Institut für Visualisierung
und Interaktive Systeme

Dr. Guido Reina  Dr. Michael Krone

# 11. Übungsblatt zur Vorlesung Computergraphik im WS 2017/18

Abgabe am Mittwoch, 24.01.2018 (12:00 Uhr)
Besprechung am Montag, 29.01.2018 (17:30 - 19:00 Uhr)

**Hinweis**: Unterstützt werden Visual Studio 2015 und 2017. Bei der Installation müssen Sie unter „Einzelne Komponenten“ den „NuGet-Paket-Manager“ aktivieren. Sollten Sie Visual Studio 2017 verwenden, brauchen Sie zusätzlich das „Toolset für VC++ 2015.3 v140“. Diese Features lassen sich auch im Nachhinein mit dem Visual Studio Installer hinzufügen. Achten Sie bitte auch darauf, beim ersten Öffnen des Projekts, die verwendete Toolset-Version *nicht* auf v141 umzustellen (kein Upgrade).

Das Programmskelett greift für die Verwendung von OpenGL auf externe Open-Source-Bibliotheken zu. In der mitgelieferten Visual-Studio-Solution sind die Bibliotheken bereits eingebunden und werden (sofern der NuGet-Paket-Manager installiert ist) automatisch nachgeladen. Falls Sie einen anderen Compiler verwenden wollen, müssen Sie sich selbst um die Einbindung folgender Bibliotheken kümmern:

- glfw: http://www.glfw.org/
- glad: https://github.com/Dav1dde/glad
- glm: https://glm.g-truc.net

*Terrain [5 Punkte]*

Wenn Sie das Projekt öffnen und ausführen, wird zunächst nur ein Elefanten gerendert. Um die Ladezeit zu verkürzen, sollten Sie zudem das Projekt im *Release*-Modus kompilieren.

Mit den Tasten W, A, S, D, *Shift* und *Space* kann man sich im Raum bewegen. Die Pfeiltasten erlauben eine Änderung der Blickrichtung, die Tasten 1 und 2 verändern die Lichtrichtung.

**Mesh-Generierung** Damit der Elefant nicht in der Luft schwebt, soll zunächst ein einfaches Terrain gerendert werden, das aus einem (bereits fraktal generierten) Höhenfeld erstellt wird. Das Gelände soll aus $n \times n$ Vertices in der $xz$-Ebene bestehen, die als `GL_TRIANGLE` gezeichnet werden. Die Anordung der Dreiecke soll derer in Abbildung 1 entsprechen.

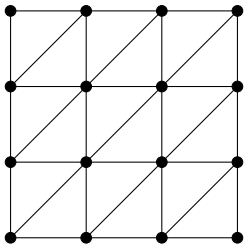

Abbildung 1: Wireframe eines Geländeausschitts in $xz$-Ebene.

Beantworten Sie zunächst theoretisch folgende Fragen zu dem beschriebenen Mesh, bestehend aus $n \times n$ Vertices und der Topologie `GL_TRIANGLE`, schriftlich:

- Wie viele Dreiecke müssen mindestens gezeichtnet werden, um das gesamte Terrain darzustellen?
- Wie viele Indices werden dafür benötigt?

Implementieren Sie nun die Generieung des Meshs für $n = 512$ (also $512 \times 512$ Vertices). Vervollständigen Sie dazu die Funktion `CreateGround` in `main.cpp`, indem Sie die Vertex- und Indexbuffer erstellen. Die Gitterpunkte entsprechen den ganzzahligen Koordinaten, die Höhe und Normale eines Gitterpunktes $(x, z)$ werden in den Funktionen `GetGroundHeight` bzw. `GetGroundNormal` berechnet. Jeder Vertex soll Information zu seiner Position, Farbe und Normalen enthalten. Als Farbe verwenden Sie bitte die vorgegebene `color`. Im Vektor `indices` stehen immer drei aufeinanderfolgende Integerwerte für ein Dreieck. Diese Integers repräsentieren die drei Indices der Vertices im Vektor `vertices`, aus denen das entsprechende Dreieck gebildet werden soll.

**Normalen** Die Vertexnormalen sollen in der Methode `GetGroundNormal` berechnet werden. Vervollständigen sie die Methode dahingehend, dass die korrekte Normale für den jeweiligen Vertex an der Stelle `coord` berechnet wird.

*Hinweise: Sollten Sie das Kreuzproduktes in Ihrer Berechnung verwenden, achten Sie auf eine korrekte Orientierung der Vektoren. Zum schnellen Debuggen der Normalen bietet es sich an, die Normale als Terrainfarbe zu übergeben (vgl. Abbildung 2).*

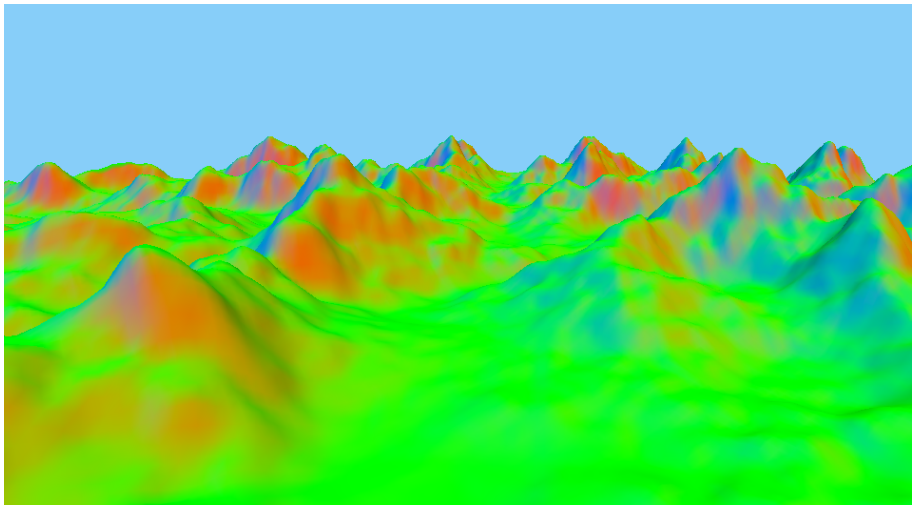

Abbildung 2: Terrain mit Vertexnormalen als Farbwerte.

*Licht & Schatten [5 Punkte]*

Eine Möglichkeit, in einer Szene Schatten zu berechnen, ist es, den Lichtstrahlen wie beim Raytracing zu folgen: Um festzustellen, ob ein bestimmter Punkt im Schatten liegt, folgt man dem Strahl von diesem Punkt zur Lichtquelle bzw. zu den Lichtquellen. Dies ist rechnerisch jedoch sehr aufwendig. Um trotzem Schatten in Echtzeit zu berechnen, gibt es die Methode des Shadow Mappings, die in dieser Aufgabe implementiert werden soll.

Beim Shadow Mapping wird die Geometrie zuerst aus der Perspektive der Lichtquelle gerendert (anstatt aus der Perspektive der Kamera). Die Lichtquelle hat dabei also ein eigenes Kamerakoordinatensystem (welches nichts mit der eigentlichen Kamera zu tun hat), eigene Clip-Koordinaten und eigene normalisierte Gerätekoordinaten. Bei diesem Renderdurchlauf sind nur die resultierenden Tiefenwerte ($z$-Komponente der normalisierten Gerätekoordinaten) relevant und werden in einer Textur – der sogenannten *Shadow Map* – gespeichert. D.h. die Shadow Map enthält einen Tiefenwert für jeden Punkt, der von der Lichtquelle aus sichtbar ist. Anschließend kommt der normale Renderdurchlauf, in dem die Geometrie wie gewohnt aus der Perspektive der Kamera gerendert wird. Im Fragment-Shader wird dann für jeden hierbei gerenderten Punkt berechnet, welche normalisierten Gerätekoordinaten dieser bezüglich der Lichtquelle hat. Hieraus ergeben sich direkt die Texturkoordinaten, an denen dieser Punkt in der Shadow Map sein müsste, falls er aus der Perspektive der Lichtquelle nicht verdeckt ist. An diesen Texturkoordinaten kann nun in der Shadow Map nachgeschaut werden, ob der dort gespeicherte Tiefenwert dem Tiefenwert des Punkts entspricht. Ist dies der Fall, handelt es sich um den selben Punkt, d.h. dieser ist nicht verdeckt und liegt demnach nicht im Schatten. Falls nein, handelt es sich in der Shadow Map um einen anderen Punkt, der den gerade verarbeiteten Punkt (aus der Perspektive der Lichtquelle) verdeckt.

1. Implementieren Sie zunächst die lokale Beleuchtungstechnik *Blinn-Phong*, indem sie die in `final_frag.glsl` gekennzeichneten Methoden vervollständigen. Beachten Sie, dass es sich bei der Lichtquelle um Sonnenlicht handelt, welche als paralleles Licht approximiert werden kann. Verwenden Sie sinnvolle Konstanten.

   *Hinweis: Die Blickrichtung lässt sich direkt aus der View-Matrix auslesen.*

2. Da die Shadow-Map in einem eigenen Durchlauf gerendert wird, gibt es hierfür eigene Vertex- und Fragment-Shader. Der Fragment-Shader `shadowmap_frag.glsl` soll die $z$-Komponente der normalisierten Gerätekoordinaten ausgeben, da diese in der Shadow-Map-Textur gespeichert werden soll. Implementieren Sie hierfür die Funktion `CalcDepth`, die ebendiesen Wert zurückliefern soll. Skalieren Sie dazu `gl_FragCoord.z` auf das richtige Interval.

   Die restliche Berechnung der Shadow-Mapping-Technik findet in `final_frag.glsl` statt, der Fragment-Shader des Renderdurchlaufs, in dem die Geometrie gerendert wird. Vervollständigen Sie dort die Implementierung des naiven Shadow-Mappings, indem Sie die Methode `CalcShadow` implementieren. Diese bekommt bereits einen Punkt in den normalisierten Gerätekoordinaten bezüglich der Lichtquelle übergeben, d.h. dieser Punkt wurde bereits rekonstruiert. Die Methode soll abhängig davon, ob der Punkt im Schatten liegt, oder nicht, eine 0 (für im Schatten) oder eine 1 (für im Licht) zurückgeben. Greifen Sie um das herauszufinden an der richtigen Stelle auf die Textur `tex_shadowmap` zu. Dieser Zugriff liefert (wie alle Texturzugriffe in GLSL) einen `vec4` zurück, wobei davon nur die $x$-Koordinate mit der Tiefe initialisiert ist, die Sie in Teilaufgabe 2 berechnet haben. Beim Anschließenden Vergleich sollten Sie auf einen der beiden Werte einen passenden Bias addieren, um den *Shadow-Acne* Effekt zu verringern.

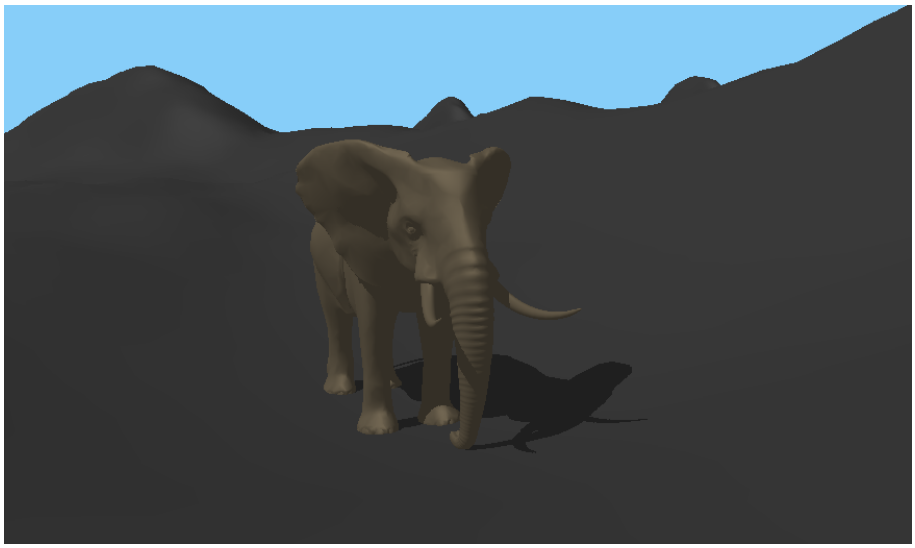

Abbildung 3: Ausgabe nach erforgreicher Bearbeitung von Aufgabe 2.2.

3. In der Implementierung wird ausschließlich ein kleiner Bereich um den Elephanten für die Shadow Map evaluiert. Beantworten Sie folgende Fragen schriftlich:

   - Wieso ist es nicht praktikabel mit der verwendeten Shadow Map das gesamte Terrain abzudecken?
   - Wie kann man dennnoch visuell ansprechende Schatten sinnvoll in Echtzeit auf dem gesamten Terrain erzeugen?

4. Durch die Auflösung der Shadow Map kommt es an den Rändern der Schatten zu einem Aliasing-Effekt (vgl. Abbildung 4 links). Um diesen Effekt zu verringern, bietet sich der Einsatz eines sogenannten *Percentage-Closer Filters (PCF)* an. Bei dieser Technik evaluiert man die Shadow-Map-Werte in der Nachbarschaft der betrachteten Position und mittelt das Ergebnis, um Übergänge zu Schatten weicher darzustellen. Implementieren Sie einen einfachen PCF in der Methode `CalcShadowPCF` in `final_frag.glsl`. Achten Sie dabei auf einen sinnvollen Tradeoff von Sample-Anzahl und Qualität. Solange die Taste 3 gedrückt ist, wird die Methode `CalcShadowPCF` statt `CalcShadow` zur Schattenberechnung verwendet.

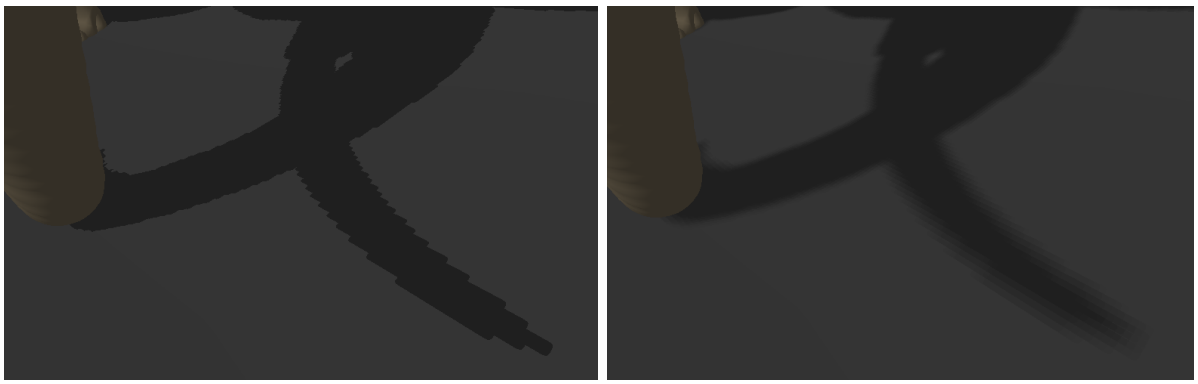

Abbildung 4: Einfache Shadow Maps (links) und Shadow Maps mit PCF (rechts).