Aufgabe Graphikprogrammierung

3 Praxisaufgaben zur Graphikprogrammierung mit OpenGL

Hinweise zur Implementierung: Der Übung liegen verschiedene Materialien bei. Dabei handelt es sich um eine Musterlösung (für Windows), in denen Sie die Ergebnisse der einzelnen Aufgaben betrachten können, ein lauffähiges Programm als Quelltext, dem die von Ihnen im Rahmen der Aufgaben zu erstellenden Algorithmen fehlen und eine allgemeine Erklärung zum Programmaufbau. Die Programmstellen, denen Sie Quelltext hinzufügen müssen, sind gesondert markiert und enthalten weitere Hinweise zur Lösung der Aufgabe. Des Weiteren finden Sie den Quellcode der Musterlösung unter `src_solution`.

Die Bedienung der Musterlösung (und Ihres Arbeitsprogrammes) erfolgt mittels eines Kontextmenüs, das Sie über die rechte Maustaste erreichen. Neben den einzelnen Menüpunkten finden Sie in Klammern den entsprechenden Tastatur-Shortcut. Außerdem können Sie das Raster mit dem Mausrad zoomen und durch Gedrückthalten des Rades und Bewegen der Maus verschieben.

Projektdateien für Visual-Studio 2012/2013, 2015 und 2017 finden Sie in den entsprechenden Unterverzeichnissen des Ordners `build`. Falls Sie nicht mit Visual-Studio arbeiten, finden Sie im Unterordner `build/cmake` entsprechende CMake-Dateien. Bitte beachten Sie, dass die Lösung auf den Rechnern im Rechenzentrum, auf denen sich Visual-Studio 2013 befindet, laufen muss! Laden Sie für die Abgabe *nur* die Quellcode-Dateien in einer einzigen `.zip`-Datei hoch.

In der Implementierung existiert eine Oberklasse `abstract_scene`, welche die gemeinsamen Funktionalitäten aller Szenen kapselt. Diese besitzt die Methode `render` welche die OpenGL-Anweisungen zur Darstellung enthält. Die zu vervollständigenden Szenen sind von dieser Klasse abgeleitet. Die Grundgerüste dieser Klassen sind bereits vorgegeben.

3.1 Einfache Körper und Transformationen

In der Vorlesung wurden Grundlagen von OpenGL, insbesondere das Definieren einfacher dreidimensionaler Körper und Objekt-Transformationen wie Verschiebung, Drehung oder Skalierung behandelt. Ziel dieser Praxisaufgabe ist es, die Erstellung eines Würfels als einer der grundlegenden Körper und die Darstellung mehrerer solcher Objekte, nachzuvollziehen. Außerdem sollen Transformationen hierarchisch angewandt werden.

3.1.1 Erstellen eines Würfels

Vervollständigen Sie die Methode `render_cube` der Klasse `cube_system`, die Sie in der Datei `cube_system.cpp` finden, um die Erstellung eines Würfels. Nutzen Sie die Befehle `glVertex3d` und `glColor3d` zur Festlegung der einzelnen Seiten. Dabei soll jede Seite eine andere Farbe besitzen. Der Würfel soll bei der lokalen Position $(-1 \quad -1 \quad -1)$ beginnen und bei $(1 \quad 1 \quad 1)$ enden. Es ist nicht erlaubt vorgefertigte Befehle wie `glutSolidCube` zu verwenden.

3.1.2 Darstellung mehrerer transformierter Würfel

In der Vorlesung haben Sie gelernt, dass Transformationen über die Matrixmultiplikation zu einer Gesamttransformation akkumuliert werden, die auf alle danach spezifizierten Vertices angewandt wird. Der Zustand dieser Gesamttransformation kann mit dem Befehl `glPushMatrix` auf einem Stack gesichert und mittels `glPopMatrix` vom Stack gelöscht und reaktiviert werden. Auf diese Weise lassen sich

beispielsweise Transformationsbäume linear beschreiben. Ihre Aufgabe besteht darin, in der Methode `render_system` der Klasse `cube_system` den in Abbildung 1 beschriebenen Transformationsbaum zu serialisieren, also als fortlaufende Folge von Anweisungen darzustellen. Für die Animationen existiert im Quellcode eine Variable `angle`, welche den aktuellen Rotationswinkel bestimmt - Sie müssen den Wert dieser Variable nicht manuell einstellen. Nutzen Sie zum Rendern der Würfel Ihre in der vorherigen Aufgabe entwickelte Methode `render_cube`.

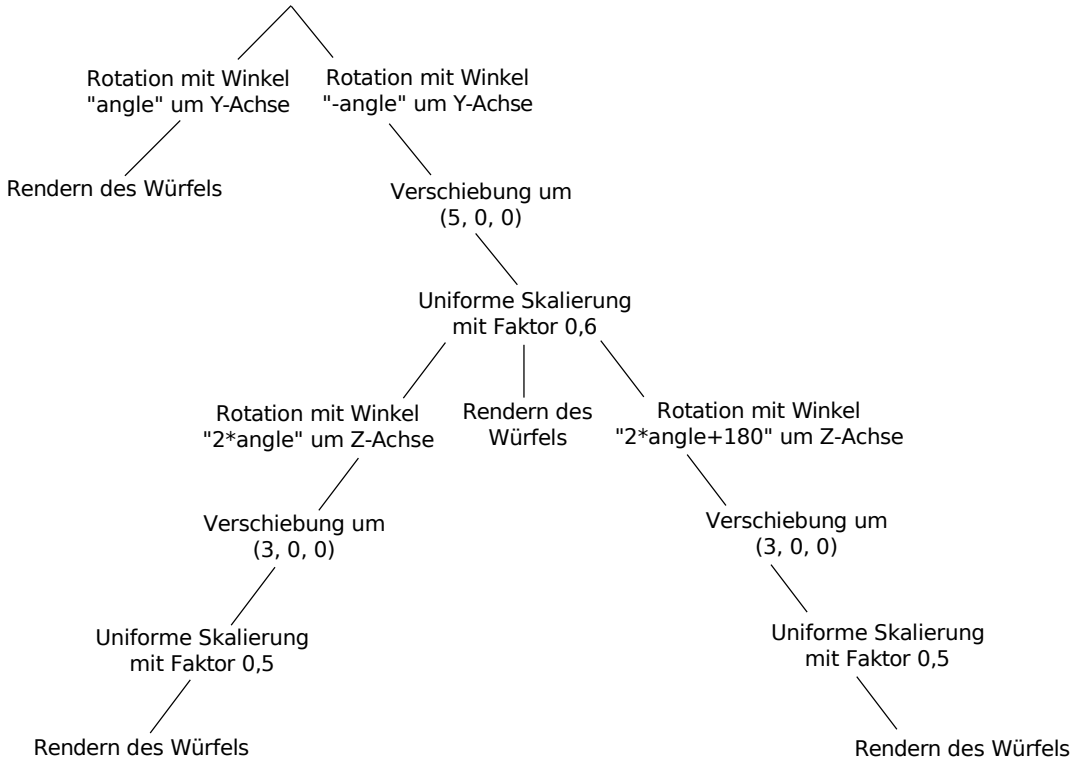

Abbildung 1: Der umzusetzende Transformationsbaum

3.2 Terrainvisualisierung

In dieser Aufgabe sollen die Grundlagen von polygonalen Netzen, Beleuchtung und Normalenberechnung anhand eines Beispiels nachvollzogen werden. Entwickeln Sie dazu schrittweise ein beleuchtetes und texturiertes Terrain. Auf einem regelmäßigen Gitter werden dabei zunächst aus einer BMP-Datei gewonnene Höhendaten abgebildet. Anschließend werden Normalen berechnet um Beleuchtungseffekte zu ermöglichen. Schließlich wird das Terrain mit einer Textur versehen. Die Resultate der einzelnen Schritte sind in Abbildung 2 dargestellt.

Es existieren verschiedene Darstellungsmodi. Mit der Taste `s` wird bei aktiviertem Terrain eine texturierte und beleuchtete Sicht gerendert, während durch die Taste `w` zu einer Drahtgitteransicht gewechselt wird. Die entsprechenden Optionen sind auch über das Kontextmenü verfügbar.

Die Implementierung erfolgt in der Klasse `terrain`, die sich in der Datei `terrain.cpp` befindet. Dabei ist der Quellcode für das Laden der Höhendaten und der Textur, das Einstellen der Projektion und Betrachterposition sowie die Aktivierung von Lichtquellen bereits implementiert. Je nach gewählter Option erfolgt innerhalb der Zeichenmethode `render` der Aufruf der Methode `render_solid_terrain`

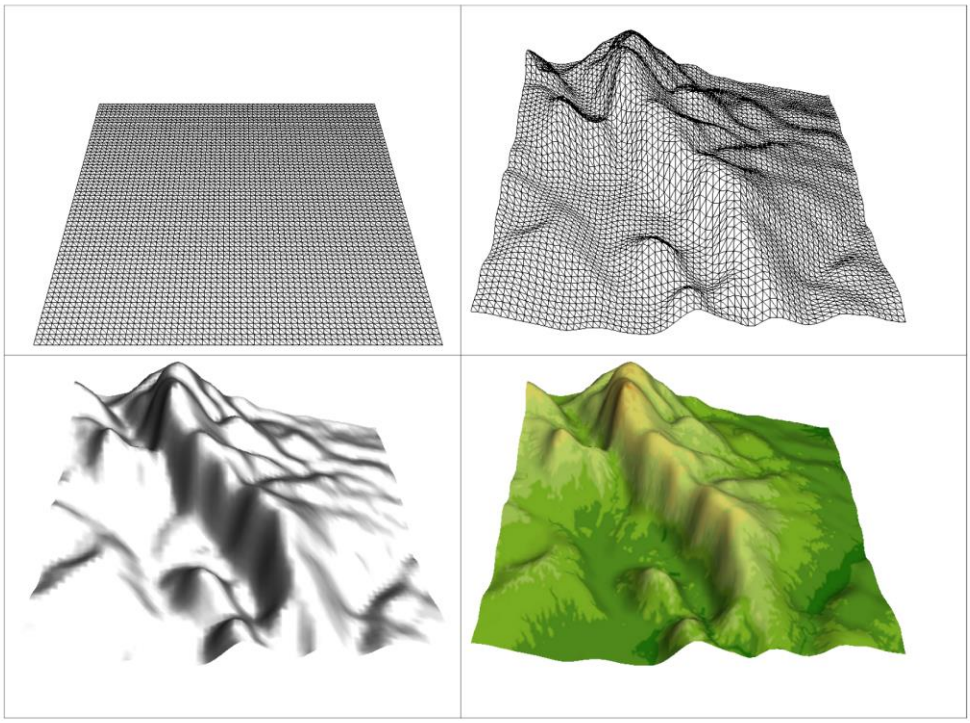

Abbildung 2: Die einzelnen Entwicklungsschritte von oben links nach unten rechts

oder `render_wireframe_terrain`, der die entsprechenden Zustände in OpenGL aktiviert und anschließend die Methode `render_terrain` aufruft, in der die eigentlichen Terraininformationen an OpenGL übergeben werden.

Verwenden Sie zur besseren Kontrolle für die ersten beiden Teilaufgaben die Drahtgitterdarstellung.

3.2.1 Erstellung eines regelmäßigen Gitters

Erweitern Sie die Methode `render_terrain` aus der Datei `terrain.cpp` um die Funktionalität ein regelmäßiges Gitter darzustellen (siehe linkes oberes Bild aus Abbildung 2). Setzen Sie das Gitter zeilenweise aus einzelnen Dreiecksstreifen zusammen. Beachten Sie dabei die richtigen Parameter für die `glBegin`-Methode und die Reihenfolge der `glVertex3d`-Befehle. Das Gitter soll sich auf der X-Z-Ebene erstrecken. Bereits vorgegeben ist eine Verschiebung und eine Skalierung durch die Sie pro Knoten einen Abstand von einer Einheit verwenden und die Knotengenerierung in X und Z bei Null beginnen lassen können.

3.2.2 Erstellung des Drahtgitter-Terrains

Mit der vorgegebenen Methode `get_heightmap_value(x, y)` wird der Höhenwert an der Stelle (x, y) der Höhenkarte zurückgegeben. Diese Methode behandelt unter anderem Bereichsverletzungen, weshalb Sie darauf hier nicht weiter achten müssen. Verschieben Sie nun jeden Knoten in Y-Richtung um den entsprechenden Höhenwert. Dabei können Sie auch hier durch die obige Skalierung von ganzen Einheiten ausgehen.

3.2.3 Berechnung von Normalen

Damit Flächen beleuchtet werden können benötigt OpenGL Informationen über die Normalenrichtung, also einen Vektor, der senkrecht zur Fläche steht. Für eine weichere Darstellung der Beleuchtung werden Normalen pro Knoten erstellt. In dieser Aufgabe sollen Sie für jeden Knoten die Normale berechnen. Dazu existiert die noch leere Methode `set_normal(x, y)`. Stellen Sie zunächst sicher, dass diese Methode mit den richtigen Parametern pro Vertex aufgerufen wird. Berechnen Sie anschließend dort die Normale, indem Sie zunächst die Richtungsvektoren vom aktuellen Knoten zu einem Nachbarn finden. Dabei kann der nächste Nachbar (Vorwärtsdifferenzen), der vorherige (Rückwärtsdifferenzen) oder der nächste und vorherige (Zentraldifferenzen) verwendet werden. Sie brauchen sich dabei nicht um Randbedingungen oder Bereichsverletzungen kümmern, da sie von der benötigten Methode `get_heightmap_value` behandelt werden.

Bestimmen Sie mittels einem der oben vorgeschlagenen Ansätze die Differenzvektoren in X- und Z-Richtung und daraus durch Vektoroperationen die Normale, die danach an OpenGL übergeben werden muss. Weitere Informationen zum Rechnen mit Vektoren in diesem Projekt finden Sie in den Kommentaren zu dieser Teilaufgabe. Das Resultat sollte dem in der unteren linken Abbildung entsprechen.

3.2.4 Texturierung des Terrains

Um einen realistischeren Eindruck zu erzeugen eignen sich Texturen. Nachdem die Texturdaten geladen wurden, müssen je nach Anwendung verschiedene Einstellungen aktiviert werden. Zum Rendern wird die Textur gebunden und anschließend pro Vertex Texturinformationen gesendet. Das Laden der Texturdaten und Einstellen der Textur wurde bereits implementiert. Aktivieren Sie in der Methode `render_solid_terrain` zunächst die Verwendung zweidimensionaler Texturen. Binden Sie anschließend die geladene Textur mit dem Handle `texture_handle`. In der Methode `render_terrain` ist es schließlich ihre Aufgabe pro Vertex eine passende Texturkoordinate an OpenGL zu senden. Beachten Sie den Wertebereich für die Texturkoordinaten.

3.3 Zusatzaufgaben

Hinweis: Die erreichbare Bonuspunktezahl für eine korrekt gelöste Zusatzaufgabe finden Sie am Ende der jeweiligen Aufgabenbeschreibung. Maximal können jedoch nicht mehr als 3 Bonuspunkte von diesem Übungsblatt eingebracht werden.

- In der Musterlösung können Sie bei aktivierter Terrainvisualisierung mittels der Taste `l` Höhenlinien anzeigen lassen. Zur Generierung dieser Höhenlinien wurde das Marching-Squares-Verfahren verwendet. Implementieren Sie diesen Algorithmus in der Methode `create_level_line` für die im Parameter `level` vorgegebene Höhe. Legen Sie die gefundenen Linienstart- und -Endpunkte in der Liste `level_lines` ab. (1 Pkt.)
- Bisher werden in der vorgegebenen Implementierung pro angezeigtem Bild alle Informationen, beispielsweise die Normalen, neuberechnet. Da sie sich über mehrere Bilder jedoch nicht verändern würde es ausreichen das Ergebnis einmal zu bestimmen und in einer *Displayliste* abzulegen. Implementieren Sie diese Funktionalität. Ob eine Displayliste neu erstellt werden muss können Sie über die vorgegebene Variable `dl_valid` abfragen. Bei der Generierung werden Sie eine Variable als Handle benötigen. Greifen Sie dazu auf `dl_handle` zurück. (1 Pkt.)
- In der Vorlesung haben sie die Hauptrisse einer Szene kennengelernt. Stellen Sie die 3 Hauptrisse des Würfelsystems und eine perspektivische Version im Splitscreen dar. Verwenden Sie dafür die bereits vorhandene Klasse `cube_system_split` aus `cube_system_split.cpp`. Nutzen Sie für die

Hauptrisse othographische Projektionen und achten Sie darauf, dass Teile eines Hauptrisses nicht über ihr Fensterviertel hinauszeichnen. (1,5 Pkt.)

- Ein Transformationsbaum kann auch über rekursive Vorschriften erstellt werden. Schauen Sie sich in der Musterlösung die Szene *Recursive Cubes* an. Ein Würfel besitzt dabei rekursiv an jeder Seite einen weiteren kleinen Würfel, der lokal um 4 Einheiten verschoben und mit dem Faktor $0,5$ skaliert wurde (dies gilt jedoch nicht für die auf größere Würfel zeigenden Seiten). Nutzen Sie die Klasse `recursive_cubes` aus `recursive_cubes.cpp` zur Implementierung und stellen sie die Rekursion über die Methode `render_recursive` her, die die noch voranzuschreitende Rekursionstiefe im Parameter enthält. (1 Pkt.)

- Implementieren Sie alle Teilaufgaben der Aufgabe 3.2 nochmals mittels dem OpenGL Core-Profil ab Version 3.2. Das bedeutet, dass Sie weder auf die Fixed-Function-Pipeline, auf die vorgegebenen Methoden zur Transformationsmanipulation oder auf Elemente des Immediate-Modes (alles zwischen und inklusive `glBegin` und `glEnd`) zurückgreifen dürfen. Stattdessen müssen alle Daten in einem Vertex-Buffer-Object abgelegt werden. Das Rendern erfolgt über einen selbst erstellten Vertex- und Fragment-Shader. Für die Implementierung der nötigen Funktionen um Shader zu laden und zu kompilieren können Sie eine beliebige Implementierung verwenden. (2 Pkt.)