Hoverbike Madness

Team:

Johannes Unterguggenberger, 0721639, Kennzahl 033532, johannes.unterguggenberger@gmail.com
Christian Möllinger, 0725979, Kennzahl 033534, ch.moellinger@gmail.com

Grundlegendes Gameplay und Steuerung

Bei Hoverbike Madness handelt es sich um ein Multiplayer Hoverbike Rennspiel. Das Spiel wird im Splitscreenmodus gestartet, sodass gleich zwei Spieler gegeneinander fahren können.

Steuerung
Die Steuerung der Hoverbikes erfolgt per Tastatur.
Spieler 1:
Tasten WASD (W … beschleunigen, S... bremsen, A … links lenken, D … rechts lenken)
PowerUp auslösen: Space oder LCTRL

Spieler 2:
Cursor-Tasten; (Oben … beschleunigen, Unten... bremsen, Links … links lenken, Rechts … rechts lenken
PowerUp auslösen: Enter oder NumPad-Enter

Andere Tasten:
F2 → Debugging-Informationen anzeigen (Framerate, Settings)
F3 → Wireframe ein/aus
F4 → Texturfilterung umschalten (nearest/linear)
F5 →MipMapping umschalten (aus/nearest/linear)
F6 →VBO vs. VertexArrays vs. Immediate-Mode
F7 → DisplayLists ein/aus (hat nur im Immediate-Mode wirkung)
F8 → FrustumCulling ein/aus
F9 → Transparenz ein/aus (beeinflusst PartikelEngine, Checkpoints, PowerUps und die Wasseroberfläche bei deaktiviertem Shader)
F10 → QuadTree-Boxen anzeigen (geht nur bei aktiviertem FrustumCulling)
F11 → Shader ein/aus
F12 → Bloom bzw. PostProcess-Shader ein/aus (geht nur bei aktiviertem Shader)
1 → (Cheat) beiden Playern das „Invert“-PowerUp
2 → (Cheat) beiden Playern das „Wirble“-PowerUp
3 → (Cheat) beiden Playern das „Boost“-PowerUp
4 → (Cheat) beiden Playern das „Jump“-PowerUp
5 → (Cheat) beiden Playern das „InvertSteer“-PowerUp
6 → (Cheat) beiden Playern das „SidekickRight“-PowerUp
7 → (Cheat) beiden Playern das „Sidekick-Left“-PowerUp
8 → (Cheat) beiden Playern das „EngineFail“-PowerUp
9 → (Cheat beiden Playern das „Steal“-PowerUp

Gewonnen hat der Spieler, der als erster ins Ziel kommt. Wenn ein Spieler ins Ziel kommt, wird ein entsprechender Text mit der Platzierung des Spielers angezeigt.
Es gibt keine Streckenbegrenzungen, aber die Checkpoints müssen in der richtigen Reihenfolge abgefahren werden.

Bewegte Objekte

Die Hoverbikes sind bewegte Objekte. Ihre aktuelle Position wird stets neu berechnet, um den Abstand zum Terrain zu berechnen und ob sie sich der Spieler bereits im Ziel befindet. Das Terrain ist statisch und wird am Anfang geladen.

Heightmap (umfasst die Punkte texturierte Objekte, einfache Beleuchtung)

Das Gelände (Terrain) wird mithilfe einer Heightmap dargestellt. Dazu wird aus einer Bitmap-Datei der rote Farbkanal ausgelesen (um Platz für zusätzliche Informationen wie z.B. Streckenbegrenzung in einem anderen Farbkanal zu lassen). Aus den daraus gewonnenen Daten wird das Gelände in Blöcke unterteilt, wobei für jeden Block eine Displaylist erstellt wird. Jede Displaylist enthält einen oder mehrere glDrawElements(…)-Befehle, die das entsprechende Gelände entweder aus einem VBO oder aus einem Vertexarray rendern.
Die Blöcke werden in einer Quadtree-Struktur gespeichert, wobei jede „Node“ 4 Kinder hat und 2 DisplayLists, einmal eine DisplayList um die gesamte Node aus dem VBO zu rendern, und eine DisplayList um die Node aus einem VertexArray zu rendern. Bei jedem Rendervorgang wird nun, ausgehend von einer Root-Node die Heightmap gerendert. Dabei wird mit FrustumCulling festgestellt ob eine Node ganz sichtbar ist (dann wird sofort die DisplayList der Node gezeichnet), teilweise sichtbar ist (alle Child-Nodes werden aufgerufen und wie die aktuelle Node rekursiv überprüft) oder gar nicht sichtbar ist (gar nichts wird gezeichnet).
Im Immediate-Mode wird die komplette Heightmap entweder als DisplayList oder direkt mit glBegin(…) gerendert.
Die Lichtquelle ist lokal und ist einfach irgendwo in der Map platziert, sowie ambient und diffus, wobei die diffuse Lichtquelle heller ist wie das ambiente Licht. Bei aktiviertem Shader wird die Beleuchtung vom VertexShader übernommen, der die FixedPipeline einfach nachahmt.
Um die grafische Qualität des Geländes zu heben wird das Gelände mit einer Detailtextur, die die Struktur des Bodens darstellen soll, texturiert. Um Unterschiede wie „Sand“ (in der Nähe von Wasser), „Gras“ (normale Umgebung) und „Schnee“ (an hohen Plätzen des Geländes) darzustellen, wird einfach die Farbe eines jeden Vertex anhand seiner Höhe berechnet.
Bei aktiviertem Shader wird die Farbe nun im Shader berechnet, wodurch die Übergänge von Sand/Gras oder Gras/Schnee nun schön gerade sind, da sie „per-pixel“ berechnet werden, nur nicht „per-vertex“.
Diese Grenzen sowie die Toleranz(für einen fließenden Übergang) können ebenso wie die Wasserhöhe, die Skalierung der Höhenwerte, die Startposition der Spieler, die Position von Objekten (momentan nur testweise <tree> implementiert, die als rot-blau überkreuzende Quads dargestellt werden) über eine Levelkonfigurationsdatei (XML) gesteuert werden.

Wasser:

Das Wasser wird ohne Shader einfach durch eine transparente blaue Oberfläche dargestellt.
Bei aktiviertem Shader wird hingegen sowohl die Spiegelung (Reflection) als auch das Gelände unter der Wasseroberfläche (Refraction) jeweils in eine Textur gerendert und dann im Shader miteinander vermischt, wobei die Wasseroberfläche auch verzerrt wird und die Spiegelung bei weit entfernten Punkten mehr Gewicht hat wie bei nahen Punkten (in der Nähe hat die Refraction mehr Gewicht).
Es wird auch das Hoverbike korrekt gespiegelt bzw. dessen Schatten gezeichnet.

Schatten:

Die Schatten der Hoverbikes werden mit ShadowMapping erzeugt. Dazu wird zuerst die ShadowMap gerendert, die Kamera befindet sich hierbei ziemlich genau ÜBER dem Bike. Beim Zeichnen der Heightmap wird nun mit einem Shader verglichen ob die Punkte im Schatten liegen – der verwendete Shader hierfür entstammt großteils aus dem Yellow-Book, Listing 13.7, und wurde um den Code zur Farbberechnung (siehe Punkt Heightmap) ergänzt.

Bloom-Shader:

Nachdem die Szene gezeichnet wurde wird der Framebuffer in eine Textur kopiert und für diese Textur ein Bloom-Shader aufgerufen, um ein Glühen der hellen Stellen zu erzeugen. Abgesehen davon befinden sich in diesem Shader auch die Effekte die durch PowerUps vom gegnerischen Spieler ausgelöst werden können, wie die Verwirbelung und Invertierung der Szene.

Transparenz

Transparenz-Effekte wurden an passenden Stellen eingesetzt – so werden etwa die PowerUps oder die Checkpoints transparent dargestellt. Auch beim Wasser kommt Transparenz vor.

Partikel

Es kommt ein Partikelsystem zum Einsatz, welches für einen optischen Effekt bei den Hoverdüsen verwendet wird. Die maximale Partikelanzahl pro Düse beträgt 40. Insgesamt werden 4 Düsen dargestellt (2 pro Spieler). Die Partikeltextur ist ein TGA-Bild mit Alpha-Kanal, welches einen Punkt darstellt der nach außen hin immer transparenter wird. Mittels Billboarding werden diese Flächen stets nach der Kamera ausgerichtet.

Kameramodell, frei bewegbare Kamera

Bei unserem Splitscreen Spiel haben wir zwei Kameras. Eine Kamera verfolgt stets einen Spieler und zeigt diesen von hinten oben. Sie befindet sich dabei jedoch nicht in einem fixen Abstand hinter dem Spieler, sondern folgt der Spielerposition leicht verzögert in einer smoothen Bewegung. Das hat u.A. den Effekt, dass die Kamera beim Beschleunigen etwas zurückfällt und beim Bremsen näher an das Hoverbike heranfährt.
Die Kamera wird bei uns durch die Klasse Camera repräsentiert.

Datenstrukturen

Das Datenformat für Mesh-Dateien ist OgreXML. Zum laden der OgreXML Meshes kommt ein selbstgeschriebener Modelloader zum Einsatz (class Modelsmanager).

Dieser Modelsmanager erzeugt Klassen des Typs Model, welche die Daten eines 3D-Modells beinhalten. Die Vertices werden in einem eindimensionalen Array gespeichert, in dem sowohl x-, y- als auch z-Koordinaten der Vertices nacheinander gespeichert werden, also folgender Aufbau:
Vertex 0, Vertex 1, Vertex 2 der Reihe nach im Array: | x0 | y0 | z0 | x1 | y1 | z1 | x2 | …
Wir haben uns für diesen Aufbau entschieden, da das Array dann ohne Umformungen als Vertexarray der OpenGL Methode glDrawElements(...) übergeben werden kann.

Hardwareunabhängige Spielgeschwindigkeit

Damit das Spielerlebnis auf unterschiedlichen Rechnern gleich ausfällt, haben wir das Spiel an die Rendergeschwindigkeit angepasst. Dazu wird die Zeit gemessen, die das vorige Bild zur Anzeige benötigt hat. Das übernimmt bei uns die Klasse GTime. Alle Bewegungen des Hoverbikes (Beschleunigung, Lenken) werden mit dem so errechneten Faktor multipliziert, sodass das Hoverbike auf allen Rechnern gleich schnell fährt.

Surround Sound

Für die Soundausgabe kommt OpenAL zum Einsatz. Somit bietet Hoverbike Madness richtigen Surround-Sound. Die Positionen der Hoverbikes im Spiel fließen also direkt in die Soundpositionsberechnung ein und werden relativ zur Hörerposition gesetzt. Da es sich bei unserem Spiel um ein Splitscreen-Spiel handelt, haben wir stets zwei Hörerpositionen – nämlich die des linken und die des rechten Spielers. Der Sound wird für beide Positionen berechnet und entsprechend ausgegeben. Das hat natürlich den Effekt, dass wenn Spieler 1 an Spieler 2 rechts vorbeifährt, man sowohl links die Motorengeräusche von Spieler 2 hört, als auch rechts den Motor von Spieler 1, welcher ja dort an Spieler 2 vorbeifährt, welcher die zweite Hörerposition darstellt. Der Surround-Sound ist, eine entsprechende Soundanlage vorausgesetzt, sehr gut wahrnehmbar.

3D Modelle

Die 3D Modelle für PowerUps und das Hoverbike wurden selbst erstellt. Dabei kam Blender zum Einsatz. Ebenso die Texturierung des Hoverbikes erfolgte mittels Blender. Mittels OgreMeshesExporter wurden die Modelle exportert – nämlich im Format OgreXML – und mittels eines selbst geschriebenen Loaders wieder eingelesen und ins Spiel integriert.

Physik

Die Physik wurde selbst implementiert. Bei einem Hoverbike werden im Prinzip folgende Dinge simuliert, welche in die Bewegung des Hoverbikes einfließen:
Gravitation
Eine Hover-Düse unten am Bike. Diese stößt das Bike vom Untergrund ab. Je näher das Bike dem Boden kommt, desto stärker wird es abgestoßen, um es in der Schwebe zu halten.
Die Antriebsdüse, welche das Bike in Richtung des Front-Vektors bewegt. Wenn das Bike also etwas nach hinten kippt, zeigt der Frontvektor nach oben. Durch Beschleunigung aus dieser Position beschleunigt das Bike nach schräg oben.

Kollisionen

Die Kollisionserkennung wurde selbst implementiert. Kollisionen können z.B. auftreten mit dem Terrain, mit dem Gegenspieler oder mit PowerUps, wobei letztere im Falle einer Kollision eingesammelt werden.

Es kommen hierbei hauptsächlich BoundingSpheres zum Einsatz. Auch die Checkpoints oder das Ziel sind Kugelförmig und es wird geprüft, ob die BoundingSphere des Players mit dem jeweils nächsten Checkpoint kollidiert.

Steuerung und Spielinhalt

Es folgt nun eine noch etwas genauere Beschreibung der Steuerung und der Spielelemente.
Ziel eines Spielers ist es, als erster das Ziel zu erreichen und seinen Gegenspieler zu schlagen.

Steuerung des Bikes
Um das Hoverbike möglichst geschickt durch den Kurs zu manövrieren, ist es wichtig, folgende Dinge bei der Steuerung zu beachten:

Prinzipiell bewegt sich das Hoverbike träge. D.h. nur durch lenken (ohne dabei zu Beschleunigen) ändert das Bike seine Richtung nicht. Man muss also Energie in die Richtung, in die das Bike zeigt, zuführen indem man beschleunigt.

Je stärker ein Bike gegen das Terrain stößt (z.B. gegen eine Wand), desto mehr wird es abgebremst. Daher ist es wichtig, im richtigen Moment abzubremsen bzw. auszuweichen.
Durch drücken der Rückwärts-Taste ([S] beim linken Spieler und [Pfeil nach unten] beim rechten Spieler) bremst das Bike leicht, jedoch hat diese Aktion noch zwei weitere wichtige Effekte: Das Bike neigt die Schnauze nach oben und die Hoverdüse wirkt etwas stärker.
Das bedeutet, dass man so einen harten Aufprall etwas besser abfedern kann. Durch Beschleunigen eines nach oben geneigten Bikes, kann man etwas Höhe gewinnen und so eventuell leichter über einen Hügel kommen.
Wichtig ist jedoch zu beachten, daß während des Bremsens der Antrieb nicht wirkt. D.h. man kann nicht gleichzeitig das Bike nach oben kippen und beschleunigen.

PowerUps
Wichtiger Bestandteil des Spiels sind die PowerUps, welche über das Level verteilt sind.
Es gibt sowohl PowerUps, die dem Spieler, der sie gesammelt hat nützen – etwa einen Geschwindigkeitsboost verleihen oder einen Jump ausführen.
Weiters gibt es noch PowerUps, die dem Gegenspieler schaden. Sie sind erkennbar durch die Stacheln. Dazu zählen PowerUps, die die Physik des Gegenspielers beeinflussen (etwa seitlicher Stoß oder Antriebsausfall) als auch PowerUps, die die Sicht des Gegners verzerren oder es gibt auch ein PowerUp, welches dem Gegner ein PowerUp stiehlt, falls dieser gerade eines besitzt.

Manchmal sind die PowerUps an schwer zugänglichen Stellen, sodass man abwägen muss, ob es mehr bringt, das PowerUp zu holen, oder lieber direkt auf der Strecke zu bleiben.

Viel Spaß!

Zusätzliche Libraries

GLFW

GLEW

FreeImage
Zum Laden der Texturen und der HeightMaps wird FreeImage verwendet.

TinyXml http://www.grinninglizard.com/tinyxml
Der Modellsmanager verwendet TinyXml um die OgreXML Meshes zu laden.
Ebenso zum Laden der Leveleinstellungen (Startkoordinaten, Zielkoordinaten, Pfad zur Heightmap, etc.), welche bei uns in XML Dateien gespeichert sind, wird TinyXml verwendet.

OpenAL
für die Soundausgabe