

Dokumentation Abgabe 3

Stream-Racer

Steiner Bernhard 532 0825391 e0825391@student.tuwien.ac.at

Mazza Sebastian 532 0825828 e0825828@student.tuwien.ac.at

Implementierung

Für unsere Anwendung wurde ein eigenes Modellformat definiert, für das wir auch einen Python-Exporter für Blender entwickelt haben. Die Modelldaten wie Vertices oder Normalen werden in einer Plaintext-Datei gespeichert, welche im Spiel mittels einer selbst geschriebenen Modellloader-Klasse wieder eingelesen wird. Alle grafikspezifischen Daten werden mit VBOs/VAOs direkt im Speicher der Grafikkarte abgelegt; es wurde aber auch ein Modus geschaffen, in dem direkt aus den Arrays gezeichnet wird. Alle Modelle können texturiert werden, ebenso ist es möglich, Normalmaps zu verwenden. Die Daten werden in einer Baumstruktur gespeichert, die als Wurzel ein Model-Objekt hat, welches einer Scene aus Blender entspricht. Darunter werden ModelPart-Objekte angeordnet, die selbst wieder ModelParts enthalten können. Diese ModelParts definieren eine Kombination aus Material und Geometriedaten. Animierte Objekte wie die Schubdüsen des Raumschiffs oder die Planeten werden direkt im Programmcode animiert.

Die Daten über Levels und Highscores werden in einer Textdatei verwaltet. Es können somit sehr einfache neue Levels dem Spiel hinzugefügt werden. Der Highscore wird levelabhängig gespeichert, wobei jeweils nur die schnellsten Einträge angezeigt werden.

Alle Modelle mit Ausnahme des Streams übernehmen ihre Materialdaten bzw. ihre Geometriedaten aus Blender. Im Modellloader werden zusätzlich noch Tangenten berechnet, um Normalmapping im Tangentenraum durchführen zu können. Die Modelle werden mit dem Beleuchtungsmodell von Phong pixelgenau schattiert und von Punktlichtquellen beleuchtet. Die Lichtquellen werden im Stream-File definiert. Es werden immer die 5 Lichtquellen zur Helligkeitsberechnung herangezogen welche am nächsten zum Benutzer stehen. Die Skybox ist nur texturiert und wird nicht dynamisch beleuchtet.

Der Stream wird beim Starten der Anwendung aus den Daten des Streamfiles, die Angaben über die Position der Kontrollpunkte und Ausdehnung des Tunnels an den Kontrollpunkten enthalten, berechnet. Hierbei werden Ellipsen erstellt, zwischen denen dann Linien gezogen werden. Alle Effekte die auf den Stream angewandt werden (Lichter die durch den Stream laufen) werden direkt im Fragment-Schader berechnet. Ebenso wie die Lichtquellen werden auch die Positionen von Planeten oder sonstigen Objekten im Raum im Stream-File angegeben.

Für die Kollisionsberechnung wird die Bullet-Physik-Engine verwendet, wobei das Raumschiff und die Planeten jeweils durch eine Bounding-Sphere repräsentiert werden. Für den Tunnel wurde direkt das Drahtgittermodell verwendet.

Textausgaben am Bildschirm sind mit orthografischer Projektion implementiert, wobei die Grafik für die Buchstaben direkt aus einer TTF-Font mit Hilfe der FreeType Library geladen wird. Die Menüs werden ebenso orthografisch projiziert, Textausgaben wie im Highscore

werden durch die Textausgabefunktionen dargestellt, für die anderen Buttons wurden Texturen vorberechnet.

Steuerung und Kamera

Die gesamte Steuerung des Raumschiffs wird über Bullet realisiert, hierzu wird mit Planes an Anfang und Ende jedes Tunnels festgestellt, ob das Raumschiff im Tunnel ist oder nicht, und je nachdem reagiert. Befindet sich das Raumschiff im Tunnel, wird es automatisch beschleunigt. Die Kamera hängt leicht nach oben versetzt hinter dem Raumschiff, sie bleibt jedoch bei höheren Geschwindigkeiten weiter zurück, um den Eindruck von Geschwindigkeit nochmals zu verstärken.

Zur Verbesserung der Darstellungsgeschwindigkeit wurde auch View-Frustum-Culling, durch die AABBs die Bullet berechnet, implementiert.

Sound

Um Sounds wiederzugeben, wird OpenAL verwendet. Es ist sowohl Hintergrundmusik in den Menüs als auch im Spiel vorhanden, ebenso gibt es eine geschwindigkeitsabhängige Wiedergabe von Düsengeräuschen für das Raumschiff. Auch Kollisionen bzw. die Zerstörung des Schiffs sind mit Soundeffekten hinterlegt.

Effekte

Folgende Effekte wurden in dem Spiel implementiert:

- **Normalmapping**
Die Normalmaps werden mit Photoshop oder einer anderen geeigneten Software entworfen und in Blender zum Objekt hinzugefügt. Ebenfalls mittels Blender werden uv-Koordinaten bestimmt. Wenn ein Model geladen wird, werden die Tangenten und Bitangenten aus diesen Informationen errechnet. Diese werden in einem VBO für jedes Vertex abgelegt. Wenn ein Vertex gezeichnet werden soll wird im Vertex-Shader aus Tangente, Bitangente und Normale eine Transformationsmatrix berechnet, welche es ermöglicht, alle Informationen in den Tangentspace zu transformieren. Für jedes Pixel wird dann die Information aus der Normalmap gelesen und anhand der Normale die Helligkeit berechnet.
Quellen:
http://gpwiki.org/index.php/OpenGL:Tutorials:GLSL_Bump_Mapping
<http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson8.php>
- **Partikelsysteme**
für die Düsen des Raumschiffs. In jedem Frame wird eine von der Geschwindigkeit des Raumschiffs abhängige Anzahl an Vertices generiert, wobei der Startpunkt zufällig zwischen Turbinen Endposition des letzten und des aktuellen Frames gewählt wird. Weiter erhält jedes Partikel einen Geschwindigkeitsvektor und eine Lebensdauer. Nach Ablauf seiner Lebensdauer wird ein Partikel im Vertex-Array wieder durch ein neues Partikel überschrieben. Partikel werden einfach als Punkte durch OpenGL gezeichnet.
Quellen:
Randi J. Rost, Bill Liceas: OpenGL Shading Language (Orange Book), Third Edition, Kapitel 16.7 – Particle Systems, Seite 469

- **Motion Blur**

wurde als reiner Post-Processing Effekt umgesetzt. Aufgrund der UV und Z Koordinaten wird berechnet wo sich jeder Pixel im 3D-Viewing-Space befindet. Mithilfe der inversen der aktuellen Abbildungsmatrix wird die Position im World-Space berechnet. Die so erhaltene Position wird jetzt noch mit der Abbildungsmatrix des letzten Frames multipliziert. Wenn man die daraus resultierenden X und Y Koordinaten mit den aktuellen UV Koordinaten vergleicht (nach entsprechendem mapping von $[0, 1] \Rightarrow [-1, +1]$) erhält man einen Vektor welcher angibt in welche Richtung sich das Pixel im fertigen Bild seit dem letzten Frame bewegt hat. Entlang dieses Vektors werden nun einige Texel gelesen und der Durchschnitt gebildet. Dieser Durchschnitt wird die neue Farbe des Pixels.

Um das Raumschiff und dessen Schild immer scharf zu halten verfügt der Haupt-Frame-Buffer über einen zweiten Color-Buffer der als boolescher Speicher zweckentfremdet wird und angibt ob ein Pixel vom Motion Blur Fragment-Shader bearbeitet werden darf oder nicht.

Quellen:
 Hubert Nguyen: GPU Gems 3, Kapitel 27 – Motion Blur as a Post-Processing Effect, Seite 575 (http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html)
- **HDR**

Die Render-Pipeline des Spiels rendert in ein FBO welches als Color Buffer eine float Textur benützt, wodurch RGB-Ergebnisse des Fragment-Schaders welche größer als 1.0 sind nicht abgeschnitten werden, sondern korrekt gespeichert werden.

Quellen:
<http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/OpenGL/hdr.zip>
- **Bloom:**

Um dem Tonemapper die Möglichkeit zu geben, Fragments die so hell sind das sie nicht mehr korrekt gemapped werden können, überstrahlen zu lassen wird eine weichgezeichnete Textur der Szene benötigt. Theoretisch wird dazu auf das fertige Bild der Szene ein Gauß-Filter mit normalverteilendem Kern angewendet. Die ist aber in der Praxis aufgrund des benötigten Rechenaufwandes nicht praktikabel. Um ein großräumig weichgezeichnetes Bild zu erhalten, müsste man bei einer Auflösung von 1024×768 Pixel einen circa 40×40 großen Gauß-Kern benutzen. Dies würde bedeuten, dass für jedes Pixel der Szene 40^2 Texel Zugriffe notwendig wären. Um eine gute Approximation zu erhalten, deren Berechnung in für Computerspiele ausreichender Geschwindigkeit zu erhalten, werden insgesamt 6 FBOs benutzt. Jeweils zwei mit einer Auflösung von 128×128 , 64×64 und 32×32 Pixeln. Zunächst wird die fertig geränderte Szene in jeweils ein FBO jeder Auflösung kopiert. Durch das Verkleinern des Szenenbildes wird bereits ein gewisses Weichzeichnen errechnet, da die Textur beim kopieren linear gefiltert wird. Anschliesend werden die Framebufferinhalte jeder Auflösung in den zweiten, noch leeren Frame-Buffer mit gleicher Auflösung vertikal geblurt (nur in eine Dimension des Gauskerns berechnet). Das Bluren in horizontaler Richtung wird erst in einem zweiten Render-Schritt erledigt, wobei wieder am ersten FB jeder Auflösung geschrieben wird. Das Aufteilen des Filterings in zwei Schritte verkürzt die Laufzeit von einer quadratischen auf eine lineare. Für jede der drei Auflösungen wird ein 7×7 Gauskern nachgeahmt. Durch das zusammenführen (onoe, one blenden) dieser drei Texturen wird ein sehr gute Approximation eines großen Gauß-Kerns erreicht. Alle beschriebenen Filtervorgänge werden durch einen Fragmentschader erledigt.

Quellen:

http://www.gamasutra.com/features/20040526/james_03.shtml

<http://harkal.sylphis3d.com/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/>

- Tonemapping

Um den hohen Tonwertumfang aus der HDR-Textur welche die Szene enthält auf einem Bildschirm ausgeben zu können, muss jeder Farbkanal eines Pixels auf den Bereich zwischen 0 und 1 (bzw. 0 und 255) gemapped werden. Dazu wird zunächst der mittlere Grauwert des aktuellen Frames benötigt. Um nicht alle Pixel eines Frames aufsummieren zu müssen wird hierfür einfach die kleinste Mipmap (1x1 Pixel) der Color-Buffer-Textur für die Berechnung herangezogen. Um eine adaptive Lichtanpassung simulieren zu können wird über eine Zeitkonstante ein Wert zwischen dem mittleren Grauwert des aktuellen und des letzten Frames berechnet. Mithilfe dieses errechneten mittleren Grauwerts des gesamten Bildes wird für jeden Pixel auf Grund dessen Helligkeit ein Kompressionsfaktor berechnet, mit dem der Farbvektor des Pixels multipliziert wird. Selbiges wird auch für jedes Pixel in der weichgezeichnete Szenen-Textur gemacht, allerdings wird hier vor dem Tonemapping ein konstanter Threshold vom Farbvektor abgezogen, damit die geblurte Textur nur an den Stellen sichtbar ist in deren Umgebung sich sehr helle Pixel befinden.

Quellen:

<http://www.xnainfo.com/content.php?content=28>

Anhang A: Zusätzliche Libraries

- GLEW für OpenGL Extensions
<http://glew.sourceforge.net/>
- DevIL zum Laden von Bildern/Texturen
<http://openil.sourceforge.net/>
- FreeType zum Laden von Fonts
<http://www.freetype.org/>
- Zlib, wird für DevIL benötigt
<http://www.zlib.net/>
- GLFW für Matrix/Vektor Operationen
<http://www.glfw.org>
- Bullet für Physik
<http://bulletphysics.org>
- OpenAL für Sound
<http://connect.creativelabs.com/openal>

Anhang B: Tastenbelegung

Spiel

Taste	Funktion
W	Raumschiff beschleunigen
A	Raumschiff bremsen
S	Nach links rollen
D	Nach rechts rollen
PFEIL_LINKS	Nach links drehen
PFEIL_RECHTS	Nach rechts drehen
PFEIL_AUF	Nach oben kippen
PFEIL_AB	Nach unten kippen
ESC	Pausenmenü
STRG+ALT+ESC	Spiel sofort beenden

Menüs

Taste	Funktion
PFEIL_AUF	Einen Menüpunkt nach oben
PFEIL_AB	Einen Menüpunkt nach unten
PFEIL_LINKS	Nächstes Level, nur im Highscoremenü
PFEIL_RECHTS	Vorheriges Level, nur im Highscoremenü
ENTER	Menüpunkt wählen
ESC	Ein Menü nach oben / Zurück
STRG+ALT+ESC	Spiel sofort beenden

Debugausgaben

Achtung: Alle Ausgaben werden nur angezeigt wenn Debugausgaben aktiviert sind [F7].

Taste	Funktion
F1	Hilfe anzeigen
F2	Framerate anzeigen
F3	Wireframemodus ändern [EIN / AUS]
F4	Texturemodus ändern [LINEAR / SCENE NEAREST / ALL NEAREST]
F5	Mipmapmodus ändern [LINEAR / NEAREST / OFF]
F6	Zeichenmodus ändern [VBO / Arrays]
F7	Anzeige von Debugausgaben aktivieren
F8	Viewfrustrumculling [EIN / AUS]
F9	AlphaTransparenz [EIN / AUS]
F10	Motion Blur [EIN / AUS]
F11	HDR [EIN / AUS]
STRG + ALT + F1	Physikdarstellung aus
STRG + ALT + F2	Physikdarstellung Kollisionspunkte
STRG + ALT + F3	Physikdarstellung Wireframe
STRG + ALT + F4	Physikdarstellung AABB
STRG + ALT + F5	Kontrolllinie des Tunnels anzeigen
STRG + ALT + F6	Gauss-Kern anzeigen (HDR)
G + NUM_PLUS / G + NUM_MINUS	Glow-Threshold erhöhen / verringern
M + NUM_PLUS / M + NUM_MINUS	Mittlerer Grauwert erhöhen / verringern
L + NUM_PLUS / L + NUM_MINUS	Maximale Helligkeit erhöhen / verringern
NUM_2	Kamera nach unten
NUM_4	Kamera nach links
NUM_5	Kamera zurücksetzen
NUM_6	Kamera rechts
NUM_8	Kamera nach oben