Abgabemodalitäten

Testatstermin: Dienstag, 23.01.2018 (während der Übung)

Das lauffähige Programm, mit allen Unteraufgaben, wird in der Übungsstunde testiert. Zusätzlich bitte vorher den Code im SVN-Gruppenordner hochladen. Während des Testats sind Verständnisfragen und ein Blick über den Code möglich. Alle Fragen sollten von allen Gruppenmitgliedern beantwortet werden können. Wenn eine Aufgabe nicht vollständig funktioniert gibt es Teilpunkte, falls der Versuch in die richtige Richtung geht.

Sie können das Programm gerne selbständig weiter ausbauen. Schöne Ergebnisse werden mit Bonuspunkten belohnt. Weiter werden wir die Bilder gerne auf die GDV1-Moodleseite stellen.

Hinweis: Bitte das Programm auf **release** (optimiert) erstellen. Das kann bei OpenGL einen riesigen Performance-Sprung ausmachen.

3. Programmieraufgabe - 10 Punkte (+2 Bonus)

In dieser Übung soll eine Szene in einer etwas *hübscheren* Umgebung gezeichnet werden. Um die GPU zu entlasten soll das View Frustum Culling auf der Anwendungsebene implementiert werden. Dadurch lassen sich auch komplexe Szenen mit vielen Objekten effizient zeichnen, sofern nicht alle Objekte auf einmal sichtbar sind.

a) **Terrain-Modellierung und -Rendering**

In dieser Aufgabe soll eine Szene, beschrieben durch ein Höhenfeld, erzeugt und gerendert werden.

Da ein rein zufälliges Höhenfeld unnatürlich aussieht, soll ein Algorithmus implementiert werden, der natürlich wirkende Gebirgsformationen generiert. Einige mögliche Algorithmen werden in [1] und [2] vorgestellt. Implementieren Sie einen der vorgestellten Algorithmen um Höhenfelddaten über einem quadratischen Gebiet zu erzeugen.

Nachdem ein Höhenfeld erzeugt wurde, muss es noch gerendert werden. Berechnen Sie dazu aus den Höhendaten 3D-Eckpunkte. Aus der Struktur des quadratischen Höhenfeldes können direkt die Indizes der Eckpunkte für Dreiecks- oder Viereckснetze entnommen werden. Benutzen Sie VBOs, sofern es die GPU unterstützt.

Einfarbige Gebirgsformationen erscheinen nicht sehr plastisch. Färben Sie daher das berechnete Netz je nach Höhenlage ein, beispielsweise von hohen zu niedrigen Regionen Weiß (Schnee & Eis), Grau, Braun, Hellgrün, Dunkelgrün, Blau (Wasser). Bestimmen Sie zu jedem Eckpunkt eine Farbe und benutzen Sie die Funktion **glColorPointer** (analog zu **glVertexPointer** und **glNormalPointer**). Damit das color array auch mit eingeschaltetem Lighting funktioniert, müssen die Befehle

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

verwendet werden (Stichwort *Color Tracking* im Lighting Tutorial [3]). Im Framework wird dies in der Initialisierung bereits gemacht.

Entwickeln Sie den Algorithmus nach belieben weiter, bis Sie mit dem Ergebnis zufrieden sind. Möglichkeiten sind zum Beispiel das Einziehen einer festen Wasserebene, Rauschen auf die Farben zu bringen, Farben über ihre Nachbarn zu glätten uvm..

3 Punkte

Prof. Dr. Dieter W. Fellner
M.Sc. Roman Getto
Julian Zobel

b) **Skyboxen**

Skyboxen sind ein schönes Mittel, um einfach und effizient ein Panorama um die Szene zu legen. Hierfür werden die sechs Seiten eines die Szene enthaltenden Würfels mit dem Panorama (Himmel, Berge, etc.) **bei ausgeschaltetem** z-Buffer texturiert.

Gehen Sie schrittweise vor und lesen Sie zunächst die Texturen ein. Ein `.bmp`-Reader ist bereits im Framework enthalten. Wir geben drei Sätze von Skybox-Texturen vor. Sie können auch eigene Skyboxen mit dem Tool *Terragen* erzeugen.

Zeichnen Sie zunächst ein beliebiges Quadrat, dass Sie mit einem der Bilder der Skybox texturieren. Wenn das texturieren eines Quads funktioniert, erstellen Sie die gesamte Skybox mit allen sechs Seiten. Achten Sie darauf, dass es niemals möglich ist, den Rand einer Skybox zu erreichen! Ein gutes Tutorial zu Skyboxen incl. Terragen finden Sie unter [4] und [5].

3 Punkte

c) **View Frustum Culling**

Implementieren Sie das *View Frustum Culling* (VFC).

Um das VFC umzusetzen müssen zunächst die sechs *clipping planes* des Sichtvolumens aus dem Produkt der Projektions- und Modelview-Matrix bestimmt werden. Die Matrizen können mittels **glGetFloatfv** ausgelesen werden. Beachten Sie, dass OpenGL die Matrizen spaltenweise als lineares Array verwaltet. Eine ausführliche Anleitung finden Sie unter [6]. Alternativ können Sie auch wie unter unter [7] gezeigt vorgehen. Weitere Erklärungen finden sie unter [8] und [9].

Jede clipping plane liegt dann in Form einer Normalen $\mathbf{n} = (a, b, c)^\mathsf{T}$ sowie dem Abstand zum Ursprung d vor (implizite Geradengleichung, bekannt als Vier-Punkt-Form oder *Hesse'sche Normalform*). Ein Punkt $\mathbf{x} = (x_1, x_2, x_3)^\mathsf{T}$ liegt auf der der Normalen zugewandten Seite der Ebene, falls $\mathbf{x} \cdot \mathbf{n} - d = ax_1 + bx_2 + cx_3 - d > 0$ und auf der abgewandten Seite, falls $\mathbf{x} \cdot \mathbf{n} - d < 0$.

Um die Sichtbarkeit der Objekte zu bestimmen, müssen bounding volumes der Dreiecksnetze gegen die sechs clipping planes des view frustums getestet werden. Implementieren Sie die Logik, mittels der entschieden werden kann, wann ein Objekt (bzw. seine bounding box) nicht im Sichtvolumen liegt und verworfen werden kann.

Platzieren Sie mehrere (viele!) Objekte in Ihrer Szene. Führen Sie in jedem Frame für jedes Objekt das View Frustum Culling durch. Nur wenn das Objekt den Sichtbarkeitstest besteht, wird es gezeichnet. Geben Sie aus, wie viele Objekte gezeichnet bzw. nicht gezeichnet werden.

4 Punkte

d) **Bonus: Texturierung**

Als **optionale** Zusatzaufgabe können mit kleinen Erweiterungen ganze Modelle texturiert werden. Dazu benötigen wir zu jedem Eckpunkt Texturkoordinaten, die beispielsweise mit dem *Zweischrittverfahren* berechnet werden können. Im ersten Schritt wird ein Eckpunkt auf einen einfachen Hüllkörper projiziert. Im zweiten Schritt wird über die Geometrie des Hüllkörpers dem Eckpunkt eine Texturkoordinate zugewiesen.

Im Framework muss dabei die TriangleMesh Klasse mit einem Vektor für Texturkoordinaten erweitert werden (analog zu den Vektoren für die Vertices, Normalen und Farben). Weiter benötigt jedes Modell eine ID für eine Textur.

Das einfachste Zweischrittverfahren ist das *sphere mapping*. Dabei wird ein Eckpunkt durch Zentralprojektion auf eine Einheitskugel projiziert. Dem projizierten Punkt wird nun eine uv-Koordinate zugewiesen. Dabei geht u um den Equator im Interval $[0,1]$ und v von Pol zu Pol im Interval $[0,1]$. Das Zweischrittverfahren kann in eine Formel gefasst werden, die jedem Eckpunkt direkt eine Texturkoordinate zuweist. Sei $(x,y,z)^\mathsf{T}$ der Vektor vom Ursprung des Modells (z.B. Mittelpunkt der AABB) zum Eckpunkt, dann wird die Texturkoordinate durch

$$u = \frac{1}{2\pi} \cdot \texttt{atan2}(x,z) + \frac{1}{2}$$
$$v = \frac{1}{\pi} \cdot \sin^{-1}\left(\frac{y}{\sqrt{x^2+y^2+z^2}}\right) + \frac{1}{2}$$

berechnet.

Texturieren Sie nun die Objekte der Szene. Berechnen Sie dazu die Texturkoordinaten jedes Eckpunktes und schalten Sie wie zuvor bei der Skybox die Texturierung an (diesmal mit aktivem z-Buffer). Wenn die Objekte mit VBOs gezeichnet werden, müssen entsprechende Arrays angelegt werden (z.B mit std :: vector<Pair<**float**, **float**>>).

2 Punkte

Literatur

[1] OpenGL Video Tutorial: Terrain `http://www.videotutorialsrock.com/opengl_tutorial/terrain/text.php`

[2] Terrain Tutorial: The Fault Algorithm. `http://www.lighthouse3d.com/opengl/terrain/index.php?fault`

[3] Lighting Tutorial. `http://www.falloutsoftware.com/tutorials/gl/gl8.htm`

[4] DelphiGL: Tutorial Skyboxen `http://wiki.delphigl.com/index.php/Tutorial_Skyboxen`

[5] Sidvind: Tutorial Skyboxen `http://sidvind.com/wiki/Skybox_tutorial`

[6] Gil Gribb, Klaus Hartmann.
Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix
`http://www.cs.otago.ac.nz/postgrads/alexis/planeExtraction.pdf`

[7] Lighthouse 3D's View Frustum Culling Tutorial. `http://www.lighthouse3d.com/tutorials/view-frustum-culling/`

[8] Modelview Matrix `http://3dengine.org/Modelview_matrix`

[9] Camera Position `http://3dengine.org/Right-up-back_from_modelview`