Dokumentation

Zeppelin Spiel

Andreas Rettig

Charles Bourasseau

Computer Graphics and Effects Sommersemester 2011

Prof. Dr. Henrik Tramberend

Beuth Hochschule für Technik Berlin

Inhaltsverzeichnis

Abbildungsverzeichnis

Einführung

Dieser Bericht ist die Dokumentation eines Projektes von Andreas Rettig und Charles Bourasseau. Die Arbeit wurde in Sommersemmester 2011 an der Beuth Hochschule für Technik in Berlin während des Masterseminars “Computer Graphics and Effects” von Prof. Dr. Henrik Tramberend erstellt. In diesem Bericht wird das Projekt vorgestellt, einige technische Aspekte erläutert und es werden Probleme und Schwierigkeiten aufgezeigt.

Aufgabe

Während des Seminars sollte ein Spielprototyp entwickelt werden. Für die Umsetzung des Projektes standen zwei Monate zur Verfügung, in denen die folgenden Meilensteine erreicht werden mussten:

- Ideenentwicklung
- Entwurf der Grundstruktur mit Spiellogik
- Optimierung des Looks mithilfe von Shadern

Die Arbeit konnte in Gruppen durchgeführt werden und technische Entscheidungen waren nicht vorgegeben. Zum Abschluss des Seminars musste ein Spieleprototyp, ein Video und eine Dokumentation des Spiels abgegeben werden.

Das Projekt wird mit dem folgenden Punkte bewertet:

Kriterium	Gewicht
Umsetzung der Idee	20%
Integration der Shadertechniken	25%
Technische Schwierigkeit	35%
Code-Qualität	10%
Dokumentation & Video	10%

Idee

Spielidee

Die erste Idee war, ein Flugzeugspiel zu programmieren. Dieser Spieltyp ist sehr bekannt und bietet für die Umsetzung der Spiellogik folgende Möglichkeiten:

- Simulation
- Wettrennen mit Checkpoints
- Angriff mit Waffen

Es wurde entschieden, dass das Spiel ein Rennen ist, in dem man durch Checkpoints fliegen muss. Das Spielprinzip ist relativ einfach. Durch das Hinzufügen von Wind und Regen kann der Schwierigkeitsgrad des Spiels erhöht und somit das Interesse des Spielers verlängert werden.

Als Alleinstellungsmerkmal wurden steuerbare Zeppeline gewählt, wodurch des Interesse des Spielers geweckt werden soll.

Abbildung 1 Zeppelin

Um ein realistisches Spielerlebnis zu erreichen, wurden alle Steuerungsmöglichkeiten des Zeppelins implementiert:

- Pitch: Rotation nach oben und nach unten
- Gas ablassen um nach oben zu fliegen
- Ballast abwerfen, um nach unten zu fliegen
- Bewegung vorwärts/rückwärts
- Kurven fliegen

Dabei sollte sich das Flugverhalten deutlich von der eines Flugzeuges unterscheiden. Ein Zeppelin schwebt alleine in der Luft, ist aber deutlich träger und langsamer als ein Flugzeug und wird durch Wind und Regen stärker beeinflusst.

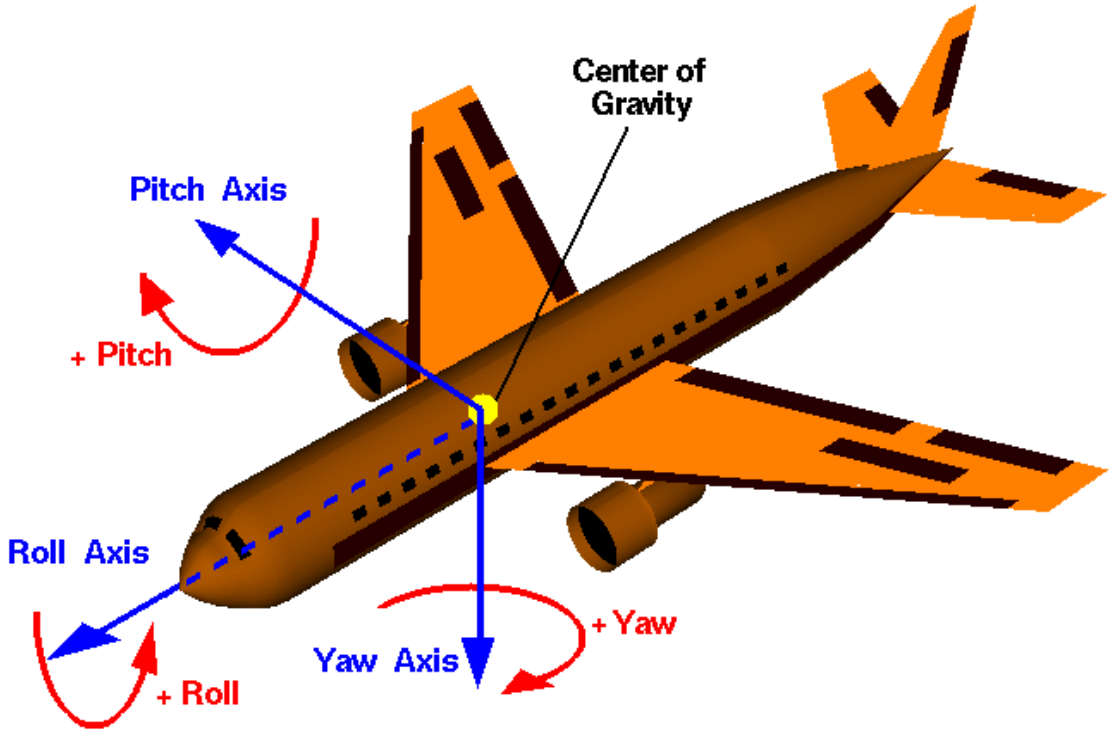

Abbildung 2 Bewegungsachsen am Beispiel eines Flugzeugs (http://commons.wikimedia.org/)

Dafür wurde folgende Tastaturbelegung definiert:

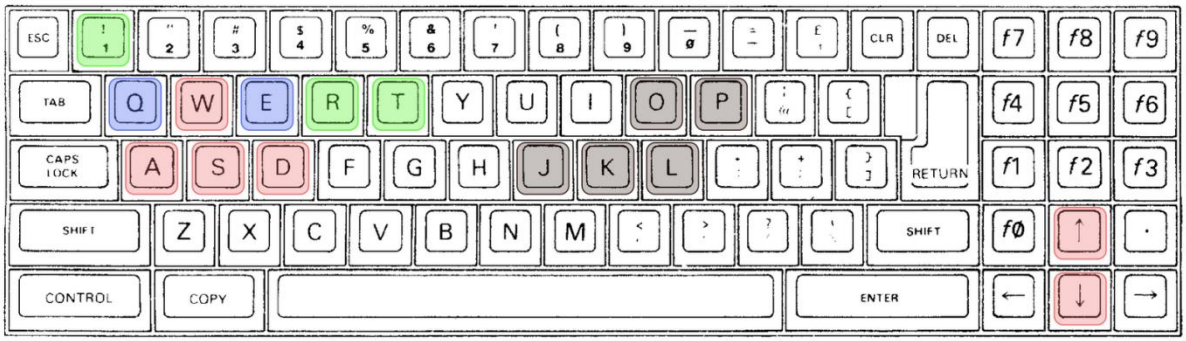

Abbildung 3 Tastaturbelegung

Der Tastatur ist in verschiedene Bereich geteilt:

- In Rot sind die klassischen Bewegungen organisiert:
  - W : Geradeaus fliegen
  - S : Zurück fliegen
  - A : Rotation nach links
  - D : Rotation nach rechts
  - ↑ : Rotation nach Oben
  - ↓ : Rotation nach Unten

- In Blau sind die Zeppelin-spezifischen Bewegungen und Spielaktionen organisiert:
  - Q : Ballast abwerfen (Zeppelin fliegt nach oben)
  - E : Gas ablassen (Zeppelin fliegt nach unten)
  - 1 : Ändert die Kameraposition

- In Grün sind die Steuerungen des Rennens organisiert:
  - R : Zurück am Anfang
  - T : Wetter aus- und anschalten

- In Grau sind die Steuerungen zum Debuggen und Testen organisiert:
  - O : Produziert Regen
  - P : Produziert Wind
  - L : Shader-Debug Mode (Shader werden nach jedem Frame neu kompiliert)
  - J : Produziert Regeneffekt
  - K : Neugenerierung des Terrains

Atmosphäre

Ursprünglich sollte das Spiel im Stil des “Steampunk” gestaltet werden. Leider hat die zur Verfügung stehende Zeit nicht mehr gereicht, um diesen Stil bis auf kleine Ansätze umzusetzen.

Steampunk

“Steampunk” gründet sich auf eine literarische Strömung der 1980er Jahre und hat sich seitdem zu einer kompletten Subkultur ausgeweitet. Der Stil des Steampunk verknüpft moderne Technologien mit Materialien und Mitteln des viktorianischen Zeitalters. Es orientiert sich an der Vision, wie unsere Welt und Technologien heute aussehen könnten, wenn sich vor allem die Mechanik und Dampfkraft durchgesetzt hätten.

Abbildung 4 Steampunk-Stil am Beispiel eines modernen Computers (http://commons.wikimedia.org/)

Technische Entscheidungen

Wie in der Einleitung beschrieben wurde, waren keine technischen Entscheidungen vorgegeben. Für die Umsetzung des Projekts konnten daher eine beliebige Sprache sowie ein beliebiges Framework gewählt werden. Es gibt viele verschiedene Gameengines, wie zum Beispiel Horde3D in C++ oder jMonkeyEngine und jReality in Java. Sie sind sehr gut dokumentiert und Informationen lassen sich schnell finden.

Im Projekt fiel die Wahl auf jVR, welche eine Rendering Engine auf Basis von OpenGL 3.1 ist. Diese Bibliothek wurde im Sommersemester 2010 von Marc Roßbach im Rahmen seiner Masterarbeit geschrieben. Viele Projekte des Seminars “Computer Graphics and Effects” aus dem letzten Semester wurden mit jVR umgesetzt.

Desweiteren wurde die Bibliothek Processing (http://processing.org/) verwendet, um Funktionen wie Perlin Noise und Mapping verwenden zu können. Dieses Projekt ist ein Framework, das es erlaubt graphische Animationen schnell zu erstellen. Es wurde in Jahre 2001 von Casey Reas und Benjamin Fry geschrieben und seitdem stark erweitert und optimiert.

Szenenarchitektur

Zeppelinmodell

Das Zeppelinmodell ist aus primitiven Objekten direkt im Code zusammengesetzt. Das Laden komplexer Modelle aus Collada-Dateien erwies sich als problematisch. Besonders die Übernahme von Normalen und Texturkoordinaten erwies sich als schwierig.
Das Modell besteht aus einer in die Länge gezogenen Kugel und drei Würfeln:
Ein Würfel stellt die Kabine dar, zwei Würfel die Leitwerke.

Abbildung 5 Modell des Zeppelins

Cockpit

Das Zeppelincockpit ist aus drei rechteckigen Flächen zusammengesetzt, auf denen zwei Würfel die Zeiger für Gas und Ballast darstellen, die den aktuellen Zustand des Gases und Ballastes anzeigen.

Flugphysik

Da gute Spielbarkeit eine höhere Priorität bei der Entwicklung hatte als ein möglichst naturgetreues Flugverhalten wurde auf eine physikalisch korrekte Umsetzung des Flugverhaltens verzichtet. Charakteristische Eigenschaften wie Trägheit und Zentrifugalkraft sollten allerdings auftreten.

Das Bewegungsmodell besteht im wesentlichen aus drei Geschwindigkeitskomponenten:

- Rotationsgeschwindigkeit um die Hoch- und Querachse
- Geschwindigkeit entlang der Längsachse des Luftschiffes.

Alle Geschwindigkeiten unterliegen einer Dämpfung, werden also ohne Einfluss des Spielers immer weniger bis das Luftschiff zum Stillstand kommt.

Die Dämpfung der Vorwärtsbewegung ist von der aktuellen Geschwindigkeit abhängig und begrenzt so die Geschwindigkeit auf ein gewisses Maximum.

Die Rotationsgeschwindigkeit um die Hochachse bewirkt ein Kippen des Zeppelins bis zu einem bestimmten Winkel, so dass sich das Luftschiff relativ realistisch in die Kurven legt.

Hinzu kommt noch die Schwerkraft, die sich aus Erdanziehung, Auftrieb durch Gas und das Gewicht des geladenen Ballastes ergibt.

Der Spieler beeinflusst also im wesentlichen die Geschwindigkeiten des Zeppelins, aus dem sich dann erst die konkrete Bewegung ergibt.

Hinzu kommen Wind und Regen, die sich zusätzlich auf die Bewegung auswirken: Wind treibt das Luftschiff zur Seite, während Regen es nach unten drückt.

Die Steighöhe des Luftschiffes wird begrenzt, damit der Spieler die Endlichkeit des Terrains weniger bemerkt und um die Bedienung zu vereinfachen.

Terrain

Das Luftschiff fliegt über einem zufällig erzeugten Terrain, das nur bis zu einer bestimmten Distanz erzeugt wird. Näheres zur Erzeugung im folgenden Kapitel.

Terrainerzeugung

Das Spiel findet über einem zufällig erzeugten, hügligen Gelände statt. Aus Performancegründen kann allerdings nicht das gesamte Gelände dargestellt werden, sondern immer nur der Bereich in einer bestimmten Entfernung um den Zeppelin. Dabei sollte sichergestellt werden, dass sich das Gelände z.B. beim Hin- und Rückflug nicht verändert - die verwendete Zufallsfunktion sich also nach der Initialisierung bei gleichen Eingangswerten deterministisch verhält. Für die Erzeugung der Geländehöhen wurde eine zweidimensionale Perlin-Noise-Funktion aus der Processing.org-Bibliothek gewählt. Nach der Initialisierung liefert diese für einen zweidimensionalen Eingangswert immer den gleichen Wert zurück. Weiterhin eignet sich Perlin-Noise gut zur Geländeerzeugung, da er kein rein zufälliges Rauschen liefert sondern weiche Rauschwolken mit sanften Übergängen.

Als Basis des Geländes wird zuerst eine gleichmäßige Dreiecksfläche aufgespannt. Die einzelnen Knoten dieser Fläche werden dann abhängig vom Wert des korrespondierenden Pixels des erzeugten Perlin-Noise in ihrer Höhe verändert.

Abbildung 6 Perlin-Noise

Große und kleine Erhebungen

Um eine optische Begrenzung des Geländes zu erreichen, steigt das Gelände zu den Rändern an. Dieser Geländeanstieg wird erreicht, indem eine zweite Perlin-Noise-Funktion mit deutlich niedrigerer Frequenz überlagert wird, die mit der Entfernung vom Geländemittelpunkt quadratisch an Einfluss gewinnt.

Wasser

Auf dem Gelände ist ein See vorhanden. Dessen Position wird bei der Geländeerzeugung zufällig festgelegt. Geländepunkte, die an einen bestimmten Abstand zur Seemitte unterschreiten werden, abhängig vom Abstand zur Mitte, unter das Null-Level abgesenkt. Ausserdem werden dort nur senkrechte Normalenvektoren erzeugt. Der Shader behandelt beim Rendern alle Punkte mit einer Geländehöhe unter Null gesondert und stellt sie als Wasser dar (siehe dazu entsprechendes Kapitel).

Dynamische Erzeugung

Das Gelände wird in einem festen Grundraster erzeugt. Es wird immer mittig unter dem Zeppelin mitgeführt, damit es auch bei schnellen Drehungen in jeder Richtung bereits gerendert ist. Überschreitet der Zeppelin einen Rasterpunkt, wird das Gelände neu erzeugt.

Normalenerzeugung

Die Erzeugung der Oberflächennormalen des Geländes findet mit einem vereinfachten Verfahren statt. Der Normalenvektor wird dabei durch die Höhendifferenz der vier nächsten, benachbarten Geländepunkte bestimmt. Dazu werden die Flächennormalen der beiden angrenzenden Flächen jeweils in X- und Z-Richtung ermittelt und addiert. Der resultierende Vektor wird normalisiert und erzeugt eine plausible Darstellung des Geländes.

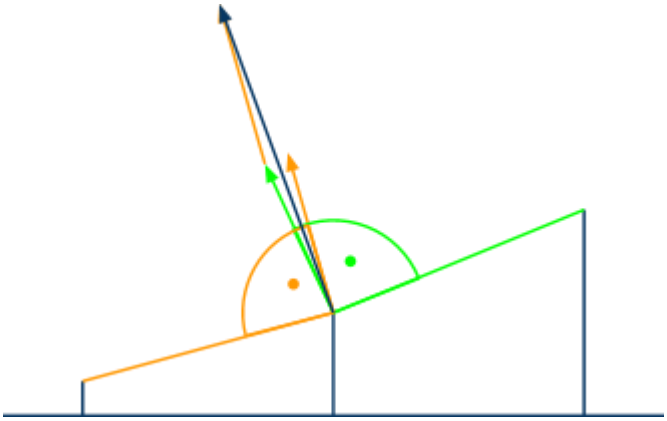

Abbildung 7 Ermittlung der Vertex-Normalen (Blau)

Debugging

Sehr nützlich zur Fehlersuche bei der Normalenberechnung hat sich erwiesen, die Normalenrichtung als Farbe darzustellen. Dazu werden die Normalen im Objektkoordinatensystem belassen und die Vektorkomponenten direkt in RGB-Farbe umgesetzt. Außerdem hat es sich als hilfreich erwiesen, das Gelände zur Fehlersuche z.B. mit einer einfachen Sinus-Funktion statt der Perlin-Noise-Funktion zu modellieren.

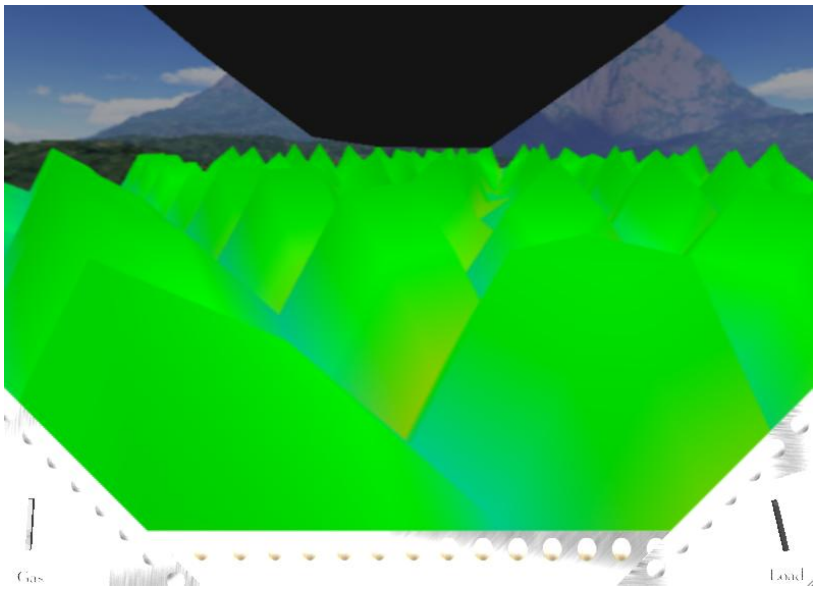

Abbildung 8 Gelände als Sinus mit Normalen-Shader

Kollision

Die Kollisionsabfrage mit dem Gelände wird benötigt, damit der Zeppelin nicht durch das Gelände hindurchfliegen kann.

Befindet sich das Luftschiff nicht genau über einem Geländeknoten muss trotzdem ein korrekter Höhenwert ermittelt werden.

Da die Perlin-Noise-Funktion, die der Geländeerzeugung zugrunde liegt, deutlich höher auflösend ist, als es für das Gelände benötigt wird, liegen zwischen zwei genutzten Werten auch höhere und tiefere Werte. Würde man für die Höhenabfrage eines beliebigen Geländepunktes also einfach den Wert der Perlin-Noise-Funktion an dieser Stelle nutzen, würde man unter Umständen in “Löcher fallen”, die man im Gelände aufgrund der geringeren Auflösung gar nicht sieht.

Es hat sich aber als ausreichend erwiesen, einfach das Maximum der Höhenwerte der umliegenden Geländeknoten als Geländehöhe für die Kollision zu nutzen. Das hat zwar zur Folge, dass man teilweise etwas über dem Gelände bleibt, stört im Spielverlauf aber nicht.

Shader

Blur

Eine künstliche Unschärfe (“Blur”) wird im Spiel zu zwei Zwecken eingesetzt:

- Eine leichte Tiefenunschärfe der gesamten Szene soll ihr eine natürlichere Anmutung geben.
- Die Partikelwolken der Checkpoints werden unscharf dargestellt, um ihnen eine wolkenähnlichere Anmutung zu verleihen.

Unschärfe ist ein klassischer Post-Processing-Shader-Effekt. Das bedeutet, man rendert das komplette Bild zuerst in eine Textur, auf der der Post-Processing-Shader dann die Effekte auf Pixelebene anwendet.

Im vorliegenden Fall müssen dabei allerdings die Partikel der Checkpoints anders behandelt werden als der Rest des Bildes. Dazu wird der Tiefenpuffer der Partikel in eine weitere Textur geschrieben und dem Post-Processing-Shader übergeben. Dieser kann dann anhand des Tiefenpuffers unterscheiden, ob der Partikel an dieser Stelle sichtbar ist und den Effekt dementsprechend anpassen. Die Tiefenunschärfe wird anschließend auf das ganze Bild angewandt.

Die Berechnung der Unschärfe erfolgt für beide Fälle identisch: Für jedes Pixel im Bild werden auch die benachbarten Pixel in die Berechnung der resultierenden Pixelfarbe einbezogen. Die Stärke des Effektes wird bestimmt durch die maximale Entfernung der einbezogenen Pixel.

Im Falle der Partikelunschärfe passiert das mit einem konstanten Wert, im Falle der Tiefenunschärfe ist die Stärke des Effektes vom Wert des Tiefenpuffers an der betrachteten Stelle abhängig.

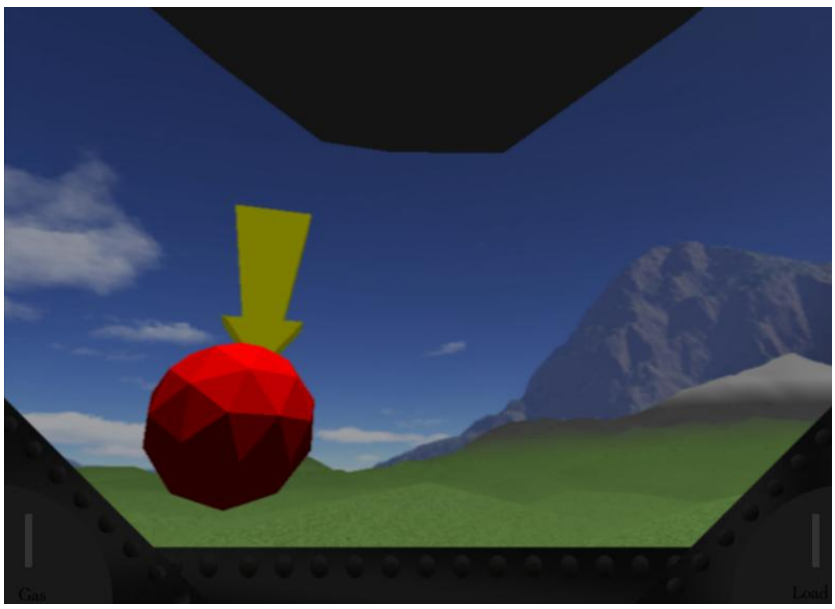

Abbildung 9 Bilder mit und ohne Blur

Partikelsysteme

Partikelsysteme sind in der Spieleindustrie eine sehr verbreitete Technik um sehr komplexe Phänomene nachzustellen. Sie ermöglichen zum Beispiel ohne großen Aufwand Wolken, Feuer, Rauch und Explosionen darzustellen, da der größte Teil der Berechnung auf die Grafikkarte ausgelagert wird.

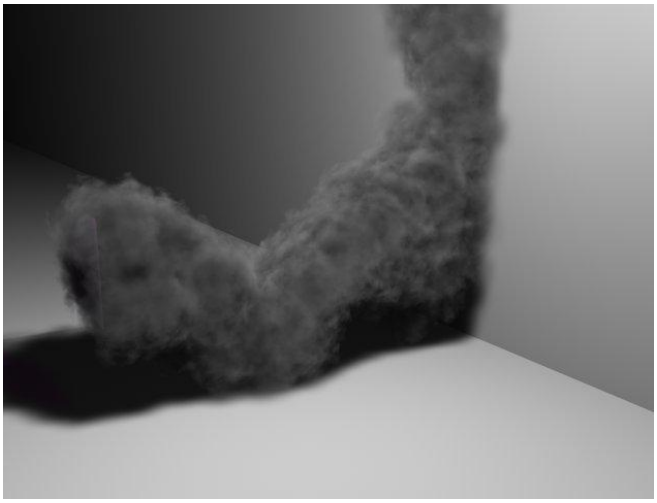

Abbildung 10 Beispiel eines Partikelsystems (http://commons.wikimedia.org/)

Im Projekt wurde dieses Verfahren auch umgesetzt, indem die Checkpoints als eine Art Wolke mithilfe eines Partikelsystems und einem Blur-Effekt umgesetzt werden. Die Partikel sind einfache Rechtecke, die alle in eine Richtung fliegen und so einen Kreis formen. Die Simulation aller Partikel wird in Java beschrieben und mithilfe eines Geometry Shader werden neue Geometrien für die Partikel erzeugt.

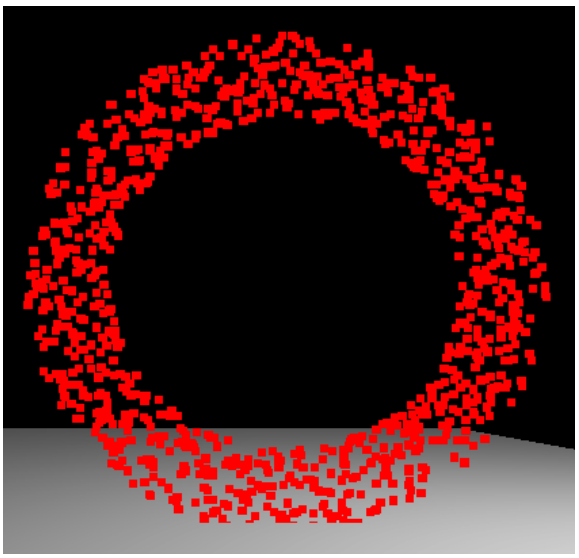

Abbildung 11 Checkpoint ohne Blur Effekt

Dazu kommt ein Blur-Shader, um die Rechtecke unscharf darzustellen und den Wolkeneffekt zu verbessern.

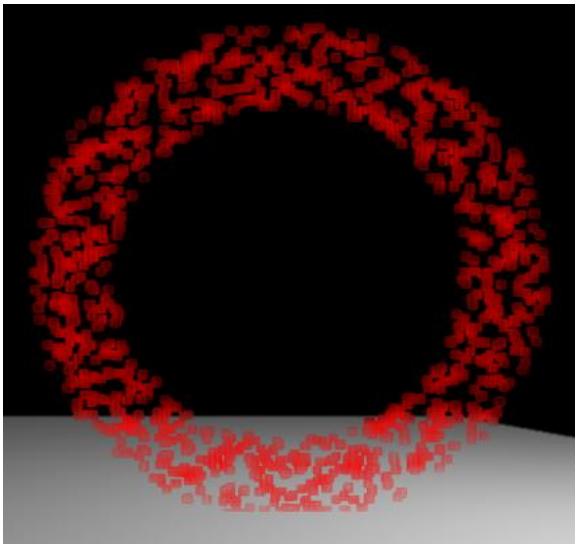

Abbildung 12 Checkpoint mit Blur-Effekt

Regen

Wie zuvor in der Beschreibung der Spielidee beschrieben wurde, sind Regen und Wind im Spiel, welche den Zeppelin bewegen. Um dies visuell darzustellen, wurde mit Hilfe von Partikelsystemen ein Regeneffekt implementiert. Hier sind die Partikel einfache graue Rechtecke mit Transparenz, die von oben nach unten fallen.

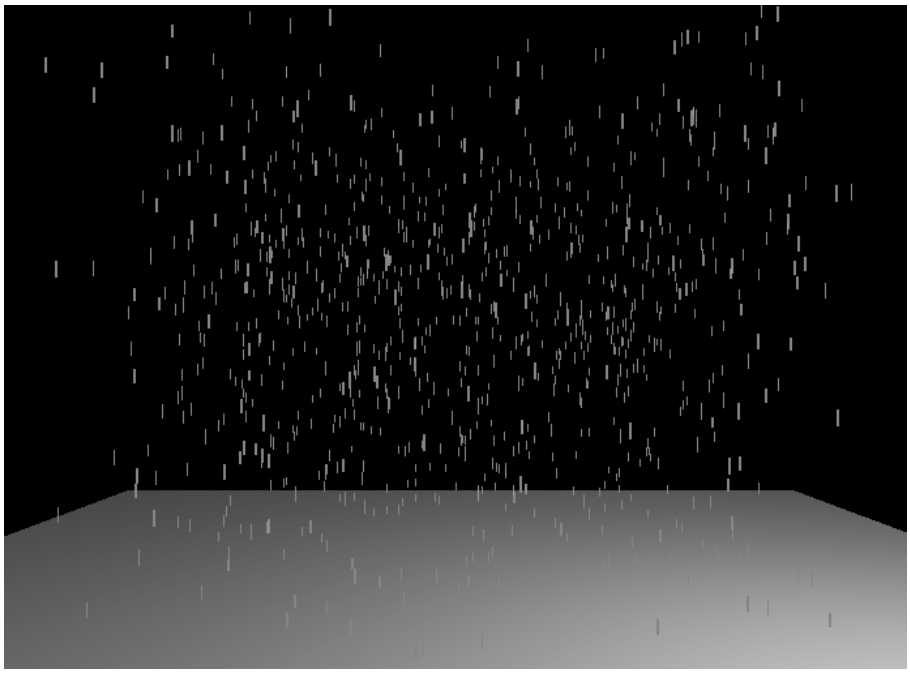

Abbildung 13 Regen Prototyp

Aus Performancegründen wurde entschieden, die Regenpartikel nicht überall darzustellen. Wie bei der Skybox befindet sich der Zeppelin in einer Regenbox, in welcher die Regenpartikel angezeigt werden. Dies ist in der dritten Kamera sehr gut sichtbar. In der nächsten Abbildung ist zu sehen, dass es nicht überall regnet.

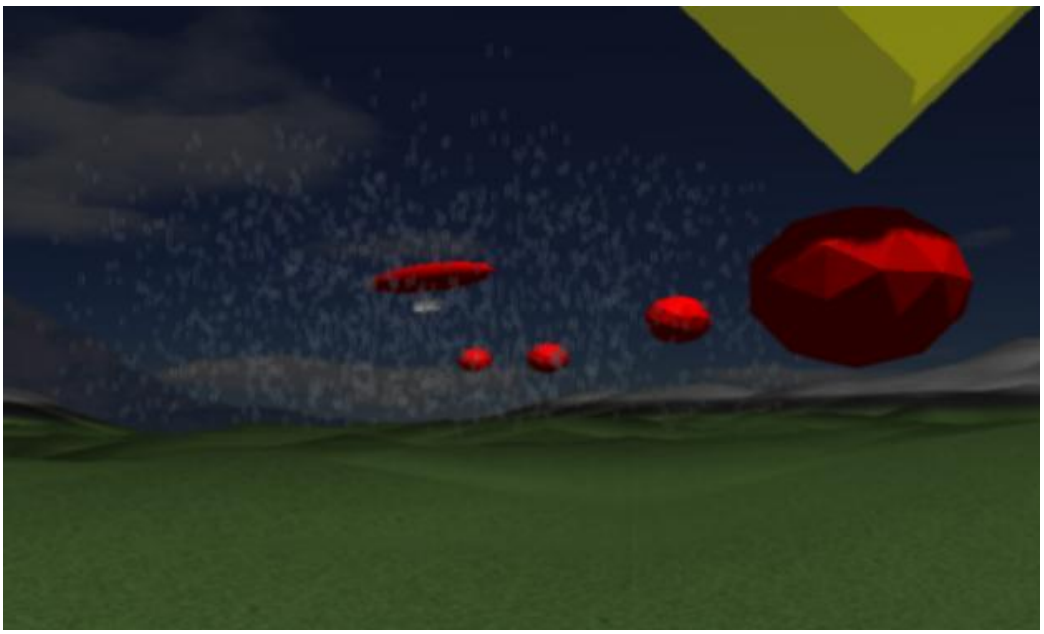

Abbildung 14 Regenbox

Befindet sich der Spieler jedoch im Spiel, ist dies aus Sicht der ersten und zweiten Kamera nicht sichtbar.

Abbildung 15 Regen aus der Sicht der externen Kamera

Terrain

Das Terrain nutzt einen speziellen Fragment-Shader, der die Textur des Geländes abhängig von Geländehöhe und Geländeneigung variiert.

Dazu werden die Normale und die Koordinaten des aktuellen Vertizes vom Vertexshader zusätzlich in Objektkoordinaten an den Fragment-Shader übergeben, um die Höhe und Neigung des Geländepunktes bestimmen zu können.

Der Fragment-Shader kennt fünf Zonen für die Texturierung:

- Low, Middle und High: Hier wird die passende Textur gewählt
- Übergänge von Low zu Middle und von Middle zu High: Hier wird zwischen den angrenzenden Texturen weich übergeblendet.

Die X- und Z-Komponenten des Normalenvektors beeinflussen ausserdem die Höhe der Grenze zwischen den Bereichen um eine natürlichere Anmutung zu erreichen. Ohne diese Variation sind die Übergänge immer auf einer Höhe und es entsteht der Eindruck von “Flutmarken”. Zusätzlich werden Punkte mit einer Höhe kleiner als Null als Wasser behandelt: Der Specular-Anteil wird berücksichtigt, um eine glänzende Oberfläche darzustellen, außerdem wird dort das Gelände blau eingefärbt. In Kombination mit den senkrecht stehenden Normalen aus der Geländeerzeugung ergibt das einen guten Eindruck einer transparenten Wasseroberfläche, ohne dass neue Geometrie erzeugt wird.

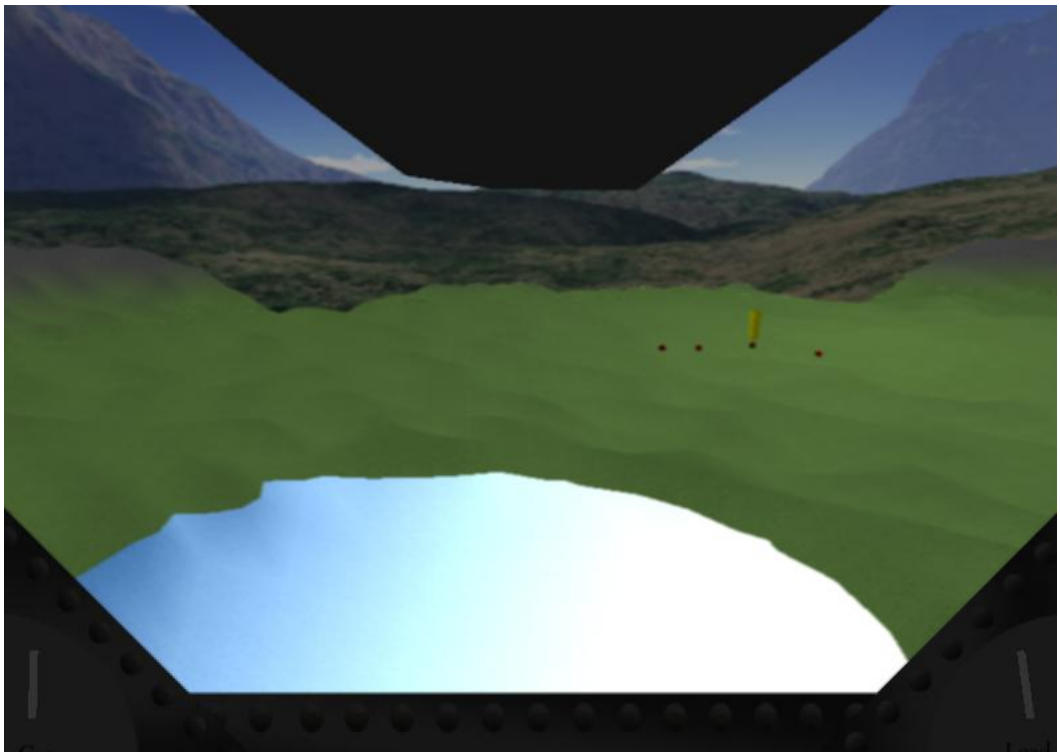

Abbildung 16 Glänzende Wasseroberfläche

Probleme

Geometry-Shader

Wie zuvor im Bericht erläutert wurde, sind Geometry-Shader sehr nützlich, um Partikelsysteme zu kreieren. Sie erlauben, Geometrie in der Renderingpipeline zu erzeugen, wodurch diese nur durch die GPU behandelt werden und somit kein Mehraufwand für die CPU entsteht.

Leider unterstützen nicht alle Grafikkarten dieses Verfahren. Das heißt, dass dieses Projekt auf manchen Computern nicht lauffähig ist, da die Geometry Shader nicht interpretiert werden können und in den meisten Fällen ein Fehler auftreten wird.

Um dieses Problem zu lösen, gibt es verschiedene Methoden. Die einfachste Möglichkeit wäre, keine Geometry-Shader zu nutzen, was die Kompatibilität zu älteren Grafikkarten verbessern würde. Dies würde aber bedeuten, dass für jeden Partikel ein 3D-Objekt erzeugt werden müsste. Theoretisch würde es funktionieren, jedoch wäre dies ein enormer Mehraufwand, wodurch die Performance des Projekts kein flüssiges Spielen zulassen würde.

Die zweite Möglichkeit wäre zu prüfen, ob die Grafikkarte Geometry-Shader unterstützt. Bei vorhandener Unterstützung müsste das Spiel nicht angepasst werden und das Partikelsystem würde gerendert werden. Für Grafikkarten ohne Geometry-Shader Unterstützung könnte man ähnliche 3D Modelle nutzen, welche nicht so gut aussehen aber trotzdem die Szene korrekt darstellen. Da es sich hier jedoch nur um einen Prototypen handelt, wurde diese Lösung nicht umgesetzt.

Terrain

Die Erzeugung des Geländes erwies sich als komplex. Besonders die Normalenberechnung hat in der Praxis einige Schwierigkeiten bereitet. Man verliert schnell den Überblick, welche Vertizes benachbart liegen und in die Berechnung einbezogen werden müssen. Ausserdem hat sich gezeigt, dass die große Vertex-Anzahl des Terrains trotz leistungsfähiger GPUs irgendwann problematisch wird.

Bei steigender Vertexzahl merkt man irgendwann eine kurze Verzögerung bei der Neuberechnung des Geländes und beim Rendering.

Hier ist noch viel Potential für Optimierung:

- Die Tesselierung des Geländes könnte abhängig von der Entfernung zum Betrachter angepasst werden, so dass nur in Kameranähe mit hoher Tesselierung gearbeitet wird.
- Eine feine Tesselierung könnte per Geometry-Shader auf der Grafikkarte erfolgen.

- Es müssten nicht bei jeder Geländeneuberechnung sämtliche Knoten neu erstellt werden. Es wäre ausreichend, eine einzige Reihe hinzuzufügen und auf der anderen Seite des Geländes eine Reihe zu verwerfen.
- Feine Strukturen des Geländes lassen sich vermutlich auch sehr gut per Bump-Mapping abbilden (Felsoberflächen u.a.) ohne die Komplexität des Geländes selbst zu erhöhen.

Modellimport

Der Import von komplexeren Modellen aus Drittsoftware hat sich leider als recht problematisch dargestellt.
Bei Modelle, die aus Google Sketchup exportiert wurden, tauchten immer wieder Probleme mit vertauschten Achsen und scheinbar auch invertierten Normalen auf. Um Texturkoordinaten zu erstellen müssen Modelle in Sketchup texturiert werden. Der jVR-Loader konnte allerdings mit den Pfaden zu den Texturdateien nicht umgehen. Um die so exportierten Modelle sinnvoll in jVR zu verwenden wäre viel Handarbeit nötig gewesen.

Der Versuch, die Modelle in Meshlab zu bearbeiten scheiterte leider am Export ins Collada-Format, bei dem das Programm reproduzierbar abstürzte und eine leere Zieldatei hinterließ.
Für Maya 2012 ist bisher scheinbar kein freies Collada-Import und Export-Plugin verfügbar. Das OpenCollada 2011-Plugin funktioniert leider nicht mit Maya 2012.
Aus Zeitgründen wurde entschieden, die Modelle auf die bereits vorhandenen primitiven Formen “Sphere”, “Box” und “Plane” zu beschränken und die Kompromisse bei der Ausgestaltung in Kauf zu nehmen.

Performance

Die Performance kann als gut bezeichnet werden. Obwohl an vielen Stellen noch sehr großes Optimierungspotential besteht, läuft das Spiel auf durchschnittlicher Hardware angenehm flüssig. Auf einem MacBook Pro 2.4Ghz mit NVIDIA GeForce 320M läuft es flüssig, d.h. mit Frameraten größer als 30fps.

Zusammenfassung

Das Projekt konnte erfolgreich bis zu einem Prototypenstatus gebracht werden, der die Spielidee vermitteln kann. Das spezielle Flugverhalten eines Zeppelins unterscheidet das Spiel von den üblichen Flugsimulatoren.

Die spezielle Atmosphäre, die das Spiel vermitteln sollte, ist leider aus Zeitgründen zu kurz gekommen.
Trotzdem konnten etliche Effekte mit eigenen Shadern umgesetzt werden, die mit einer herkömmlichen Fixed-Function-Pipeline kaum möglich sein würden.

Shader haben sich als ein mächtiges Werkzeug herausgestellt, eigene Vorstellungen von der Darstellung einer 3D-Umgebung sehr effektiv umzusetzen.

Die Shaderprogrammierung in GLSL setzt die Einstiegshürde dabei angenehm niedrig und macht Lust darauf, tiefer in die Materie einzusteigen.

Ausblick

Bei einer Weiterentwicklung des Spieles sollten etliche Punkte verbessert, ergänzt oder verfeinert werden:

- Anzeige des Spielstandes, Zeitmessung
- Verschiedene Schwierigkeitsgrade
- Verfeinerung der Modelle
- Gestaltung im Stil des “Steampunk”, um dem Spiel einen besonderen Charakter zu verleihen.
- Motivationssteigerung, z.B. durch Mehrspielermodus oder Highscore.
- Einbindung eines speziellen Gamecontrollers, um die komplexe Steuerung besser beherrschen zu können.