Martin Kinkelin, #0326997, Studienkennzahl 534
Christian Liensberger, #0107660, Studienkennzahl 932

Edelweiss

Wir haben eine Szene in einem kuriosen Museum modelliert. In der Mitte eines abgeschlossenen Raumes steht ein Dinosaurier (Raptor). Dieser wird von vier Anubisstatuen, welche jeweils auf einem Steinsockel sitzen, angebetet. Außerdem finden sich in der Szene noch zwei Drachen, die entgegengesetzt angeordnet sind.

Die Kamera ist nicht interaktiv; sie fährt auf einer festen Bahn durch die Szene. Dabei werden die einzelnen Effekte an den einzelnen Objekten demonstriert (die Kamera fährt zu Stellen, an denen der jeweilige Effekt gut sichtbar ist und das Programm blendet die Bezeichnung des betrachteten Effekts ein).

Folgende Effekte haben wir in der finalen Fassung implementiert:

Bloom

Zur Simulation des HDR (High Dynamic Range)-renderings wurde ein Bloom-Effekt implementiert. Dieser Effekt hebt Flächen, die direkt vom Licht getroffen werden besonders hervor; dadurch entsteht ein Blendungseffekt. Stellen die direkt im Licht stehen leuchten besonders. Da es sich um einen post-processing-Effekt handelt, wird die Szene zunächst mit Hilfe eines FBO (framebuffer object) in eine Textur gerendert. Anschließend werden mit Shaderprogrammen helle Pixel in eine andere Textur extrahiert und diese per Gauß-Filter (5x5 pixel window) in weiteren 2 passes geblurrt. Das Ergebnis wird dann über die eigentliche Szene gelegt, sodass helle Stellen und deren nähere Umgebung besonders hervorgehoben werden. Information dazu haben wir in [1], [2], [3], [4] und [5] gefunden.

Shadow maps

Dynamische Schatten haben wir mit Hilfe von shadow maps [6, 7, 8, 9] umgesetzt. Shadow maps werden in den meisten aktuellen Spielen eingesetzt, weil sie simpler als shadow volumes zu berechnen sind. Die Idee dabei ist, dass die Szene zunächst aus der Sicht der Lichtquelle gerendert wird. In diesem ersten pass interessiert uns nur die Tiefe der dem Licht nächsten Flächen (auf welche demzufolge das Licht trifft, welche aber evtl. andere Flächen verdecken). Daher wird wieder ein FBO verwendet, sodass der Tiefen-Puffer in eine sog. shadow-map-Textur (2048x2048 Pixel) gerendert wird.

Beim eigentlichen Rendern der Szene wird diese shadow map von der Lichtquelle aus auf die Szenerie projeziert; jedes Fragment führt einen look-up durch, erhält die entsprechende Referenztiefe und vergleicht diese mit der eigenen. Ist die Tiefe größer, ist das Fragment verdeckt, und die diffusen und spekulären Beleuchtungsterme fließen nicht in die Beleuchtungsgleichung ein. Wir benutzen ein omnidirektionales Punktlicht und daher eine (für die Szene maßgeschneiderte) perspektivische Projektion, da direktionales Licht in einem geschlossenen Raum unrealistisch ist.

Um das aliasing etwas zu kaschieren, benutzen wir PCF (percentage-closer filtering) in einem 3x3-Fenster. Hier wird die eigene Tiefe mit den 9 umgebenden Referenztiefen verglichen und die 2 Beleuchtungsterme entsprechend skaliert.

Normal mapping

Um zum Echtzeitrendering geeignete meshes mehr Oberflächendetails und Detailtreue einzuhauchen, haben wir normal mapping [10] eingesetzt. Hierbei wird für das Blinn-Phong-shading der Normalvektor eines Oberflächen-Fragments durch einen look-up aus einer sog. normal-map-Textur gelesen. Dadurch ist ein Großteil des Detailgrads eines detaillierten meshes auf ein entsprechendes, in der Polygonanzahl reduziertes mesh übertragbar, Performance und Speicherverbrauch werden optimiert. Wir haben aufgrund der größeren Flexibilität das normal-mapping im tangent-space betrieben. Zum Verständis der TBN-Matrix hat [11] wesentlich beigetragen

So haben wir aus einem ursprünglich fast eine Million Dreiecke umfassenden Drachen ein mesh aus 50.000 Dreiecken kreiert und uns mit dem nvidia-Tool Melody eine normal map generieren lassen. Das ursprüngliche Raptor-Modell bestand aus genau 2 Millionen Dreiecken, unser mesh besitzt nur mehr wiederum 50.000 – bei sehr gut erhaltenem Detailreichtum.

Specular mapping wurde ebenfalls integriert. Hierbei bestimmt eine Graustufen-Textur den Reflexionsgrad der Oberflächenfragmente. So zaubern z.B. metallische Oberflächen hübsche Glanzeffekte auf den Monitor.

Parallax mapping

Parallax mapping haben wir auch implementiert, da wir bei der Suche nach geeigneten Boden- und Wandtexturen auf nette Pakete mit diffuse, normal und displacement maps gestoßen sind. Dieser Effekt erlaubt plastischeres bump mapping, da hierbei konventionell der Blickwinkel nicht berücksichtigt wird und die vorgetäuschte zusätzliche Geometrie daher nur bei senkrecht zur Oberfläche stehendem Blickvektor realistisch wirken kann. Beim parallax mapping hingegen wird eine displacement map (Graustufen-Textur, welche das Höhenprofil der eigentlich ebenen Fläche definiert) dazu benutzt, um abhängig von jeweiliger Höhe und Blickvektor die Texturkoordinaten so zu verschieben, dass an diesem Fragment das eigentlich sichtbare Fragment (also dieses, welches bei einer reell displaceden Geometrie sichtbar wäre) gerendert wird. An den Rändern ist aber natürlich Schluss mit Vortäuschen, da lässt sich ohne zusätzliche Polygone nichts mehr machen. Da der Verschiebungsvektor bei extremen Sichtwinkeln sehr groß und daher Artifakte sichtbar werden, haben wir ein Minimum von 30° gewählt.

Animierte Objekte

Die Engine verfügt über ein simples keyframe-animation-Verfahren zur Animation von Objekten. Die Interpolation wurde mit Hilfe von erweiterten Befehlssätzen (SSE) realisiert. Doch aufgrund der knappen Zeit und mangelnder 3D-Modellierungs-Erfahrungen wollten wir kein Objekt mühselig animieren, weshalb dieses feature leider nicht demonstriert wird.

Die geometrischen Daten werden mit Hilfe von vertex buffers an die Grafikkarte weitergegeben, wobei geometric-instancing für nicht-animierte Objekte eingesetzt wird.

Gespeichert werden unsere Modelle in einem eigenen binären Format, in dem wir auch direkt den Tangent-Vektor zur Berechnung der TBN-Matrix integrieren, um die Ladezeiten klein zu halten.

View-frustum culling, S3-Texturkompression

View-frustum culling und S3-Texturkompression sind mehr oder weniger Standardfeatures heutiger 3D-Renderer - wir haben sie ebenfalls eingesetzt. View-frustum culling sorgt dafür, dass nur Objekte

gerendert werden, welche sich im sichtbaren Pyramidenstumpf (sog. view frustum) befinden. Wir haben dies durch eine simple bounding sphere unserer Objekte realisiert.

Die Texturkompression spart einiges an Videospeicher, da sich Texturen generell sehr gut (verlustbehaftet) komprimieren lassen. Wir unterstützen die DXT-Texturformate und können auch vorkomprimierte DDS-files direkt laden. Für die diffuse maps wird DXT1 eingesetzt (Kompressionsrate 1:6), für normal maps das DXT5_NM-Format (1:3), bei dem nur die (x,y)-Koordinaten der Normalen gespeichert werden. Natürlich haben wir eine anisotropische Texturfilterung eingesetzt.

[1] http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf
[2] http://de.wikipedia.org/wiki/High_Dynamic_Range
[3] http://en.wikipedia.org/wiki/Bloom_(shader_effect)
[4] http://www.gamasutra.com/features/20040526/james_01.shtml
[5] http://www.gamedev.net/community/forums/topic.asp?topic_id=405591
[6] http://en.wikipedia.org/wiki/Shadow_mapping
[7] http://developer.nvidia.com/object/hwshadowmap_paper.html
[8] http://www.gamedev.net/community/forums/topic.asp?topic_id=393224&forum_id=25&gforum_id=0
[9] http://www.paulsprojects.net/tutorials/smt/smt.html
[10] http://www.blacksmith-studios.dk/projects/downloads/bumpmapping_using_cg.php
[11] http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php