Rotation vom Fahrzeug in Abhängigkeit zum Drehwinkel des Vorderrades. Hier muss man anmerken, dass jedes Fahrzeug vier Wheelcollider benötigt, um mit diesen zu fahren. Dadurch ist der Radboden planar und beim Drehen wird eine Kante in den Boden hinein gedreht. Dies sorgt dafür, dass man entweder auf dem Terrain durch den Boden fällt oder von der Straße wieder nach oben auf die Straße gesetzt wird, sodass man sich in die entgegengesetzte Richtung bewegt, in die man lenkt.

Im dritten Versuch sollte eine Drehkraft auf den Rigidbody angewendet werden. Allerdings überschlug sich dadurch das Fahrzeug deutlich drastischer als es realistisch wäre, da die Drehkraft konstant angewendet wird, während man die Kurve fährt.

Der letzte Ansatz bezog sich darauf, die Wheelcollider nicht zu rotieren, da diese aber nicht sichtbar sind, stört dies nicht, allerdings mussten dazu alle Komponenten des Fahrzeugs außer die Wheelcollider und die Kamera des Fahrzeugs in einem eigenen, leeren Gameobject zusammengefasst werden. Nun tat sich noch ein Problem auf, denn das Kippen der Elemente war sichtbar, allerdings waren diese plötzlich vollkommen falsch rotiert. Dies konnte jedoch verhindert werden, indem man die Rotation nicht direkt anwendet, sondern zur vorherigen Rotation hinzufügt.

7.5 zusätzliche Funktionen

Autoren: Felix Sandfort, Patrick Pfeiffer

Hier werden die Unterpunkte vorgestellt, die nicht notwendigerweise in einem Rennspiel vertreten sein müssen, sich jedoch während der Testphasen als praktische und sinnvolle Funktionen für die Fahrt herausgestellt haben.

7.5.1 Vollbremsung

Autoren: Felix Sandfort, Patrick Pfeiffer

Nach unseren ersten Testfahrten, mussten wir schnell feststellen, dass die Autos durch eine entgegengesetzte Beschleunigung mittels der *Wheelcollider* nur sehr langsam zum stehen kommen. Deswegen haben wir uns überlegt eine starke Bremse einzubauen, die dazu dienen soll das Fahrzeug schneller anzuhalten. Hierfür bauten wir eine Abfrage ein, die prüft, ob der Spieler die Leertaste drückt. Wenn er dies tut sollte sich die starke Bremse aktivieren. Diese bestand anfangs einfach daraus eine Bremskraft an die Reifen zu legen, welche das Auto bremst. Nach unserem ersten Versuchen stellten wir fest, dass das Auto äußerst abrupt abbremst und entschlossen uns die Bremskraft sehr stark zu verringern.

Diese Änderung wirkte sich zunächst positiv aus, doch dann fiel uns auf, dass man sich trotz unserer Überlegungen bezüglich der maximalen Steigung mit der Handbremse am Hang festkrallen kann, was aufgrund der Unrealistik nicht der Fall sein sollte. Nach kurzer Überlegung kamen wir somit zu dem Schluss, die Bremskraft der Handbremse wieder zu erhöhen und dafür die Bodenhaftung der Reifen stark zu reduzieren, was dazu führte, dass man auch mit Handbremse steilere Hänge herunterrutscht und man sich nicht mehr festkrallen kann, was unser gewünschtes Ergebnis war.

7.5.2 Turbo

Autoren Patrick Pfeiffer, Alexander Tessmer

Da wir an einem Rennspiel arbeiteten, musste selbstverständlich eine Turbofunktion her. Die erste Idee bestand aus einem einfachen Boost, der sich auf Knopfdruck aktiviert. Dann vollständig ausbrennt und sich wieder auflädt, um dann erst wieder erneut benutzbar zu sein. Dies wird mit einem einfachen Zeitzähler realisiert, der sich hoch zählt und an bestimmten Zeitpunkten ist die Turbofunktion dann verbraucht bzw. wieder aufgeladen.

```
If(!verbraucht && Input.getKeyDown(KeyCode.Shift)){
        turbo=true;
}
if(turbo){
        counter+= Time.fixedDeltaTime;
        if(counter>turboDauer){
                verbraucht=true;
                turbo=false;
                counter=0f;
        }
}
if(verbraucht)
        counter+= Time.fixedDeltaTime;
        if(counter>refreshDauer){
                verbraucht=false;
                counter=0f;
        }
}
```

Die Umsetzung der Turbofunktion geschieht durch einen konstanten Faktor, der auf die Beschleunigungskraft multipliziert wird und eine Erhöhung der maximal erreichbaren Geschwindigkeit. Hier fiel uns auf, dass die Turbofunktion leider auch nach hinten beschleunigt. Dies konnten wir verhindern, indem wir nicht die Beschleunigungskraft verändern, sondern eine positive Beschleunigung hinzufügen.
Außerdem haben wir ein Partikelsystem angehängt, welches einen Impulsantrieb darstellt, und dieses wird, während die Turbofunktion aktiv ist, abgespielt.

Diese Implementation war uns allerdings nicht authentisch genug. Die Turbofunktion sollte nicht anhand der Zeit, sondern in Form eines Volumens zum Beispiel als Tank dargestellt sein. Dementsprechend zählt der Zähler nun von der maximalen Kapazität herunter und ist dann irgendwann auf null und damit leer.

Eine weitere Veränderung sollte in der Bedienung implementiert werden. Man sollte die Turbofunktion auch abbrechen und wieder starten können, sofern die Turbofunktion nicht verbraucht ist, denn dann sollte sie sich erst wieder ganz aufladen, bevor man sie wieder starten kann. In dieser Implementation ist die Turbofunktion nicht mehr anhand eines Boolean aktiv, sondern während Shift gedrückt wird.

7.6 Effekte

Autoren: Felix Sandfort, Patrick Pfeiffer

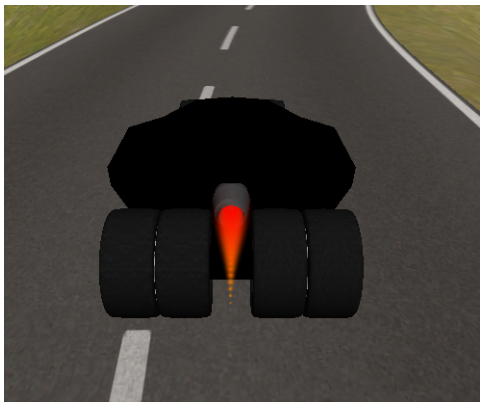
Die hier gezeigten Unterpunkte dienen dem besseren und glaubwürdigeren Aussehens des Rennspiels.

7.6.1 Partikelsysteme

Autoren: Felix Sandfort, Patrick Pfeiffer

Als unsere Physik soweit fertig war, überlegten wir, wie wir diese wenigstenes ein wenig sichtbar machen können. Eine gute Lösung hierfür schienen uns die bei Unity schon vorhandenen Partikelsysteme zu sein. Wir kamen zudem zu dem Schluss, dass es am sinnvollsten wäre, diese für den Turbo und die Handbremse zu benutzen. Um die Partikelsysteme an die jeweiligen Fahrzeuge anzupassen, nutzten wir die bei Unity schon vorhandenen Parameter. Für den Turbo nutzten wir, da wir eine Flamme darstellen wollten, die Änderung der Größe über die Lebenszeit und ließen die Größe konstant bis zum Ende eines einzelnen Partikels dadurch verringern. Wir haben uns ebenso entschieden, dass die Partikel nicht von der Schwerkraft beeinflusst werden, da wir dies für logisch nachvollziehbar hielten, denn eine Flamme wird nicht von der Schwerkraft beeinflusst und der Rauch, der bei einer Vollbremsung enstehen würde, sinkt wenn überhaupt auch nur minimal zu Boden. Außerdem entschieden wir uns dagegen, die Partikel mit einer Kollisionsabfrage zu versehen, da dies nur rechenintensiver wäre, die Partikel nur mit Fahrzeugen in Berührung kommen würden und es zwar realistischer aussehen würde, wenn der Rauch von einem Fahrzeug zur Seite geschoben wird. Allerdings bestand der Rauch nur aus wenigen Partikeln, somit wäre dieser nur in eine Richtung geschoben worden, was noch unrealistischer ausgesehen hätte als ihn einfach durch das Auto gleiten zu lassen, sodass wir uns dagegen entschieden haben, das dies passiert. Die Dauer der Partikelsysteme stellten wir niedrig ein, da nicht noch 2 Sekunden oder später nach dem Turbo Partikel entstehen sollten, da dies nicht passend gewesen wäre, wenn es ein Indikator für den Turbo sein soll. Als Textur, die wir über die einzelnen Partikel setzen wollten, nahmen wir bei den Partikelsystemen für die Handbremse schwarzen Rauch, da wir der Meinung waren, dass dieser am besten zu abnutzenden Reifen passen würde.

Hier war allerdings auch das Problem, dass die Partikel nicht einfach so erscheinen sollten, wenn man die Handbremse drückt, sondern erst, wenn das Fahrzeug an Geschwindigkeit verloren hat. Damit dies auch auf alle Fahrzeuge übertragbar ist, haben wir eine Variable eingefügt, welche es uns ermöglichen sollte eine Differenz für jedes Fahrzeug festzulegen. Dieses Partikelsystem haben wir logischerweise bei Fahrzeugen wie dem *Hovercar* nicht eingefügt, da diese Fahrzeuge schweben und keine Reifen besitzen, wodurch es keine Reifenabnutzung geben kann. Da der Rauch, der von den Reifen abgesondert wird, an der Stelle bleiben sollte an der dies geschah, haben wir bei den Partikelsystemen für den Staub als *Simulation Space*, also in welchem Koordinaten sie simuliert werden sollen, Weltkoordinaten angegeben, während wir bei einigen Partikelsystemen für den Turbo *local* anwählten, damit die Flammenpartikel immer in gleicher Relation zum Fahrzeug liegen. Damit sie dann trotzdem wie eine Flamme aussehen, mussten wir noch eine Geschwindigkeit einbinden, die die Partikel vom Fahrzeug fortbewegt. Man hätte dies über eine Kraft tun können, diese wäre jedoch immer schneller geworden, was nicht wirklich Sinn ergeben würde. Sinnvoller wäre es gewesen, die Geschwindigkeit der Partikel zu dämpfen, was wir allerdings nicht gemacht haben, da dies für uns unwichtig erschien, weil die Partikelsysteme so schon recht gut aussahen. Des Weiteren haben wir bei dem Partikelsystem für den Turbo die Farbe verändert.

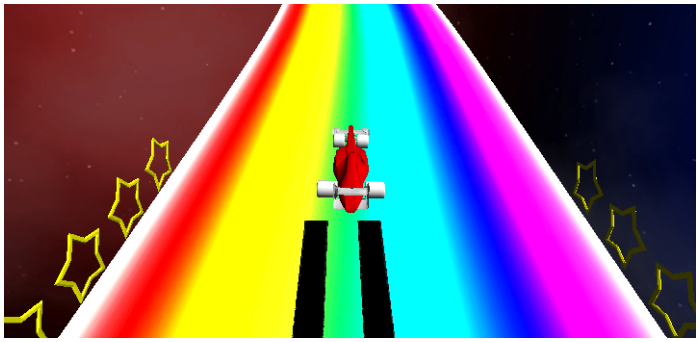


Man kann bei Unity die Startfarbe einstellen, mit der ein Partikel versehen sein soll. Dies haben wir bei den meisten allerdings nicht gemacht, sondern die Farbe der Partikel über *Color over Lifetime* geregelt, was bei einer Flamme mehr Sinn macht, da diese anfangs eine andere Farbe hat als gegen Ende. Bei Unity lassen sich 8 verschiedene Farbwerte auswählen und der Zeitpunkt zu dem diese jeweils auftreten sollen. Hinzuzufügen ist, dass man bei Unity auch noch die Form des Partikelemitters auswählen kann. Dieser kann entweder eine Kugel, Halbkugel, Kegel, Rechteck oder ein *Mesh*(passt sich den *Faces* eines Objektes an) sein und es ist einstellbar, von wo aus die Partikel emittiert werden sollen.

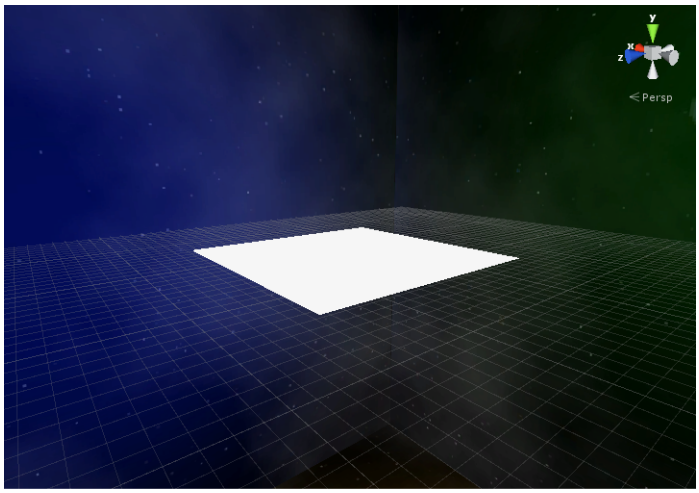
Nachdem wir alle Parameter an die Autos angepasst hatten, stellte sich uns nur noch die Frage, wie wir auf diese Objekte in unserem Skript zugreifen, sodass sie zur richtigen Zeit gezündet werden und nicht dauerhaft aktiv sind. Die Lösung war eine öffentliche Variable, auf die wir die einzelnen Partikelsystem per *Drag&Drop* ziehen, damit das Skript diese kennt. Dasselbe haben wir zuvor schon mit den *WheelCollidern* gemacht. Das einzige, was wir im Skript jetzt noch tun mussten, war die Emissionsrate an den richtigen Stellen an die jeweilige Situation anzupassen.

7.6.2 Bremsspuren

Autoren Felix Sandfort, Patrick Pfeiffer



Um das Spiel besser aussehen zu lassen, wollten wir neben dem Rauch noch Bremsspuren einbauen, die etwa im Fall einer Vollbremsung erscheinen. Hierbei stellte sich zuerst die Frage, wie man Bremsspuren auftreten lässt. Nachdem wir uns hierzu hinreichend informiert haben, haben wir Bremsspuren entstehen lassen, indem wir ein Gameobject der Klasse Plane erstellen.



Das besagte Gameobject texturieren wir mit einer Bremspuren-Textur. Danach wird das Skript des Fahrzeuges dahingehend verändert, dass diese Planes an den Reifen erzeugt werden, wenn dieselben Bedingungen erfüllt sind, wie für den Bremsrauch. Hierfür nutzen wir folgenden Code, der für jeden Wheelcollider genutzt wird, sofern dieselben Bedingungen wie beim Bremsrauch erfüllt sind:

```
if (velo - rigidbody.velocity.sqrMagnitude > faktor)  {

        if(bl.GetGroundHit(out hitt)){
                skidmarkPos = hitt.point;
                skidmarkPos.y+=0.2f;
                bl.particleSystem.emissionRate = 10;
                Instantiate(skidmark,skidmarkPos,transform.rotation);
        }else{
                bl.particleSystem.emissionRate=0;
        }
        if(br.GetGroundHit(out hitt)){
                skidmarkPos = hitt.point;
                skidmarkPos.y+=0.2f;
                br.particleSystem.emissionRate = 10;
                Instantiate(skidmark,skidmarkPos,transform.rotation);
        }else{
                br.particleSystem.emissionRate=0;
        }
        if(fr.GetGroundHit(out hitt)){
                skidmarkPos = hitt.point;
                skidmarkPos.y+=0.2f;
                fr.particleSystem.emissionRate = 10;
                Instantiate(skidmark,skidmarkPos,transform.rotation);
        }else{
                fr.particleSystem.emissionRate=0;
        if(fl.GetGroundHit(out hitt)){
                skidmarkPos = hitt.point;
                skidmarkPos.y+=0.2f;
                fl.particleSystem.emissionRate = 10;
                Instantiate(skidmark,skidmarkPos,transform.rotation);
        }else{
                fl.particleSystem.emissionRate=0;
} else {
        fr.particleSystem.emissionRate = 0;
        fl.particleSystem.emissionRate = 0;
        br.particleSystem.emissionRate = 0;
        bl.particleSystem.emissionRate = 0;
                }
velo = rigidbody.velocity.sqrMagnitude;
```

Zuerst wird überprüft, ob der entsprechende Wheelcollider den Boden berührt. Sollte dies der Fall sein, so erält das Wheelhit-Attribut hitt den Berührungspunkt als Wert. Diese Position wird nochmals von der skidmarkPos übernommen, die den Punkt repräsentiert, auf dem später die Bremsspur gelegt wird. Die Position von skidmarkPos wrd dabei noch etwas in y-Richtung erhöht, damit die Bremsspur später nicht von der Straße überdeckt wird. Am Ende erstellen wir die die Plane mit der Bremsspur-Textur, im Code als skidmark bezeichnet,

an der Position von skidmarkPos. Die Rotation wird dem Fahrzeug angepasst. Zudem wird der Bremsrauchmit der entsprechenden Emission erzeugt. Sollte der Wheelcollider nicht den Boden berühren, so entsteht weder Bremsspur noch Bremsrauch.
Während der Testphase stellt es sich als Problem heraus, dass die Bremsspuren nicht gelöscht werden. Da es sich bei diesen um Objekte handelt, kann dies auf lange Sicht ein Problem werden, weswegen ein Weg gefunden werden musste, wie man die Planes zerstören könnte. Letzendlich konnte das Problem gelöst werden, indem der Datei der Plane, die erzeugt wird, ein Skript angehangen wird. Dieses besteht nur darin, dass sich das Objekt nach 5 Sekunden wieder auflöst.

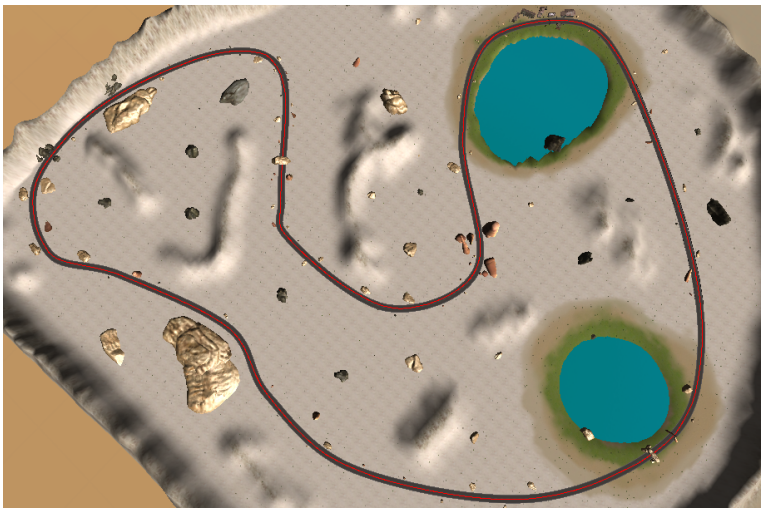
8. Spiellogik

Autor Alexander Tessmer

Die Spiellogik bestimmt den sinnvollen Ablauf des Spiels.
Sie zählt die Zeit von Start bis Ende und rundenweise, sowie die Runden selbst und sorgt für ein gesichertes Abfahren der Runden.
Dies wird dann im HUD, dem *Head-up-Display*, angezeigt.

Streckenablauf

Der gesicherte Ablauf und die Rundenerkennung erfolgt durch die Überprüfung, ob man noch in einem sinnvollen Abstand vom nächsten Streckenpunkt ist.
Dafür wurde die Strecke mit leeren Objekten gespickt und so der Streckenpfad definiert.



Streckenverlauf der Wüstenstrecke

Respawn

Im Falle eines gravierenden Fahrfehlers, wie Fahren ins Wasser oder in den Abgrund, wird der Spieler im letzten befahrenen *Checkpoint*, einem der Streckenmarkierer, *respawned*.
Außerdem besteht die Möglichkeit, sich per Knopfdruck zurückzusetzen, damit man nicht verzweifelt, wenn man steckenbleibt.

9. KI

Autor Alexander Tessmer

KI bezeichnet in diesem Fall einen pseudo-intelligenten Algorithmus, der menschliches Verhalten für eine bestimmte Aufgabe simuliert.
In diesem Fall die Steuerung eines Fahrzeugs in einem Rennspiel.
Für das Erstellen einer KI sind vier Arbeitsschritte relevant.
Diese können beliebig stark ausgebessert werden.
Sie werden im Folgenden näher erläutert.

Situationsanalyse

Grundverhalten

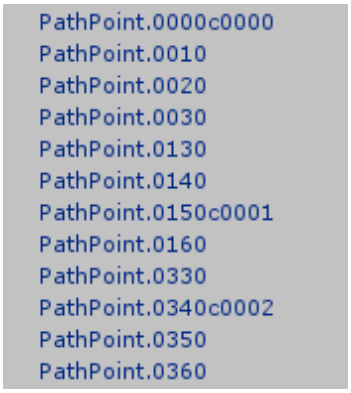
Sonderverhalten

Balancing

9.1 Situationsanalyse

Autor Alexander Tessmer

Damit die KI sich zurechtfinden kann, um die derzeitige Aufgabe zu erfüllen, braucht sie viele Orientierungspunkte. Wo der Mensch in etwa 15 *Checkpoints* benötigt, um die Strecke zwangsweise sinnvoll abzufahren, braucht die KI mindestens 200.



Liste der Wegpunkte


Leerer Wegpunkt

Hiermit wurde ein Wegpfad definiert, allerdings braucht die KI für ein logisches Abweichen vom mittigen Pfad am besten fünf Pfade, also 1000 *Checkpoints*. Diese haben wir allerdings nicht mehr erstellt. Unsere KI weicht den anderen Spielern bei Bedarf relativ zu deren Position aus, da sie dies aber blind tun muss, kann es hier leichter zu Fehlern, wie Collisionen, kommen. Das mittige Fahrverhalten wird im Balancing kaschiert.

9.2 Grundverhalten

Autor Alexander Tessmer

Das Grundverhalten in diesem Rennspiel besteht aus dem normalen Abfahren der Strecke.

Die KI soll menschliches Verhalten im Spiel simulieren, daher benutzt man am besten Methoden die einen Tastendruck simulieren und lässt diese dann von der KI ausführen.
Das Verhalten sollte möglichst verallgemeinert werden. Das bedeutet in diesem Fall, dass die KI im Stande ist ein beliebiges Ziel direkt anzufahren und getrennt davon das nächste Streckenziel zu bestimmen.
Das allgemeine Fahren besteht aus drei Abschnitten. Zuerst wird die benötigte Geschwindigkeit berechnet. Daraufhin wird die Art der Beschleunigung, also Vollbremse, Turbo oder Beschleunigen vorwärts oder rückwärts, bestimmt und letzten Endes folgt die Abfrage für das Lenken.

```
void lenken (){
       Vector2 right = new Vector2 (transform.right.normalized.x, transform.right.normalized.z);
        if(Vector2.Dot(right, target.normalized)>lenkFaktor){
                pressRight();
        }
        if(Vector2.Dot(right, target.normalized)<-lenkFaktor){
                pressLeft ();
        }
}
```

Diese Abfragen benutzen die Informationen aus der Situationsanalyse, um zu bestimmen, welches Fahrverhalten derzeit durchgeführt werden soll.
Sie benutzen meist Distanz- oder Winkelabfragen.
In diesem Fall sorgt der Lenkfaktor für gleichmäßigeres Fahren.

Wo man mit deutlich mehr Aufwand ein Fahrverhalten anhand mehrerer kommender *Checkpoints* bestimmen könnte, benutzt unsere KI nur den jeweils nächsten *Checkpoint* und visiert diesen direkt an.
Der jeweils neue *Checkpoint* wird bestimmt, wenn der alte *Checkpoint* erreicht wird. Hierbei ist zu beachten, dass sich die Checkpoints für KI und Mensch unterscheiden und die jeweilige Erkennung wird durch eine einfache *Regex*-Abfrage anhand vorher abgestimmter Namensmerkmale durchgeführt.

9.3 Sonderverhalten

Autor Alexander Tessmer

Durch das Grundverhalten kann die KI nun die Strecke simple abfahren. Dies reicht allerdings nicht. Das Sonderverhalten kümmert sich um spezielle Situationen, wie eine falsche Ausrichtung oder eine Collision in Fahrtrichtung, sowie diverse spezielle Verhalten, die durch bestimmte Spielinhalte, wie *PowerUps*, nötig werden. Außerdem wird auch das Zusammenspiel mit anderen Spielern mit einbezogen.

Beginnen wir mit der falschen Ausrichtung. Standartmäßig würde die Ki einfach rückwärts fahren, da eine negative Geschwindigkeit benutzt werden muss, um ein Ziel hinter der KI zu erreichen. Da dies unpraktisch ist, muss die KI wenden. In unserem Fall ist dies durch eine einfache Drei-Punkt-Wende implementiert.

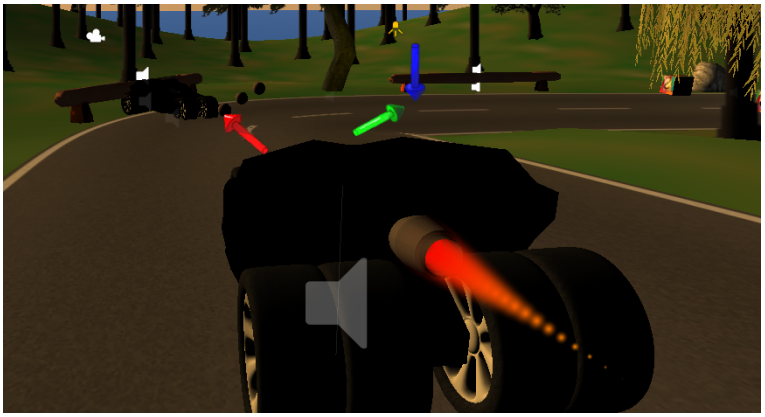
Drei-Punkt-Wende der KI

Die KI sucht sich in diesem Fall einen neuen Zielwegpunkt rechts hinter der KI. Dadurch fährt sie rückwärts und schlägt nach rechts ein.
Steht sie nun wieder mit maximal 80 Grad in die Richtung vom normalen Wegpunkt, ist der Sonderfall vorbei. Die KI lenkt nach links und fährt vorwärts normal weiter.

Für eine Collision mit dem Terrain in einem Rennspiel müsste man die derzeitige Fahrtrichtung wechseln und eine Wende mit einem geringeren Wendwinkel durchführen. Da dies auch während einer Drei-Punkt-Wende, also innerhalb eines Sonderfalls passieren kann oder auch innerhalb des Sonderfalls der Collision selbst, wird dies rekursiv immer schwerer zu implementieren.
Eine einfache Abhilfe war in diesem Fall einfach ein *respawn* sollte die KI nicht mehr in der Lage sein sich zu bewegen.

Da wir mit nur einem Wegpfad nur relative Abstände von anderen Spielern zur KI bestimmen können und nicht wirklich Abstände in Streckenrichtung, gestaltet es sich als äußerst schwierig auf andere Spieler intelligent zu reagieren. In unserem Fall wird einem anderen Spieler einfach ausgewichen oder es werden intelligent spielerbezogene *PowerUps* eingesetzt.

Sollte die KI das Abschiessen *PowerUp* besitzen, wartet sie ab, bis ein Spieler in ihre Nähe kommt. Ist dies der Fall, wird überprüft, ob sich der andere Spieler in einem vertretbaren Winkel vor der KI befindet und sollte dies zutreffen, startet das Sonderverhalten für das Abschiessen *PowerUp*.



KI benutzt Abschiessen *PowerUp*

In dem Bild steht der blaue Pfeil für den bisherigen nächsten Streckenwegpunkt und der grüne Pfeil für den Fahrtvektor am Anfang oder Ende des Sonderverhaltens.
Der rote Pfeil ist der gesuchte Abschussvektor.

Innerhalb dieses Sonderverhaltens richtet sich die KI nun schnellstmöglich zum anderen Spieler aus, um dann auf diesen zu feuern. Danach geht die Fahrt in Richtung nächster Streckenwegpunkt normal weiter.

9.4 Balancing

Autor Alexander Tessmer

Eine KI wird standartmäßig so entworfen, dass sie ihre Aufgabe perfekt erfüllt, dass sorft dafür, dass zum Beispiel die Lenkung anfängt hin und her zu flimmern, da die KI extrem schnell die Richtung ändert, um die Zielrichtung einzuhalten. Dies ist natürlich für eine menschliche Handlung, die simuliert werden soll, unnatürlich. Daher müssen Faktoren her, mit denen die mathematischen Abfragen modifiziert werden, um Verzögerungen oder Ungenauigkeiten einzubauen.

Zwei Arten von Faktoren sind hier relevant. Zum Einen die festen Faktoren und zum Anderen die Zufallsfaktoren.

Ein Beispiel von einem festen Zufallsfaktor sieht man in der Lenkmethode vom Grundverhalten. Hier wird das mittige fahren, sowie eine flimmernde Lenkung, durch einen Lenkfaktor in der scalaren Richtungsabfrage

verdeckt, in dem ein ungenaues Fahren simuliert wird. Ein fester Zufallsfaktor wird durch Testen auf realistisches Aussehen bestimmt und bleibt dann konsistent erhalten.
Feste Zufallsfaktoren werden verwendet, wenn Modifikationen nur dafür sorgen würden, dass das Fahrverhalten nicht mehr realistisch wirkt oder es sich gar nicht sichtbar verändert.

Im Gegensatz zu den festen Faktoren, ändern sich die Zufallsfaktoren in Zusammenhang mit den KI Schwierigkeitsgraden und werden für jeden sinnlogischen Aufruf zufällig neu modifiziert.

Als Erstes bestimmt man einen festen Zufallsfaktor, um das Verhalten so zu modifizieren, sodass die KI noch gut funktioniert und das Verhalten realistisch aussieht.
Dann werden Fahrfehler für die verschiedenen Schwierigkeitsgrade simuliert, in dem man die vorher bestimmten festen Zufallsfaktoren anhand des Schwierigkeitsgrades verändert.

```
int useTurboBrakeFaktor=4;
float driveSpeedFail=0.1f;
int maxSpeedFaktor=12; // +/- 12

void setRandomness (){
        useTurboBrakeFaktor+=Difficulty;
        driveSpeedFail+=Difficulty/4f;
        maxSpeedFaktor -= Difficulty * 3;
}
```

Das menschliche Reaktionen nicht jedesmal gleich ausfallen, kann man simulieren, in dem man jedes mal, wenn ein sinnlogischer Aufruf von einem Zufallsfaktor passieren soll, stattdessen eine Zufallsmethode ausführt, die einen zufällig modifizierten Zufallsfaktor zurückliefert. Sie erwartet den Zufallsfaktor der modifiziert werden soll und einen Modifikator, damit die zufällige Veränderung innerhalb einer sinnvollen Stärke bleibt.

```
float randomRoll(float faktor, float modifikator){
        int zufall = Random.Range(0, 100);
        return faktor + zufall/modifikator;
}
```

10. Multiplayer

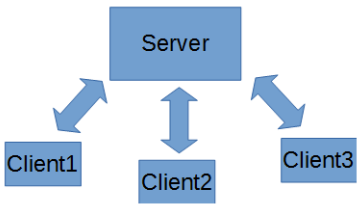
Autor: Frederic Marchand

Unity bietet uns eine gute Grundlage um einen Multiplayer in unser Spiel zu implementieren. Es stellt etliche Methoden zur Verfügung, die bei bestimmten Ereignissen während unseres Programmablaufs aufgerufen werden und mit deren Hilfe wir bestimmt Aktionen der Spiellogik ausführen können.

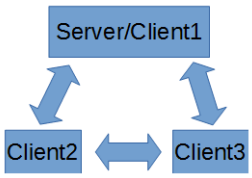
10.1 Konzepte

Autor: Frederic Marchand

Eine der ersten Fragen, die wir uns stellen mussten war wie wir unseren Multiplayer implementieren wollen. Dabei hatten wir die Wahl zwischen zwei Konzepten. Das Erste ist das des autoritativen Servers, der eine Schnittstelle zwischen den Clienten darstellt, sodass eine Kommunikation unter den Clienten nur über den Server möglich ist. Dieses Konzept ist das sichere von den beiden Möglichen, da hier der Server nur anhand der Tasteneingaben von den Clienten die Berechnungen vornimmt und danach die Veränderungen an den Clienten zurück übermittelt.



Im Gegensatz dazu steht das Konzept des non-autoritativen Servers. Hierbei verarbeiten die Clienten die Eingabe der Spieler anhand der Spiellogik eigenständig und schicken dann die ermittelten Daten an den Server weiter. Dieser synchronisiert die Daten und teilt sie den anderen Clienten mit. Leider haben die Clienten dadurch die Möglichkeit die Spielumgebung zu ihren Gunsten zu beeinflussen und sich so einen Unfairen Vorteil gegenüber den anderen Mitspielern zu verschaffen.



Jedoch haben wir uns trotzdem für die unsichere Variante entschieden, da wir sonst innerhalb der begrenzten Zeit und der geringen Erfahrung im Umgang mit Unity kein vorzeigbares Ergebnis hätten fertig stellen können.

10.2 NetworkView

Autor: Frederic Marchand

Essenzieller Bestandteil für den Multiplayer ist die NetworkView. Sie kann als Komponente an Objekte in unserer Spielwelt angehängt werden um Veränderungen an diesen zu bemerken oder diese zu beeinflussen. Die Beeinflussung ist durch den Zugriff auf Komponenten, die an das Objekt angeheftet wurden, möglich, so kann zum Beispiel die Position der Rigidbody oder auch Variablen innerhalb eines angehefteten Skripts verändert werden.
Wir unterscheiden zwei Arten von Kommunikation zwischen Clienten und Server.

Die erste und auch die gewöhnlich verwendete Variante ist die der Zustands Synchronisation. Sie übermittelt, je nachdem wie die NetworkView eingestellt ist, unterschiedlich die Datenpakete an die anderen Teilnehmer im Netzwerk. Es gibt drei mögliche Optionen. Zum einen gibt es die Möglichkeit die NetworkView aus zu stellen. Diese wird zum Beispiel benötigt, falls es sich dabei um ein GameObject handelt, das nur zum managen des Spiels da ist und die Fähigkeit besitzen soll mit den anderen Objekten, die auch eine NetworkView Komponente angehängt haben. Eine weitere Option ist es die NetworkView auf „Unreliable“ einzustellen. Hierbei schickt sie so oft es geht den aktuellen Zustand des Objektes an dem sie befestigt wurde. Die letzte Variante ist die NetworkView auf „Reliable Delta Compressed“ einzustellen, welche wir auch bei einen großteil unserer Objekte verwendet haben. Dieser Modus sorgt dafür, dass die NetworkView ständig überprüft ob sich etwas am Zustand des Objektes geändert hat, falls dies zutrifft werden nur die geänderten Werte an die Teilnehmer im Netzwerk weiter gegeben. Dadurch kann bei der Übertragung der Daten sehr viel Bandbreite eingespart werden, was die Kommunikation effektiver gestaltet.
Ein Beispiel für die Verwendung finden wir, wenn wir unsere Autos anschauen. Sie sind auf „Reliable Delta Compressed“ eingestellt und werden den aktuellen Zustand mit dem Zustand davor vergleichen, falls sich dabei etwas geändert hat, wird dies weiter an die anderen Clienten gegeben.

Die zweite Variante der Kommunikation ist die des Remote Procedure Calls. Hiermit können wir Methoden bei anderen Teilnehmern im Netzwerk aufrufen. Dazu muss beim Aufrufer der Methodenkopf mit einem Vorgestellten RPC- Tag und beim Aufgerufenen eine Methode mit einem identischen Methodenkopf vorhanden sein. Mit diesem Aufruf können wir unbegrenzt viele Parameter an andere Teilnehmer weitergeben. Gleichzeitig haben wir die Möglichkeit anzugeben mit wem wir kommunizieren wollen. Also ob wir nur beim Server, bei allen außer dem Server oder bei allen eine Methode ausführen wollen. Dieser Aufruf wird hauptsächlich bei unregelmäßig auftauchenden Ereignissen verwendet. Ein Beispiel für die Verwendung des RPC Aufrufes innerhalb des Rennspiels ist das des Unsichtbarkeits-Power-Ups.(siehe Unsichtbarkeit)

10.3 Multiplayer im Rennspiel

Autor: Frederic Marchand

In Unity existiert bereits die Grundzüge eines Multiplayers. Daher war der Anfang recht schnell erstellt. Jedoch beschränkten sich diese darauf einen Server zu erstellen und mit Hilfe einer separaten Anwendung sich mit diesen Server zu verbinden. Die Schwierigkeit war es nachdem ein Server am laufen war auch unsere Spielobjekte zu erstellen und jedem Spieler richtig zu zuordnen. Aber auch einen gewissen Komfort zu bieten, damit das Miteinander spielen auch Spaß macht. Dazu haben wir zu erst eine Oberfläche erstellt mit diversen Buttons mit denen ein Server erstellt, auf ein Server verbunden oder Optionen zur Grafik verändert werden können.

Des Weiteren wurde ein Masterserver, der auch schon von Unity gegeben war, eingebaut. Er organisiert die Verbindung zwischen Server und Clienten. Der Masterserver trägt jeden neuen Server auf eine sogenannte HostList ein, die wiederum von den Clienten abgefragt werden kann.

Um einen Server zu erstellen wird direkt beim drücken des jeweiligen Buttons eine standardisierte maximale Spieleranzahl, in unserem Fall sind es vier Spieler, und einen Port zugewiesen. Danach wird der Server mit einem Spielnamen und einen vom User selbst gewählten Servernamen auf der HostList des Masterservers registriert.

Falls jetzt ein Spieler sich mit dem soeben erstellten Server verbinden möchte muss er erst einmal vom Masterserver die HostList anfordern. Das Anfordern geht über einen weiteren Button auf unserer Benutzeroberfläche. Nachdem der Button gedrückt wurde kann eine Liste durch scrollt werden die aus den derzeit offenen Servern im Netzwerk besteht.

Hinzu kommt ein Pausen Menü, das im Spiel mit einem Druck auf die Escape Taste aufgerufen werden kann. Hierbei hat der Spieler, der auch gleichzeitig den Server darstellt, die Möglichkeit mit einem „ShutDown“ Button den Server zu beenden. Falls dieser Button gedrückt wird sollen alle Objekte, die als Player getagt wurden, gelöscht werden und der Server soll disconnecten. Auf der Clientseite hingegen gibt es ein „Disconnect“ Button. Er sorgt dafür, dass die Clients vom Server jeder Zeit ausloggen können. Danach werden beide wieder zurück in zum ersten Menü befördert.

10.4 NetworkManager

Autor Nils Baumgartner

Aufgaben

Der Networkmanager ist zuständig für den Ablauf im gesamten Mehrspieler.
Auf Basis des „Non-autoritativen Servers“ bekommt jeder Spieler alle wichtigen Informationen um einerseits selber einen Server erstellen zu können, als auch die Handlungen der anderen Spieler zu erhalten.
In unserer Implementierung wird jeweils bei allen Spielern (dem Host mit eingeschlossen) beim Beitritt ein neuer Spieler erstellt, welcher mittels einer „Networkview ID“ eindeutig einem Spieler zuzuordnen ist.
Mittels „Network Methoden“ ist es möglich bei "Instanziierung" von Objekten diese ebenfalls bei allen weiteren Mitspielern zu erstellen. Hierbei wird dank der „non-autoritativen Servers „ Implementation ein Umweg über den Host nicht nötig.
Nun folgt ein kleiner Auszug der signifikanten Stellen aus dem „Networkmanager Script“

Spiel Beitritt

Bei dem beitritt zu einem Server wird eine Methode aus Unity überladen, damit Objekte bei allen Spielern erstellt werden.

```
private void OnConnectedToServer (){
                print ("Connected");
                SpawnPlayer ();
        }
```

Die „SpawnPlayer“ Methode "instanziiert" über die Network Klasse bei allen verbundenen zunächst ein Fahrzeug, welches bei späteren beitritt eines anderen Spieler ebenfalls getätigt wird.
Zunächst hat der verbindende Spieler seine ausgewählte Fahrzeug ID, wodurch das passende Fahrzeug erstellt wird.
Dabei wurde darauf geachtet, dass lediglich bei dem eigenen Fahrzeug die Steuerung aktiviert wird, wodurch die ungewollte Steuerung der Mitspieler ausgeschaltet wird.
Dies geschah in enger Zusammenarbeit mit der Fahrphysik, wodurch auch das KI verhalten möglich wurde.
Ebenfalls wird die Ausrichtung des Fahrzeugs angepasst an den vordefinierten Richtungen bei der Streckenerstellung.

```
private void SpawnPlayer ()
                {
                                print ("Spawning Player");
                                GameObject go;
                                switch (carNumber) {
                                case 0:
                                                go = Network.Instantiate (car [carNumber],
Vector3.up + spawns [Network.connections.Length].position,
                                                  spawns [Network.connections.Length].rotation,
 0) as GameObject;
                                            Car_Batmobil script_batmobil_new = go.GetComponent
 (typeof(Car_Batmobil)) as Car_Batmobil;
                                                script_batmobil_new.used (false);
                                                break;
                                . . .
```

Ein weiterer wichtiger Aspekt wurde in dem obigen Quellcode mitberücksichtigt, wodurch Spieler an ihrer passenden Position erstellt werden, welche bei unserer Implementation von der Beitrittsnummer abhängig gemacht wurde.
Neben dem „Ingame Menü“ welches an dieser Stelle nicht weiter angesprochen wird, da es im Kapitel vorher schon angesprochen wurde, kommt noch das Problem bei einem Austritt eines Spielers.

Spiel Austritt

Bei einem regulären Austritt eines Spielers aus dem Spiel („Disconnect Button“) als auch bei einem nicht vorgesehenen, so wird gewährleistet, dass das "instanziierte" Fahrzeug nicht weiter existiert bzw. gelöscht wird , da sonst ungewollte Objekte in dem Spiel vorhanden bleiben.
Dies wird umgesetzt, indem bei jedem Spieler eine überladene Methode von Unity aufgerufen wird.

```
private void OnPlayerDisconnected (NetworkPlayer player)
                {
                                print ("Player Disconnected");
                                destroyAllFromPlayer (player);
                }


private void destroyAllFromPlayer (NetworkPlayer player)
                {
                                Network.RemoveRPCs (player);
                                Network.DestroyPlayerObjects (player);
                }
```

Hierbei wird implizit darauf geachtet nur Objekte des verlassenen Spielers zu entfernen.

10.5 Verwaltung

Hauptanwendung

Autor : Sebastian Dieckmann

Ursprünglicher Gedanke war es, die verschieden Karten, jede als eigene Szene, in einem Unity Projekt Zusammenzufügen, wodurch ein sehr leichtes wechseln zwischen jenen möglich gewesen wäre.
Nach anfänglichen Schwierigkeiten, wie zum Beispiel das Überschreiben von Terrains, hatte man 2 Karten zusammengelegt, aber beim näheren Hinschauen wurden die nächsten Fehler sichtbar, es hatten sich nämlich auch einige Texturen überschrieben. Da eine Lösung für dieses Problem wahrscheinlich zu viel Zeit in Anspruch genommen hätte, entschieden wir uns dazu, eine Hauptanwendung, also eine *Launcher* Anwendung, und für jede Karte eine eigene Anwendung zu erstellen, die von ersterer ausgeführt wird. Die erforderliche Kommunikation zwischen diesen, zum einen die Übergabe der Nummer des vorher ausgewählten Autos, zum anderen die Auflösung und die Qualität der Hauptanwendung, die von der Kartenanwendung übernommen wird, lösten wir über die Kommandozeilenargumente. Dazu werden erst die eigenen Werte ermittelt, dann mit einem *Stringbuilder* zusammengesetzt und als einen gesammelten *String* übergeben. Nach dem Starten werden diese Argumente in einem *Stringarray* gespeichert und die einzelnen Werte sind der Reihe nach abrufbar und ermöglichen ein Setzen der erforderlichen Variablen.

Beispiel für die Übergabe

```
void KZAOutput ()
{ // Beispiel fuer das Erstellen von Kommandozeilenargumente, die uebergeben werden

        // Eigene Werte ermitteln
        int swidth = Screen.width;
        int sheight = Screen.height;
        bool sfull = Screen.fullScreen;
        int quallevel = QualitySettings.GetQualityLevel ();

        int car = index;

        Process mapprocess = new Process ();
        System.Text.StringBuilder sb = new System.Text.StringBuilder ();

        sb.Append (true);
```

```
        sb.Append (' ');
        sb.Append (car);
        sb.Append (' ');
        sb.Append (swidth);
        sb.Append (' ');
        sb.Append (sheight);
        sb.Append (' ');
        sb.Append (sfull);
        sb.Append (' ');
        sb.Append (quallevel);

        string args = sb.ToString ();

        mapprocess.StartInfo.FileName = ".\\Maps\\Berg.exe";
        mapprocess.StartInfo.Arguments = args;
        mapprocess.StartInfo.WindowStyle = ProcessWindowStyle.Normal;
        mapprocess.Start ();
}

void KZAInput ()
{ // Beispiel fuer das Einlesen von Kommandozeilenargumente

        string[] args = System.Environment.GetCommandLineArgs ();

        // Einlesen der Variablen in der richtigen Reihenfolge
        if (args.Length >= 7) {

                bool multi = Convert.ToBoolean (args [1]);

                int carNumber = Convert.ToInt32 (args [2]);

                int swidth = Convert.ToInt32 (args [3]);
                int sheight = Convert.ToInt32 (args [4]);

                bool sfull = Convert.ToBoolean (args [5]);

                int quallevel = Convert.ToInt32 (args [6]);

                // Setzen der Aufloesung und Qualitaet
                Screen.SetResolution (swidth, sheight, sfull);
                QualitySettings.SetQualityLevel (quallevel);
        }
}
```

Playersettings

Autor : Sebastian Dieckmann

In den *Playersettings*, kann man verschiedene Einstellungen vornehmen, die in dem danach gebauten *Standalone Player*, also dem eigentlichen Spiel, zum tragen kommen. Zu finden sind diese Settings entweder unter *Edit -> Project Settings -> Player* oder indem man bei den *Build Settings* auf *Player Settings* klickt.
Zunächst einmal kann man an dieser Stelle die *Default-Werte* für den Bildschirm bestimmen, also die Höhe und Breite sowie die Auswahl zwischen Vollbild- und Fenstermodus.
Darunter befindet sich die Option *Run in Background* welche ein gleichzeitiges Ausführen von zwei *Unity Anwendungen* auf einem PC ermöglicht.
Außerdem gibt es die Möglichkeiten ein größenveränderbares Fenster zu erstellen oder Spiele die im Vollbildmodus laufen hervorzuheben, indem sekundäre Darstellungen verdunkelt werde. Des weiteren kann man erzwingen, dass nur eine Unity Anwendung ausgeführt werden darf, jedes weitere geöffnete Spiel wird mit einer Fehlermeldung beendet.
Zum Schluss gibt es die Option beim Starten der Anwendung ein Fenster anzeigen zu lassen, in dem man die Auflösung, Qualität und die Auswahl zwischen Vollbild- und Fenstermodus vorgenommen werden kann.
In der Hauptanwendung ist dieses Fenster aktiviert, in jeder anderen Anwendung, also Kamerafahrt und Karte, ist es deaktiviert, da diese Fenster die Werte der *Launcher* Anwendung übernehmen.

Ordnerstruktur

Autor : Sebastian Dieckmann

Ein weiterer Aspekt dieses Systems, ist die richtige Ordnerstruktur, also wo welche Anwendungsdatei hingehört. In unserem Fall sind die Kartenanwendungen in einem Ordner namens *Maps*, der im selben Verzeichnis wie die Hauptanwendung sein muss. Die Dateien der Kamerafahrt sind in einem Unterordner der *Maps* namens *Kamerafahrt*. Eine richtige Benennung der Datei ist natürlich auch wichtig, eine Datei heißt zum Beispiel *wüstecamera*. Natürlich könnte man diese Namen auch ändern, aber damit diese veränderten Dateien auch aufgerufen werden, müsste man in einem Script den richtigen Namen ändern und dann die Anwendung von Unity noch einmal komplett neu bauen lassen, was, je nach Anwendung, ziemlich lange dauern kann.

11. PowerUps

Autor: Dennis Altenhoff

Um das Spiel interessanter zu gestalten, wurden *Power-Ups* zum Spiel hinzugefügt.
Power-Ups sind temporäre Fähigkeiten, die man einsammeln und verwenden kann, um entweder Vorteile für sich selbst oder Nachteile für die Gegenspieler zu erreichen.

Jedes Fahrzeug hat einen eigenen *Code*, der durch kleine Änderungen auf das entsprechende Modell zugeschnitten wurde, da manche *Power-Ups*, je nach Fahrzeugeigenschaften, unterschiedliche Reaktionen hervorrufen.

Für das Spiel wurden folgende *Power-Ups* eingeführt:

- 1. Der Geschwindigkeitsboost
- 2. Die Unsichtbarkeit
- 3. Die Skalierung
- 4. Das Geschoss
- 5. Der Impuls

Probleme beim Integrieren in den *Multiplayer*

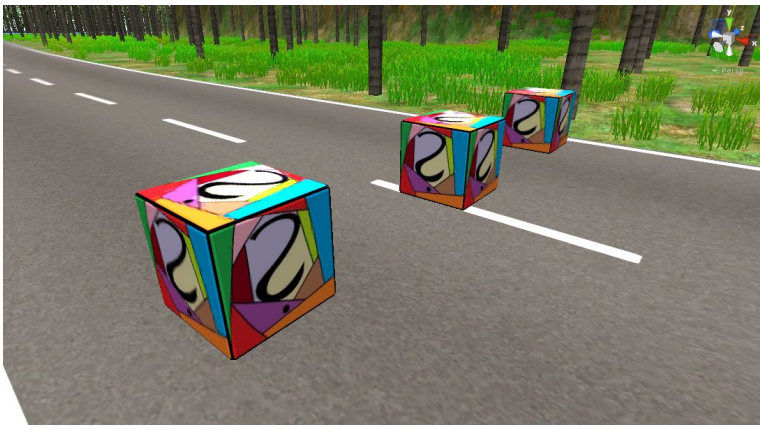
Beim Einfügen des *Power-Up-Skripts* in den *Multiplayer-Modus* ergaben sich diverse Probleme, da immer nur die Physik, aber nie das tatsächliche Aussehen der Autos übermittelt wurde. Daher musste über *RPC-Methoden* die Änderungen an dem Auto in allen anderen *Network-Views* ebenfalls vorgenommen werden, um zu den gewünschten visuellen Effekten zu führen. (Für Nähere zu *RPC-Methoden* und *Network-Views* siehe Multiplayer)

11.1 PowerUpBoxen

Autor: Michael Kaufmann

Die *PowerUpBoxen*

In einem Spiel in dem es *PowerUps* geben soll, muss es auch Möglichkeiten geben an diese heranzukommen. Unsere Idee um an jene heranzukommen ist es, in unregelmäßigen Abständen auf der Strecke Boxen zu platzieren, welche beim hindurchfahren *PowerUps* verleihen, dann verschwinden und nach einiger Zeit wieder auftauchen.
Die So genannten *PowerUpBoxen*.



Drei *PowerUpBoxen* auf einem Abschnitt der Vulkanstrecke

Bauart

Für die *PowerUpBoxen* haben wir in *Unity* integrierte Würfel Objekte genommen, und auf die eine in *Microsoft Paint* modellierte Textur gelegt.
Anschließend haben wir ein *Skript* an die Boxen gehängt welches dafür sorgt dass diese in regelmäßigen abständen hoch und runter hüpfen, und sich langsam im Uhrzeigersinn drehen.
Dies geschieht ganz einfach indem wir in der *Update-Methode*, welches eine Methode ist die jeden *Frame* (jedes mal wenn ein neues Bild kommt, bedeutet bei 60 FPS (Frames per Second) 60 mal in der Sekunde) ausgeführt wird.

Zum detaillierten Code folgen Sie folgendem Link:
https://github.com/tbourdon/CGPrak14-Unity/commit/33820d39282217e992188c9a02c6aa1633ee09a7

Respawn

Um zu *despawnen* (verschwinden) und zu *respawnen* (wiederzukommen) Bedarf es eines weiteren Skripts. Im Prinziep macht es, dass sowohl die *Collider* der *PowerUpBox* als auch der *Renderer* ausgeschaltet werden, und somit die *PowerUpBox* sowohl visuell als auch als Gegenstand verschwindet, wenn ein als *Player getagtes GameObject* mit ihr in Berührung kommt. Sobald Sie dann verschwunden ist fängt ein *Timer* von 5 Sekunden an runter zu zählen, und sobald er auf 0 ist werden *Renderer* und *Collider* wieder aktiviert.

Zum detaillierten Code folgen Sie folgendem Link:
https://github.com/tbourdon/CGPrak14-Unity/commit/15721de1f2be0ac835a6ed0c6c29774c9ca95c3a

11.2 Boost

Autor: Alexander Altemöller

Anders als beim Turbo, welcher durch gedrückt halten der Shift-Taste getriggert und auch wieder beendet werden kann, beschleunigt der Geschwindigkeitsboost das Fahrzeug für 3 Sekunden lang auf eine konstante, aber deutlich höhere Geschwindigkeit.

Zu Beginn aufgetretene Probleme und unsere Lösungen

Ein Problem, welches aufgetreten ist, war, dass das Fahrzeug nach Ablauf der 3 Sekunden seine Boost-Geschwindigkeit behielt. Die Lösung hierfür war glücklicherweise recht trivial und bestand darin die aktuelle Geschwindigkeit vor der Boost-Nutzung abzuspeichern und nach Ablauf der 3 Sekunden wieder zu setzen.

Ein etwas komplexeres Problem war die Lenkung des Fahrzeuges während des Boostes. Um eine korrekte Lenkung weiterhin zu gewährleisten, mussten wir direkt auf das Fahrphysik-Skript des entsprechenden Autos zugreifen und die aktuelle Geschwindigkeit an dieses weitergeben.
Das hatte leider zur Folge, dass wir für jedes Fahrzeug ein eigenes Skript schreiben mussten, da jedes Fahrzeug eine vollkommen individuelle Fahrphysik besitzt.

Wie berechnet sich die neue Boost-Geschwindigkeit?

Um zu gewährleisten, dass ein Fahrzeug nicht überproportional schnell wird, ist es nicht möglich, die aktuelle Geschwindigkeit einfach mit einer Konstanten zu multiplizieren. Stattdessen wird ein Boost-Vektor berechnet, welcher in Abhängigkeit von der aktuellen Fahrzeuggeschwindigkeit größer oder kleiner wird.
Dieser Boost-Vektor wird nun mit der aktuellen Fahrzeuggeschwindigkeit multipliziert und ergibt somit die neue, schnellere Boost-Geschwindigkeit.

Der etwas umfangreichere Programmcode für den Boost-Algorithmus, lässt sich bei Interesse hier finden: https://github.com/tbourdon/CGPrak14-Unity/blob/master/unity_terrain/Geschwindigkeitsboost .

11.3 Unsichtbarkeit

Autor: Dennis Altenhoff

Die Idee

Die Idee war, wie der Name selbst erklärt, das der Spieler unsichtbar werden soll. Dies wird dadurch erreicht, dass der Renderer des Autos temporär ausgeschaltet wird. Nach einiger Zeit wird er wieder eingeschaltet und das Auto wieder sichtbar.

Etwa eine Sekunde vor dem kompletten wieder einschalten, sollte das Fahrzeug anfangen aufzublinken, sodass man weiß, dass das *Power-Up* bald endet.

Probleme und Lösungen

Das Problem das sich ergab, war, dass die einzelnen Autos nicht als einzelnes GameObject in der *Unity-Hierarchie* auftauchten, sondern viele "Unter-Objekte" besaßen, bei denen der *Renderer* ausgeschaltet werden musste. Die Lösung war, alle *Childs*, also eben jene "Unter-Objekte", einzeln anzusprechen und ihre *Renderer* umzuschalten.
Um das Blinken zu erzeugen, wurde, sobald nur noch eine Sekunde über ist, in der *Update-Methode*, die jedes Mal aufgerufen wird, wenn ein *Frame* geladen wird, der aktuelle *Renderer-Zustand* dieser "Unter-Objekte" invertiert.

Das fertige *Power-Up*

Fertig implementiert ergab sich bei Verwendung des Power-Ups schließlich:

11.4 Skalierung

Autor: Alexander Altemöller

Vergrößerung

Hierbei wird das Fahrzeug mit dem Faktor 10 vergrößert und die Masse entsprechend erhöht. Das hat den Vorteil, dass das Fahrzeug nun schwerer ist, und andere Fahrzeuge besser von der Straße drängen kann.

Vergleich zweier Ferraris.
Das hintere Fahrzeug ist groß skaliert, das vordere ist normal.

Verkleinerung

Hierbei wird das Fahrzeug mit dem Faktor 10 verkleinert und die Masse entsprechend verringert. Der Vorteil hierbei ist, dass das Fahrzeug analog zum Geschwindigkeitsboost schneller wird, und somit höhere Gewinnchancen hat.

Vergleich zweier Ferraris.
Das hintere Fahrzeug ist klein skaliert, das vordere ist normal.

Die Dauer dieser beiden Power-Ups beträgt 10 Sekunden.

Der Programmcode für den Vergrößerungs-Algorithmus lässt sich bei Interesse hier finden:
https://github.com/tbourdon/CGPrak14-Unity/blob/master/unity_terrain/Vergr%C3%B6%C3%9Ferung .

11.5 Geschoss

Autor: Dennis Altenhoff

Die Idee

Das Umsetzungsidee zu diesem *Power-Up* war die Instanziierung eines *Kugel-GameObjects* aus einem fertigen *Prefab* mit zugehörigem Skript, an einer festgelegten Position vor dem Auto (dem *Kugel-Spawn*). Diese Kugel besitzt einen rigidbody (Für Näheres dazu: Fahrphysik) und kann so mit einer Kraft versehen werden, die dafür sorgt, dass die Kugel "geschossen" werden kann.

Probleme und Lösungen

Da die Kugel einen *rigidbody* besitzt, wird sie von der Schwerkraft der Szene beeinflusst. Daher war es notwendig bei jedem geladenen *Frame*, die y-Position festzusetzen.
Zudem wären die Kugeln nicht verschwunden und hätten so zu Logik- und *Performance-Problemen* geführt. Dies wurde durch einen Timer und einen Kollisions-Zähler behoben.

Ebenso wurde auch das Problem des Nicht-Verschwindens des Geschosses nach einem Spieler-Treffer durch eine Abfrage gelöst, ob das andere Objekt ein Spieler ist.

Tritt eines dieser Ereignisse ein, wird *"Destroy"* ausgelöst und das Geschoss-Objekt zerstört.

Das fertige *Power-Up*

Das fertige *Power-Up* hatte schlussendlich folgenden Effekt:

Der *Code*

Der zum Abschießen gehörige Code findet sich hier:

```
/*
 * Programmiert wurde in C#.
 */
public class CollectPowerUp_Batmobil: MonoBehaviour {

        /*
         * Methode zum Verschießen der Kugeln.
         * Wird von der Koroutine aufgerufen.
         */
        void Shoot(){
                if(munition>0){
                        GameObject Geschoss =
((GameObject)Network.Instantiate(Resources.Load("Kugel"),
                                                        KugelSpawn.transform.position, new
Quaternion(), 0));
                        Geschoss.rigidbody.AddRelativeForce(transform.forward*schusskraft);
                        Geschoss.transform.parent = KugelSpawn.transform;
                        munition--;
                }
        }
}


/*
 * Programmiert wurde in C#.
 */
```

```
using UnityEngine;
using System.Collections;

/*
 * Das ShotScript fuer das Power-Up "Abschiessen".
 * Geschossene Kugeln verschwinden nach kurzer Zeit.
 */
public class ShotScript : MonoBehaviour {

        private readonly float power=10000;
        private float spawnTime;
        public float shotTime = 5;
        public int maxColl = 3;
        private int Coll = 0;

        // Use this for initialization
        void Start () {
                if (networkView.isMine) {
                        enabled = true;
                } else {
                        enabled = false;
                }
                spawnTime = Time.time;
        }

        // Update is called once per frame
        void Update () {
                rigidbody.velocity = new Vector3 (rigidbody.velocity.x,0,rigidbody.velocity.z);
                if ( Time.time >= spawnTime+shotTime || maxColl == Coll ){
                        Network.Destroy(this.gameObject);
                        print ("Destroy");
                }
        }

        /**
         * Kollidiert die Kugel, wird entsprechend gehandelt:
         * 1. Kollision mit einem Spieler: Schleudere ihn weg und zerstöre dich
         * 2. Erhöhe Kollsionszähler um 1
         */
        void OnCollisionEnter (Collision other) {
                print ("Kollision");
                int velBack = 10;
                Vector3 dir = (transform.position -
other.gameObject.transform.position).normalized;
                rigidbody.velocity = new Vector3(dir.x*velBack,0,dir.z*velBack);
                if(other.gameObject.tag == "Player"){
                        other.gameObject.rigidbody.AddExplosionForce(power,
other.gameObject.transform.position, 5f, 30.0f);
                        Network.Destroy(this.gameObject);
                        print("Player");
                }
                Coll++;
        }
}
```

11.6 Impuls

Autor: Michael Kaufmann

Der Impuls

Die Idee war, dass es ein *Power-Up* geben sollte welches einem ermöglicht Mitspieler auf kurze Distanz von dem Benutzer hinfort zu stoßen.
Wir kamen auf unterschiedliche Lösungsansätze, jedoch erwies sich nur Einer als umsetzbar.

Die Theorie

Nach einiger Überlegung war unser Plan schließlich folgender:
Zu Erst wird nach Zielen, das heißt *GameObjects* welche als *Playergetagt* sind gesucht.
Danach wird die Position des Benutzers des Powerups, und der jeweiligen Ziele als Vektoren gespeichert.
Im weiteren Verlauf, wird eine *Force Direction*, das heißt eine Richtung in der das Fahrzeug geschleudert werden soll ermittelt, indem von der Position des Benutzers die Position des Zieles abezogen wird.
Nun wird überprüft, ob sich dieses Ziel in einem Radius von x Längeneinheiten befindet.
Sollte dies zutreffen, so wird an den *Rigidbody* des Ziels eine Explosionskraft mit der integrierten Methode *AddExplosionForce* gehängt, welche das Ziel um x Längeneinheiten explosionsartig in die Höhe schießen lässt.
Zu guter Letzt, wird wieder an den *Rigidbody* des Ziels welches durch die Explosionskraft schon in die Luft gehoben wird, noch die *ForceDirection* mittels der Methode *AddForce* anmultipliziert, welches somit dafür sorgt, dass das Ziel von dem Benutzer weggeschleudert wird.
Theoretisch klingt das machbar und leicht nachdem man das alles weiß.
Praktisch allerdings hat jedes Fahrzeug eine andere Masse und verhält sich anders. Somit können wir nicht einfach die gleichen fixen Werte an jeder Fahrzeug anmultiplizieren, da dies entweder dafür sorgen würde dass sich am Ende Fahrzeuge mit einer sehr hohen Masse als immun erweisen, oder Fahrzeuge mit einer sehr geringen Masse Wort wörtlich einmal quer über die ganze Karte gesprengt werden.

Die Umsetzung

Nachdem die Probleme klar vor uns lagen war die Lösung logisch.
Wir mussten eine Abfrage einfügen welche herausfindet welchen Namen unser Ziel hat, und dann dementsprechend die *ForceDirection* mit individuellen Werten multiplizieren.
Das einzige Problem hierbei war, dass sich manche Autos anders verhielten als erwartet.
Durch ausprobieren, kamen wir schließlich auf absurde Werte, welche allerdings den gewünschten Effekt erzielten, das heißt, das Auto nicht zu weit weg schleudern, jedoch weit genug weg um für den Benutzenden einen anständigen Vorteil zu erzielen.
Die Individuellen Werte reichen hierbei von über 100.000 (Starwarsgleiter), bis unter -60.000 (Motorrad).
Für den Bereich in dem Ziele vom Impuls erfasst werden, entschieden wir uns auf 10 Längeneinheiten, und die höhe auf welche die Fahrzeuge geschleudert werden wurde auf 30 Längeneinheiten festgelegt. Nach mehrmaligem Ausprobieren erschien uns das als angemessen und fair.

Der fahrer des Batmobiels in der Mitte hat Impuls eingesetzt um seine beiden Kontrahenten wegzustoßen

Bei interesse findet sich der Programmcode für den Impuls hier:
https://github.com/tbourdon/CGPrak14-Unity/commit/ea13b4b91fede731be678c3047afddb71fb4cb59

12. Audio

Autoren: Benedikt Schumacher, Sargini Rasathurai

Dieses Kapitel widmet sich der akustischen Untermalung der erstellten Software. Im Folgenden wird darauf eingegangen, wie Fahrzeuge, Umgebung und Gegenstände mit Soundeffekten belegt werden.

12.1 Vorbereitung

Autoren: Benedikt Schumacher, Sargini Rasathurai

In der Vorbereitungsphase hat sich die Arbeitsgruppe mit dem Heraussuchen der Audiodateien befasst und diese gegebenenfalls nachträglich bearbeitet, was eine Endloswiedergabe ermöglichte, ohne den erneuten Beginn der Wiedergabe erkennen zu können. Hierfür erwies sich das Dateiformat *.ogg* als besonders tauglich.

12.2 Umgebungsgeräusche

Autoren: Benedikt Schumacher, Sargini Rasathurai

Um ein möglichst reales Fahrerlebnis zu bieten, bedarf es neben Motorengeräuschen und quietschenden Reifen auch Geräuschen aus der Umgebung. Fährt der Spieler durch einen Wald soll dementsprechend raschelndes Laub zu hören sein, begibt der Spieler sich ein eine Wüste soll ein seichter Wüstenwind wehen und gelangt der Spieler in die Nähe eines Gewässers, soll je nach Quelle eine leichter Wellengang oder ein Plätschern zu vernehmen sein. Weiter ist zu beachten, dass die Lautstärke der jeweiligen Töne mit zunehmender Entfernung von der Quelle abnimmt und endlich ausklingt.

Für die Realisierung von Umgebungsgeräuschen wurden in die jeweiligen Areale in Unity leere *GameObjects* erzeugt. Diese erhielten im Anschluss die für das Umfeld vorgesehenen Audiodateien als sog. *Audio Source Component*. In den Einstellungen mussten dann die Felder *Play on Awake* und *Loop* aktiviert werden damit die jeweiligen Effekte von Beginn an wiederkehrend abgespielt würden. Um die Geräusche nur in einem jeweiligen Umkreis um ein Gebiet hörbar zu halten wurde in den Einstellungen unter dem Menüpunkt *Volume Rolloff* der Punkt *Custom Rolloff* gewählt. Anschließend wurde eine sanfte Kurve von maximaler Lautstärke bei minimalem Abstand von der Audioquelle zu der Distanz, ab der nichts mehr zu vernehmen sein sollte, gezogen.

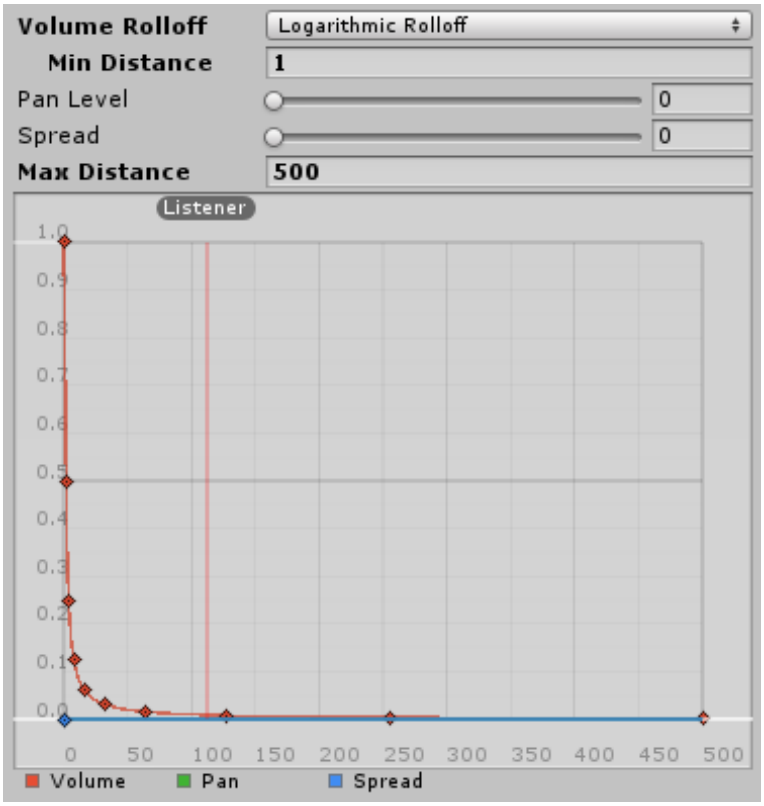
Fahrt durch eine Umgebung

12.3 Fahrzeugsounds

Autoren: Benedikt Schumacher, Sargini Rasathurai

Grundsätzlich enthält jedes Fahrzeug bei Projektabschluss drei Audiodateien; Eine für den Motorstart, eine für das eigentliche Motorengeräusch und eine für das bei rasanter Kurvenfahrt auftretende Reifenquietschen. Die Audiodateien sind jeweils den Fahrzeugen als sog. *Audio Source Component* angehängt. Eine gleichmäßige Geräuschkulisse wurde durch die vorgenommenen Einstellungen unter dem Menüpunkt *Pan Level*, welches

auf null gesetzt wurde, um zu vermeiden, dass der Ton bei Lenkbewegungen die Balance verlagert obgleich die Kameraposition fix am Fahrzeugheck ist, erzeugt. Der Punkt *Volume Rolloff* muss unverändert auf *Logarithmic Rolloff*, also der logarithmischen Lautstärkeabnahme, bleiben, damit im Multiplayer nicht auch sämtliche Fahrzeuggeräusche der Mitfahrer in konstanter Lautstärke unabhängig von ihrer Position zu hören sind. Durch den logarithmischen Lautstärkeabfall ist gewährleistet, dass die übrigen Autos nur hörbar sind wenn sie sich in der näheren Umgebung des Spielers befinden.



Logarithmische Lautstärkeabnahme des Motorengeräusches auf 500 LE

12.3.1 Das Starten

Autoren: Benedikt Schumacher, Sargini Rasathurai

Das Startgeräusch wird mittels Auswahl des Menüpunktes *Play on Awake* einmalig bei der Instanziierung des jeweiligen Fahrzeugs abgespielt. Der eigentliche Motorton wird durch die Auswahl der Option *Loop*, so im späteren Verlauf gestartet, permanent abgespielt. Gestartet wird der eigentliche Motorton durch einen Aufruf im Fahrzeugeigenen Skript. Das Reifengeräusch wird nur per Aufruf aus dem jeweiligen Fahrzeugskript abgespielt.

12.3.2 Das Schalten

Autoren: Benedikt Schumacher, Sargini Rasathurai

Da das Fahrgeräusch nach Möglichkeit realistisch gehalten werden sollte bedurfte es auch einem akustischen Schaltgefühl. Die Einbindung mehrerer Audiodateien für unterschiedliche Drehzahlen erwies sich schnell als untauglich, denn es folgten unvermeidbare abrupte Tonwechsel sobald ein neuer Drehzahlbereich erreicht worden ist. Sanfte Drehzahlbedingte Tonveränderungen waren hiermit nicht zu erreichen.

Als Lösung für dieses Problem erwies sich die Einstellungsmöglichkeit des Pitches, also der Tonhöhe. Zugrunde gelegt wurde im Folgenden eine einzige Audiodatei, die den Leerlaufton eines Motors wiederspiegelt. Die Tonhöhe lässt sich durch ein Skript verändern. Es wurde also das Fahrzeugskript dahingehend verändert, dass

nun in jedem *Frame* die Tonh%ouml;he des aktuellen Motorengeräuschs anhand der anliegenden Drehzahl neu berechnet wird. Jedes Fahrzeugskript enthält eine Update-Methode, die in jedem *Frame* ausgeführt wird. Aus dieser Methode heraus wird eine andere Methode zur Berechnung und Aktualisierung der Tonhöhe aufgerufen.

In der eigentlichen Motorgeräusch-Methode wird zunächst die anliegende Drehzahl an den Reifen abgefragt und in einer Variablen hinterlegt. Anschließend wird geprüft, ob das Fahrzeug nach vorn oder nach hinten fährt und eine boolsche Variable dementsprechend gesetzt. Wenn der Wagen von der Drehzahl her offensichtlich fährt und nicht nur rollt, wird eine Berechnung der eigentlichen Tonhöhe vorgenommen, andernfalls wird die festgelegte Tonhöhe für den Leerlauf ausgewählt. Fährt das Fahrzeug rückwärts wird auf eigentlich erfolgende Schaltvorgänge verzichtet.

Das eigentliche Schaltgeräusch durch den akustischen (vermeintlichen) Drehzahlabfall wird durch Modulo-Rechnung erzeugt. Nach einigen Versuchen erwies sich die Reifendrehzahl in 300er-Schriten als am besten geeignet um einen akustischen Schaltvorgang zu vollziehen. Da der *Pitchwert* in *Unity* nur im Wertebereich von -3 bis +3 einzustellen ist, wird das Ergebnis der bisherigen Berechnung durch 300 geteilt, mit dem Faktor zwei multipliziert und um eins aufaddiert. Dies hat zur Folge, dass der Drehzahlabfall nicht zu intensiv ist und somit unrealistisch wirkt. Am Ende der Methode wird die berechnete Tonhöhe, sollte sie nicht in den Werteberiech passen, korrigiert. Abschließend wird die neue Tonhöhe gesetzt. Da das eigentliche Geräusch in einer Endloswiedergabe ist kann auf einen erneuten Aufruf der Audiodatei verzichtet werden.

Fällt die Drehzahl bei diesem Verfahren der Tonhöhenermittlung ab, kommt es durch den verwendeten Algorithmus auch zu einem vernehmbaren Herunterschalten, da durch die Restwertberechnung der Wertebereich invers zum Hochschalten verläuft.

```
// Drehzalabhaengiges Motorengeraeusch wird erzeugt
void motorGeraeusch() {

        // Reifendrehzahl holen
        drehzahl = bl.rpm;

        // auf Rückwaertsgang prüfen
        bool back = false;

        if (drehzahl < 0) {
                drehzahl *= -1;
                back = true;
        }

        // auf Leerlauf prüfen
        if (drehzahl < 30) {
                // Leerlaufgeraeusch setzen
                sound.pitch = 0.6f;
                ersterGang = true;
        } else {
                // ggf 'schalten'
                if (drehzahl > 300 && !back) {
                        drehzahl = drehzahl % 300;
                        ersterGang = false;
                }

                float pitch = 0;

                // Fahren in hoeherem Gang
                if(!ersterGang) {
                        pitch = ((drehzahl / 300) * 2) + 1;
                }
                // Fahren im ersten Gang
                else {
                        pitch = ((drehzahl / 300) * 3);
                }

                // ggf Korrekturen
                if(pitch < 0.6f)
                        sound.pitch = 0.6f;
                if(pitch > 3f)
                        sound.pitch = 3f;
                else
                        sound.pitch = pitch;
```

```
        }
}
```

Beispiel des Motorsounds inkl. Reifenquietschen

12.3.3 Die Reifengeräusche

Autoren: Benedikt Schumacher, Sargini Rasathurai

Bei rasanten Kurvenfahrten beginnen bekanntlich die Räder eines jeden Automobils zu quietschen. Damit dieser Effekt auch in dem Spiel Einzug gehalten hat, musste das Fahrzeugskript um eine weitere Methode ergänzt werden. Gleich der Motorengeräusch-Methode wird auch Diese *Frame* für *Frame* aufgerufen. Auch diese Methode erfragt die anliegende Reifendrehzahl, aber auch den aktuellen Winkel, in dem die Reifen eingeschlagen sind. Zur weiteren Berechnung wird der Betrag des Winkels verwendet, da die Einschlagsrichtung im Gegensatz zur Intensität für die Berechnung irrelevant ist. Hat der Wagen nun eine gewisse Geschwindigkeit (Drehzahl) und überschreitet der Einschlagswinkel einen kritischen Wert, so wird aufgrund der Länge der zu spielenden Audiodatei alle drei Frames das dementsprechende Geräusch so lange wiedergegeben, bis die Kriterien dafür nicht mehr erfüllt sind.

```
// Methode um ggf Reifenquietschen abzuspielen
void reifenGeraeusch() {

        // Drehzahl und Einschlagwinkel holen
        drehzahl = bl.rpm;
        winkel = fl.steerAngle;

        // ggf Betrag von Winkel holen
        if (winkel < 0)
                winkel = winkel * -1;

        // Wenn Kriterien erfuellt sind, Sound abspielen
        if(drehzahl > 300 && winkel > 5)
                quietschen.Play ();

}
```

12.4 Kollisionsgeräusche

Autoren: Benedikt Schumacher, Sargini Rasathurai

Da die Vertonung eines Rennspiels mit Motoren- und Umgebungsgeräuschen ohne die Berücksichtigung von Kollisionen befremdlich wirkt, widmet sich dieses Kapitel der Einbindung von Soundeffekten die im Falle einer Berührung von Fahrzeugen mit Umgebungsgegenständen, wie z.B. Bäumen oder Steinen, abgespielt werden.

Generell unterscheidet sich die Einbindung solcher Soundeffekte nicht wesentlich von Der anderer Effekte zuvor. Wie auch bei den Motorengeräuschen und in der Umgebung benötigen alle Objekte in der Szene, die im Falle einer Kollision mit dem Spieler einen Ton erzeugen sollen, ein sog. *Audio Source Component* in welchem der Kollisionssound als Audiodatei hinterlegt ist.

Um zu realisieren, dass der jeweilige Ton ausschließlich an treffender Stelle abgespielt wird bedarf es ein simples Skript, was dafür sorge trägt, dass ein Objekt die Kollision mit dem Spieler erkennt und daraufhin den gewünschten Ton abspielt. Hierfür ist es von Wichtigkeit, dass das Objekt über einen *Collider* (Der *Collider* macht ein Objekt undurchquerbar) verfügt. Durch das beigefügte Skript besitzt das Objekt durch die Funktion *OnCollisionEnter* eine Arbeitsanweisung für den Fall einer Berührung. Kommt es zu einem Kontakt von Spieler und Umgebung wird also der Methodeninhalt ausgeführt, hier ausschließlich das Abspielen der hinterlegten Audiodatei mit dem Befehl *audio.Play()*.

```
// Im Falle einer Kollision Sound abspielen
void OnCollisionEnter() {
    audio.Play();
}
```

Beispiel einer Kollision ohne weitere Sounds

13. Android

Autoren: Benedikt Schumacher, Sargini Rasathurai

Dieses Kapitel befasst sich mit der Portierung des im Rahmen des Praktikums entwickelten Rennspiels auf Android. Im Wesentlichen wurden die Eigenschaften des Spiels aus der PC-Version beibehalten, es wurden lediglich einige Kürzungen vorgenommen, um eine gute Performanz auf den mobilen Geräten erzielen. Aus den ursprünglich existierenden drei Strecken wurde nur die mit minimalem Speicherverbrauch ausgewählt. Außerdem wurde auf das Feature des Multiplayer-Modus verzichtet. Weitere Anpassungen wurden bei der Steuerung des Fahrzeuges und im Auswahlmenü vorgenommen.

13.1 Entwicklungsumgebung

Autoren: Benedikt Schumacher, Sargini Rasathurai

Bevor man mit der Programmierung/Anpassung des Spiel für Android anfangen kann, muss man einige Einrichtungen vornehmen. Wichtig ist es, die Android SDK (http://developer.android.com/sdk/index.html) auf den Entwicklungsrechner zu laden, da man sonst die Applikation für Android nicht erstellen kann. Beim ersten Build muss der Pfad zum SDK-Ordner angegeben werden.
Unity Remote ist ein Tool zum parallelen Testen des Spiels auf dem mobilen Gerät. Es ermöglicht dem Benutzer, das Handy mit dem USB - Kabel an den Entwicklungsrechner anzuschließen und das Spiel direkt auf dem Handy zu testen, sodass Steuerungen, zum Beispiel das Kippen des Handys oder Berühren eines Buttons, übernommen und im Unity-Editor angezeigt werden. Dazu muss auf dem mobilen Gerät „UnityRemote“ installiert sein (https://play.google.com/store/apps/details?id=com.unity3d.genericremote).

Außerdem muss das USB-Debugging bei den Entwickler-Optionen auf dem Smartphone aktiviert sein.
Am Entwicklungsrechner muss bei Unity im Menü Edit-> Project Settings -> Editor die Erkennung der Androidgeräte aktiviert sein.

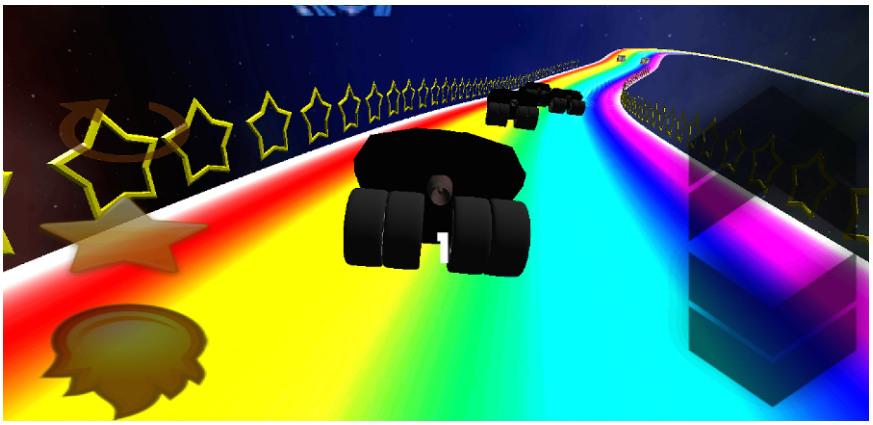
Der Code kann, wie bei der PC-Version auch, in JavaSript oder C# verfasst werden. Jedoch ist es nicht möglich, Dynamic Typing für Android Unity zu verwenden.

Die Bildschirmausrichtung kann ebenfalls eingestellt werden. Generell unterscheidet man zwischen Portrait („Hochformat“ mit dem Home-Button unten), *PortraitUpsideDown* („Hochformat“ mit dem Home-Button oben), *LandscapeLeft*(„Querformat“ mit dem Home-Button rechts) und *LandscapeRight*(„Querformat“ mit dem Home-Button links). Für die vorgestellte Version wurde das *LandscapeLeft* - Orientierung ausgewählt.

13.2 Spielszenario

Autoren: Benedikt Schumacher, Sargini Rasathurai

Das Spiel besteht aus einer einzelnen Strecke, in der der Spieler gegen computergesteuerte Autos antreten kann. Das Ziel des Spieles ist es, als erster und mit der kurzmöglichen Zeit die Runden zu fahren. Dabei besteht die Möglichkeit, den Boost zu verwenden, um die Geschwindigkeit zu erhöhen, und PowerUps einzusammeln und diese für die Spieltaktik zu benutzen. Die Anzahl der zufahrenden Runden kann beliebig eingestellt werden. Als Strecke für das Rennspiel wurde die „Rainbow Road“ ausgewählt, da diese am wenigsten Speicher verbraucht (nur 18 Mb, zum Vergleich: Wüstenstrecke ca. 150 Mb) und die beste Performanz auf dem Smartphone erzielt. Somit hat das Spiel einen Minispielcharakter und läuft flüssig auf allen Androidgeräten (ab Version 4.3). Zur Realisierung der computergesteuerten Autos wurde an allen Fahrzeugen das Skript der Gruppe „KI“ angehängt. Ebenfalls wurden die Kameras der mitfahrenden Autos entfernt, sodass es kein „Ruckeln“ während der Fahrt gibt.



13.3 Steuerung

Autoren: Benedikt Schumacher, Sargini Rasathurai

Realisierung für das Touchscreen

Zwecks der besseren Übersicht wurden nur folgende Symbole für die Steuerung/Aktivierung verschiedener Features ausgewählt:

Boost

Aktivierung des PowerUps

Respawn

Vorwärtsfahren

Rückwärtsfahren

Auf die Funktion „Vollbremsung“, die in der PC-Version enthalten ist, wurde hier verzichtet.
Die Steuerung des Fahrzeuges zum Vorwärts- und Rückwärtsfahren wurde durch Gedrückthalten des entsprechendes Buttons realisiert. Die Buttons sind vom Typ GUITexture. Sobald der Benutzer ein Button anklickt, erkennt das UnityGameEngine dieses Ereignis. Über die Methode Input.Touches können die „Touches“, also die Berührpunkte am Bildschirm, abgefragt werden. Androidgeräte (und iOS-Geräte) sind multitouching-fähig, sodass alle Berührpunkte (beim gleichzeitigen Drücken) mit dem Bildschirm erkannt werden und Input.Touches ein Array an „Touches“ zurückgibt.
Die Klasse Touch enthält Informationen wie Koordinaten des Punktes, der auf dem Bildschirm berührt wurde und die Phase, das heißt, ob man den Finger gedrückt hält (*touch.phase == TouchPhase.Stationary*) , loslässt (*touch.phase == TouchPhase.Ended*) oder gerade anfängt zu drücken (*touch.phase == TouchPhase.Began*).

Man iteriert über alle Berührpunkte, die erkannt wurden. Sobald die Koordinaten der Touchpoints mit den Koordinaten des jeweiligen Buttons, beispielsweise mit dem Vorwärts-Button, übereinstimmt und die Phase der TouchPhase.Stationary entspricht, wird der Methode die Methode userInput(), die auch in der PC-Variante zu finden ist, aufgerufen mit den Argumenten passend zum Vorwärtsfahren. Beim Vorwärts- und Rückwärtsfahren bewegt sich das Auto solange man den Finger an der entsprechenden Stelle gedrückt hält. Boost und das PowerUp werden beim Anfang der Drückphase aktiviert.

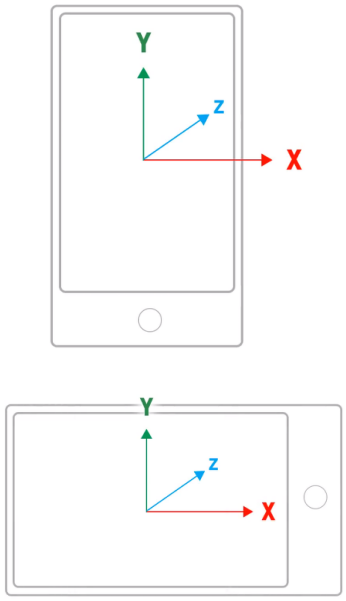
```
foreach (Touch touch in Input.touches) {

        if (touch.phase == TouchPhase.Stationary && boostButton.HitTest (touch.position)) {
                shift = true;
        }
        if (touch.phase == TouchPhase.Ended && boostButton.HitTest(touch.position)) {
                shift = false;
        }
        if (touch.phase == TouchPhase.Stationary && vorwaertsButton.HitTest (touch.position))
{
                up_down = 1;
        }
        if (touch.phase == TouchPhase.Stationary && rueckwaertsButton.HitTest (touch.position))
 {
                up_down = -1;
        }

}
```

Tilting

Das Lenken des Fahrzeuges wird durch das Neigen des Smartphones gesteuert.
Die drei Richtungen, in denen man das Handy bewegen kann, werden in den drei Hauptachsen aufgeteilt. Sowohl im Querformat als auch im Hochformat wird diese Aufteilung beibehalten (siehe Abbildung).



Der Beschleunigungsmesser der Androidgeräte gibt Daten zu linearen Beschleunigungsänderungen entlang der Hauptachsen weiter. Kippt man nun das Handy senkrecht haltend nach rechts oder links, wird, nicht wie anzunehmen, der Wert über die Z-Achse abgefragt, da die Rotation um die Z-Achse stattfindet, sondern über die X-Achse. Die Unity-Klasse Input bietet die Abfrage über diese Informationen über Input.acceleration, wobei der Rückgabewert vom Typ Vector3 ist.

```
if(Input.acceleration.x> 0)
```

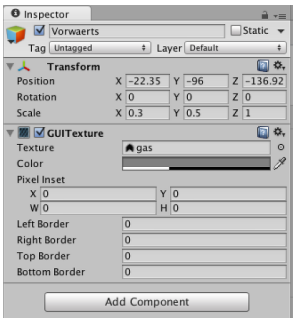
```
        keyInput(101, 1f, true);
if(Input.acceleration.x< 0)
        keyInput(101, -1f, true);
```

13.4 Weitere Besonderheiten

Autoren: Benedikt Schumacher, Sargini Rasathurai

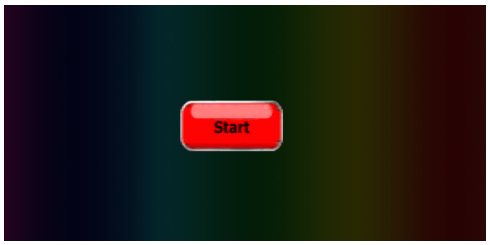
Anordnung der Symbole

Bei der Anordnung der GUITexture-Elemente sollte Folgendes beachtet werden, da sonst die Elemente abhängig von der Auflösung des mobilen Gerätes unterschiedliche Größen und Platzierungen haben: im Inspector (Unity) die Pixel-Inset-Werte alle auf null setzen und die Skalierung und Position im Menü Transform verändern.



Menü

Das Menü, das beim Öffnen der Applikation erscheint, besteht lediglich aus einem einzigen Button, einem Objekt der Klasse GUITexture. Sobald dieser berührt wird (Erkennung realisiert wie bei der Steuerung), wechselt die Szene zur Fahrbahn.



Countdown

Das Spiel beginnt mit einem Countdown, währenddessen die angehängten Skripte des Fahrzeuges deaktiviert sind, sodass man noch nicht fahren kann.
Es wird ein Objekt vom Typ GUIText in der Szene platziert. Sobald das Spiel startet, wird von drei runtergezählt. Um Sekundenpausen einzulegen, wird auf die Funktion WaitForSeconds() zurückgegriffen.

```
for(countDown = countMax ; countDown > 0 ; countDown--){

                guiCountDown.text = countDown.ToString();
                yield WaitForSeconds(1);
```

```
}
```

Nachdem der Countdown abgelaufen ist, werden wieder die Skripte aktiviert und das Objekt mit dem Text des Countdowns wird deaktiviert (enabled = false). Da die Funktion enabled für die Klasse Unity.Component nicht vorhanden ist, muss das Objekt zunächst in ein MonoBehaviour-Objekt umgewandelt und dann deaktiviert werden.

```
(this.GetComponent( "Car_Batmobil" ) as MonoBehaviour).enabled = true;
```

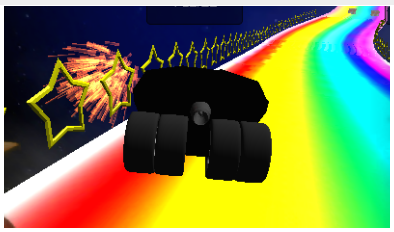
Vibration

Um mehr Leben ins Spiel zu bringen, wird bei Kollisionen des Fahrerautos mit anderen Objekten eine Vibration erzeugt. Dies geschieht mit dem Unity-Interface Handheld, das verschiedene Funktionalitäten für tragbare elektronische Handgeräte bietet. In diesem Fall wird die Funktion Handheld.Vibrate() aufgerufen, die die Vibration auslöst, falls das mobile Gerät Vibrationen unterstützt. Der Code wurde in der Methode OnCollisionEnter() eingefügt.

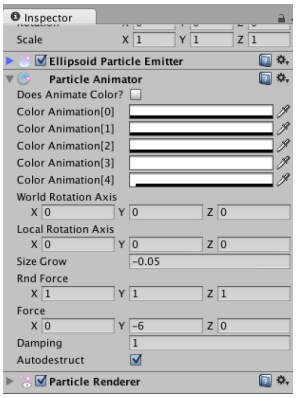
Sparks

Neben der Vibration werden bei Kollisionen Funkeneffekte erzeugt. Dem Skript „Car_Batmobil“ wurde eine Klassenvariable vom Typ GameObject hinzugefügt, dem das Prefab „Sparks“ aus den Standardassets zugeordnet wird. Sobald das Auto mit anderen Objekten kollidiert, werden Funken in den Kollisionspunkten erzeugt und danach zerstören sie sich selbst, sodass kein unnötiger Speicher für nicht angezeigte Funkenobjekte verloren geht. Die Methode Instantiate erzeugt Klone vom Orginal, also vom Spark.

```
if (other.transform != transform && other.contacts.Length != 0) {
        for(int i = 0; i < other.contacts.Length; i++){
                Instantiate(spark, other.contacts[i].point, Quaternion.identity);
        }
}
```



Die Selbstzerstörung muss manuell im Prefab Sparks (Inspector) aktiviert werden.



13.5 Kurzer Ausblick

Autoren: Benedikt Schumacher, Sargini Rasathurai

Einige interessante Erweiterungen, die das Spiel noch ansprechender gemacht hätten, konnten aus zeitlichen Gründen nicht vorgenommen werden.

Portierung für iOS

Bei der Portierung des Spiels für iOS-Geräte ist analog zur Portierung für Android-Geräte zu verfahren, da die Methoden zur Steuerung die gleichen sind. Jedoch braucht man zur Übertragung der Applikation auf ein Iphone ein Entwicklerkonto bei Apple, das kostenpflichtig ist.

Multiplayer

Ebenfalls wurde sowohl aus Zeit- als auch aus Kostengründen auf die Realisierung des Multiplayermodus für die mobile Version des Spiels verzichtet. Das Photon Plus - Paket bietet eine API zum Aufsetzen eines Netzwerkes für Android. Um nun ein Multiplayermodus zu implementieren, muss analog zur PC-Version, aber mit Rückgriff auf diese spezielle API vorgegangen werden.

13.6 Download

Autoren: Benedikt Schumacher, Sargini Rasathurai

Die Applikation kann unter dem folgenden Link heruntergeladen werden und ist spielbar ab Android-Version 4.3:
http://project.informatik.uni-osnabrueck.de/rennspiel/Rennspiel.apk

14. Webplayer

Autoren: Benedikt Schumacher, Sargini Rasathurai

Beim Builden des Spiels als Webplayer wird automatisch eine HTML-Seite generiert, in der das Spiel eingebettet ist. Selbstverständlich ist es möglich, diese Seite anzupassen. Der Unity-Inhalt wird durch das Unity-Plugin geladen. Die HTML - Inhalt kommuniziert über das Skript „Unity3“ mit dem PlugIn. Dieses Skript ist unter anderem dafür zuständig, den Benutzer zur Installation des PlugIns aufzufordern, falls dies nicht vorhanden ist. Es wird im Header der HTML geladen.
Schließlich wird ein UnityObejct2 initiiert und die Methode initPlugin aufgerufen. Der erste Parameter der Methode stellt die ID des HTML-Elements dar, welches durch den Unity-Inhalt ersetzt wird. Der zweite Parameter beinhaltet den Pfad zu der Unity3d-Datei.

```
var u = new UnityObject2(config);
u.initPlugin(jQuery("#unityPlayer")[0], â€ StreckeMitKi_Aktuell.unity3d");
```

Im Body der HTML-Datei kann nun die tatsächliche Gestaltung der Seite erfolgen. Dabei wird an die Stelle die Spielebox geladen.
Um den Inhalt im Browser anzeigen zu können, muss das Unity-Webplayer-Plugin installiert sein.

14.1 Link

Autoren: Benedikt Schumacher, Sargini Rasathurai

Hier geht es zum Webplayer:

http://project.informatik.uni-osnabrueck.de/rennspiel/

15. Resümee

Zusammenfassenden kann man sagen, das wir in diesen drei Wochen des Praktikums sehr viel erreicht und gelernt haben, sei es nun die Blender- oder die Unitygruppe.
Obwohl die erste Woche nahezu komplett der Einarbeitung gewidmet wurde, kannten wir uns danach doch recht gut mit den jeweiligen Programmen aus. Sehr geholfen hat uns dabei die, zwar etwas verspätet angekommene, aber große fachliche Kompetenz von Christoph Eichler. Die Anwesenheit von Timo Bourdon und Niels Meyering darf dabei natürlich auch nicht vergessen werden, sowie die Organisation von Prof. Vornberger.
Wir bedanken uns an dieser Stelle bei den oben genannten Personen und allen anderen, die dieses Praktikum ermöglicht haben.

Ausblick

Da wir uns wahrscheinlich an einigen Stellen wegen Unwissenheit oder zu tiefe Bearbeitung *(oder weil Unity nicht wollte)* aufgehalten haben, ist das Projekt etwas unfertig abgeschlossen worden. In der PC-Version fehlen daher die Einbindungen der anderen drei Karten sowie der Einzelspielermodus und noch einige andere Kleinigkeiten. Natürlich hätte man auch vorhandene Dinge verschöneren bzw. verbessern können, aber alles in allem kann man doch sehr zufrieden sein.