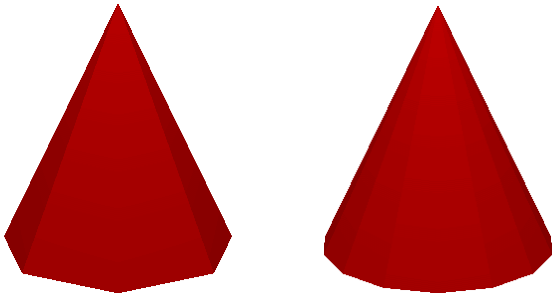
3D-Computergrafik I

Wintersemester 2015/16 – Aufgabenblatt 6

Hasso-Plattner-Institut
Fachgebiet Computergrafische Systeme

Aufgabe 6.1: Anwendung von OpenGL-Buffern (6 Punkte)



(a) 8 Seitenflächen (b) 16 Seitenflächen

Abbildung 1: Gerenderter Kegel mit 8 bzw. 16 Seitenflächen unter Verwendung von Flächennormalen.

Viele zum Rendern benötigte Daten werden mit Hilfe von Buffern auf die GPU übertragen. Für 3D-Modelle werden beispielsweise *Vertex Buffer Objects* (VBO) und *Vertex Array Objects* (VAO) verwendet. In dieser Aufgabe sollen Sie die Geometrie eines Kegels CPU-seitig berechnen und mittels OpenGL rendern (Abb. 1). Nutzen Sie dafür ausschließlich die direkten OpenGL-Befehle. Die Nutzung von Kapselungen, wie z.B. `QOpenGLBuffer`, sind nicht erlaubt. Die einzige Ausnahme stellt die Verwendung von `QOpenGLShaderProgram` dar.

Erweitern Sie die Implementierung der Klasse `Exercise61` wie folgt:

a) In der Methode `initialize` müssen alle notwendigen Buffer erzeugt werden. Legen Sie sich dazu ggf. notwendige Membervariablen an. Ergänzen Sie ggf. weitere Methoden um die reservierten Ressourcen wieder freizugeben, sobald diese nicht mehr benötigt werden.

b) Implementieren Sie die Methode `updateCone`, die die Buffer mit den für das Rendering eines Kegels benötigten Daten befüllt. Beachten Sie dabei, dass die Anzahl an Seitenflächen durch die Variable `m_lateralSurfaces` spezifiziert wird und die Grundfläche aus einer minimalen Anzahl an Dreiecken bestehen soll. Implementieren Sie dabei auch eine Normalenberechnung - ob Sie die Normale pro Vertex oder je Fläche berechnen, ist Ihnen überlassen.

c) Ergänzen Sie die Methode `render` um den notwendigen Aufruf von `glDrawArrays` zum Rendern des Kegels.

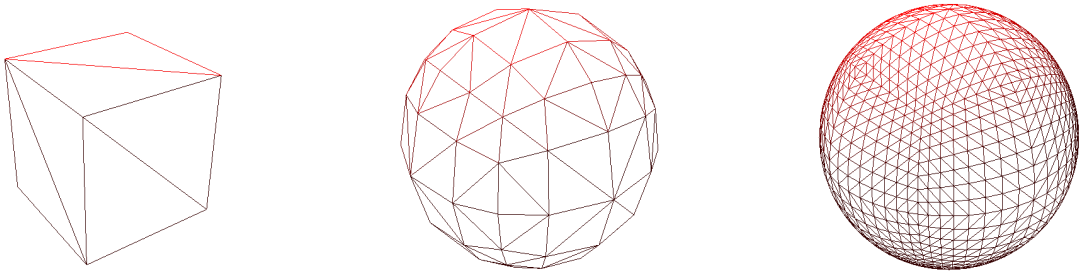
Aufgabe 6.2: Explosionseffekt durch Transformationen (7 Punkte)

Abbildung 2: Ein 3D-Objekt bei Beginn des Explosionseffekts.

In dieser Aufgabe sollen Sie ein 3D-Objekt "explodieren" lassen (Abb. 2), welches bereits vom Programmrahmen aus einer `.obj` ausgelesen wird. Das 3D-Objekt ist durch ein Dreiecksnetz definiert. Um einen Körper "explodieren" zu lassen, sollen Sie die einzelnen Dreiecke eines Körpers von seinem Ursprung weg translieren und um sich selbst rotieren lassen. Dies soll nicht zeitgesteuert, sondern durch Nutzerinteraktion passieren. Dazu sollen Sie mittels eines Sliders den Explosionsfortschritt selbst steuern.

Passen Sie hierzu den Funktionsrumpf von `main()` im Shader `model.geom` an. Der Explosionsfortschritt soll aus der Variable `animationFrame` ($\in [0; 1]$) ausgelesen werden. Die genauen Translations- und Rotationsrichtungen stehen Ihnen dabei frei. Die oben genannten Kriterien sollen jedoch erfüllt werden. Beachten Sie, dass Matrizen in GLSL **column-major** definiert werden.

Aufgabe 6.3: Tessellation (6 Punkte)



(a) Tessellations-Level 1 (b) Tessellations-Level 4 (c) Tessellations-Level 16

Abbildung 3: Tessellation eines Würfels zur Approximierung einer Kugel.

Ziel dieser Aufgabe ist es, mit Hilfe des OpenGL Tessellators eine Kugel zu approximieren. Dazu wird ein Würfel als Ausgangsgeometrie verwendet. Für die Approximierung werden die Seitenflächen des Würfels nicht nur feiner tesselliert, sondern es werden auch die Vertices der neu geschaffenen Geometrie auf die Oberfläche der Umkugel des Würfels transformiert. Für die Lösung der Aufgabe bearbeiten Sie die `exercise63.cpp` und die dazugehören `cube`-Shader wie folgt:

a) Definieren Sie in der Methode `initialize` die Geometrie der 6 Seitenflächen des Würfels, indem Sie anstelle des vorgegebenen Dreiecks die Vertices der Seitenflächen in der Variablen `m_vertices` ablegen. Achten Sie dabei darauf, dass alle Seitenflächen dieselbe Orientierung haben (d.h., im oder gegen den Uhrzeigersinn). Setzen Sie außerdem mittels OpenGL die Anzahl der Vertices pro Patch auf `4`.

b) Ändern Sie in der Methode `render` den von `glDrawArrays` verwendeten Primitivtyp auf den, welcher für Tessellation-Shader benötigt wird.

c) Binden Sie in der Methode `initialize` den Rahmen des Tesselation-Control-Shaders, Tessellation-Evalulation-Shaders und Geometrie-Shaders ein, indem Sie die auskommentierten Zeilen zum

Hinzufügen dieser Shader wieder aktivieren. Außerdem deaktivieren Sie in der `cube.vert` die Zeile mit der Zuweisung auf `gl_Position`, da dies jetzt vom Geometrie-Shader übernommen wird.

d) Implementieren Sie den Tessellation-Control-Shader in der `cube.tcs`. Achten Sie dabei darauf, mit Hilfe der `layout` Spezifikation für die Ausgabe des Shaders die Anzahl der Vertices pro Patch anzugeben. Außerdem sollen sowohl das innere als auch das äußere Tessellierungslevel auf den Wert der bereits vorhandenen Variable `animationFrame` gesetzt werden.

e) Implementieren Sie den Tessellation-Evaluation-Shader in der `cube.tes`. Achten Sie dabei darauf, mit Hilfe der `layout` Spezifikation für die Eingabe des Shaders den Primitivtyp und die Orientierung der Primitive passend zur Eingabe anzugeben. Außerdem müssen Sie den Ausgabe-Vertex `te_vertex` für die Ausgabe des Shaders unter Verwendung der Eingabe-Vertices `tc_vertex[]` des aktuellen Patches anhand der von OpenGL gegebenen Tessellierungskoordinate interpolieren. Ferner soll `te_vertex` auf eine feste Länge normalisiert werden.

f) Erweitern Sie den Geometrie-Shader in der `cube.geom` um die Berechnung von per Patch-Normalen.

g) Binden Sie ein einfaches Shading ein, indem Sie in der `cube.frag` die Platzhalterfarbe (rot) durch die auskommentierten Zeilen ersetzen.

Aufgabe 6.4: Code Qualität (1 Punkt)

Bei dieser Aufgabe sollen mit Hilfe von *Cppcheck*[1] möglicherweise vorhandene, problematische Stellen im Quellcode erkannt und verbessert werden. D.h., dass als Endergebnis Cppcheck keine Verbesserungsvorschläge für den Quellcode der vorangegangenen Aufgaben hat. Als Referenz gilt *Cppcheck 1.70*[2] und die Aktivierung aller Anzeigekategorien bis auf die Kategorie „Information“.

Allgemeine Hinweise zur Bearbeitung und Abgabe

- Die Aufgaben sollen maximal zu zweit bearbeitet werden; Ausnahmen sind nicht vorgesehen. Je Gruppe ist nur eine Abgabe notwendig.
- Bitte reichen Sie Ihre Lösungen bis **Dienstag, den 05. Januar um 13:30 Uhr** ein.
- Die Aufgaben können auf allen wesentlichen Plattformen (Windows, Linux, OS X ($\geq$ 10.09)) bearbeitet werden.
- Die Aufgaben benötigen mindestens *OpenGL 4.0*, welches von allen halbwegs aktuellen GPUs unterstützt wird. Weichen Sie ggf. auf ein Alternativgerät (von Ihnen oder Ihrem Übungspartner) aus oder verwenden Sie einen PC im Grafikpoolraum im Erdgeschoss des HPI Hauptgebäudes. Alternativ kann die Software-basierte Implementierung von MESA verwendet werden, wobei der aktuelle Entwicklungszweig benötigt wird.
- Die Durchsicht zur Bewertung kann sowohl auf Windows, Linux oder OS X erfolgen. Es wird folglich eine **plattformunabhängige Lösung** der Aufgaben erwartet.
- Archivieren (Zip) Sie zur Abgabe Ihren bearbeiteten Programmrahmen und ergänzen Sie Ihre Matrikelnummer(n) im Bezeichner des Zip-Archivs entsprechend im folgenden Format: `uebung6_matrikNr1_matrikNr2.zip`. Geben Sie die schriftlich bearbeiteten Aufgaben als Textdokument (.txt) oder PDF im Archiv hinzugefügt ab. Beachten Sie, dass dabei nur die vollständigen Quelltexte, die CMake-Konfiguration sowie eventuelle Zusatzdaten gepackt werden (alles was im gegebenen Programmrahmen entsprechend vorhanden ist). Senden Sie uns keinesfalls Kompilate und temporäre Dateien (*.obj, *.pdb, *.ilk, *.ncb etc.). Testen Sie vor dem Verschicken, ob die Projekte aus den Zipdateien fehlerfrei kompiliert und ausgeführt werden können.
- Reichen Sie Ihr Zip-Archiv im Moodle ein: `https://moodle.hpi3d.de/course/view.php?id=81`.

[1] `http://cppcheck.net/`
[2] `http://sourceforge.net/projects/cppcheck/files/cppcheck/1.70`