Aufgabe 2: Displacement

(Neue) Techniken:

Normal-Mapping, Parallax-Mapping, Displacement, Tangent-Space, Geometrie & Tessellation-Shader

Beschreibung:

In dieser Aufgabe sollen unterschiedliche Displacement Mapping Verfahren umgesetzt werden.
Als Einstieg soll das Pixel-basierte Normal Mapping und Parallax-Mapping implementiert werden, das anschließend mit Unterstützung der Tessellation-Funktionalität von OpenGL um eine Vertex-basierte Komponente erweitert wird. Der Fokus dieser Aufgabe liegt auf der Meso-Struktur der Oberfläche. Für die Beleuchtung soll weiterhin das Phong-Modell eingesetzt werden.

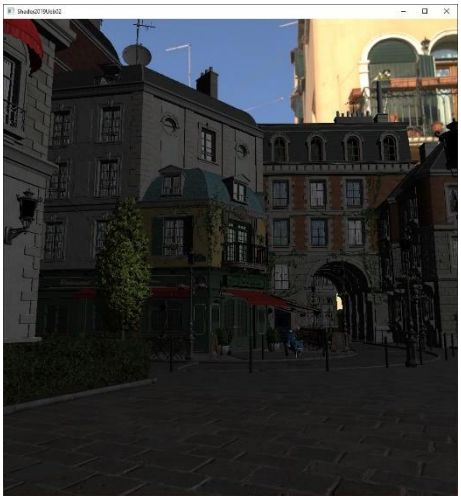

Figure 1 Scene Render A2

Es soll die in Aufgabe 1 angefertigte Lösung weiterverwendet werden.

Brick-Textur

Figure 5 Keine Technik (Phong Shading & Geo Normal)

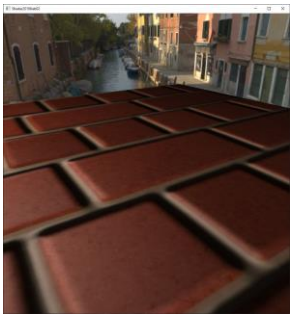

Figure 4 Normal Mapping

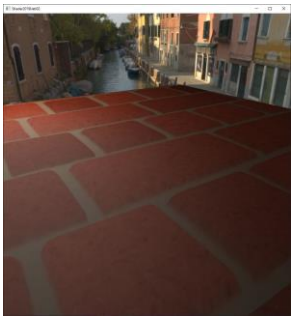

Figure 3 Parallax Mapping

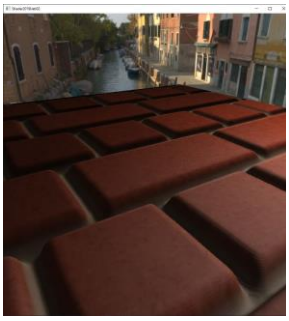

Figure 2 Normal & Parallax Mapping

Für die Bearbeitung dieser Aufgabe sollte OpenGL 4.0 und GLSL 4.0 (oder höher) verwendet werden. Tessellation ist erst ab 4.0 im Core von OpenGL.

Kernanforderung:

- Die Tangent Spaces für die Vertices werden korrekt berechnet (beim Modell geladen). Die zweite Achse (Tangente oder Bitangente) wird gespeichert und die dritte wird im Shader mit dem Kreuzprodukt berechnet.

- Die resultierende Normale wird anstelle der Geometrienormale (Normale der Makro-Struktur) für die Beleuchtungsberechnung verwendet.
- Es findet Geometrie Tessellation statt.

Weitere Funktionalitätsanforderungen:

- Für die Fläche soll der Tangent-Space korrekt berechnet werden. Für das Modell kann das Assimp-Flag gesetzt werden und der Modelloader um das einlesen der Tangenten (srcMesh->mTangents[i]) des Mesh ergänzt werden.
- Es soll korrektes Normalmapping und *(Für das Grid)* Parallax-Mapping umgesetzt werden. Beide Techniken soll mittels Tastendruckes jeweils umschaltbar sein.
  Für das Parallax-Mapping soll die Technik Steep Parallax Mapping umgesetzt werden, da diese guten optischen Resultate liefert trotz stärkerer Rechenbelastung. Die Anzahl der Sample Ebenen soll abhängig vom Blickwinkel sein.
- Es wird eine Punktlichtquelle ergänzt, die um die Szene bzw. Grundflache kreist (nahtloses Pausieren ist gefordert und ist hilfreich für optisches Debugging). Diese soll korrekt mit Abschwächung implementiert werden. (Siehe CG1).
- Ein rudimentärer Tessellation Evaluation Shader ist implementiert und die Tessellation Level sind so eingestellt, dass eine Unterteilung der Patches stattfindet.
- *(Für das Grid)* Der Tessellation Evaluation Shader wird verwendet, um die Positionen der Vertices entsprechend der übergebenen Displacement Map anzupassen. Dabei werden die Vertexpositionen entlang der Normale (nicht die Normale aus der Normal Map, sondern die Geometrienormale) verschoben. Die Normale für die Beleuchtung wird weiterhin durch das Normal Mapping ermittelt.
- Der **Tessellation Control Shader** steuert die Tessellation Level abhängig von der *Entfernung* und *Größe der Patches*. Zusätzlich können weitere Parameter wie der Winkel der Oberfläche zum Betrachter einfließen. An den Patch-Kanten (zwischen den beiden Dreiecken) entstehen keine Löcher, d.h. für die äußeren Tessellation-Level wird für beide Kanten der Patches der gleiche Wert berechnet.
- Die **Tangenten, Bitangenten und Normalen** werden mit einem **Geometry Shader** angezeigt. Um die verschiedenen Vektoren unterscheiden zu können, werden diese in unterschiedlichen Farben dargestellt. (Tangente = Rot, Bitangent=Grün, Normale=Blau)
  Die Bitangente soll im Shader berechnet werden, sodass lediglich die Normale und Tangente als Attribute an den Shader übergeben werden müssen!
    - **Geometry Instancing** wird genutzt, um die zusätzliche Geometrie im Geometry Shader parallel zu erzeugen. Dabei werden *keine* Verzweigungsanweisungen oder Bedingungsoperatoren verwendet.
- Die Lichtquelle soll visualisiert werden in der entsprechenden Lichtquellenfarbe. Dafür kann entweder ein Würfel oder eine Kugel gerendert. Diese Geometrien sollen wiederum keine Beleuchtung besitzen (Da diese Licht-Emitter sind).

- In der Shading Stage „Normalmap“ soll diese nicht im Tangent-Space sondern konvertiert in den World-/Model-Space angezeigt werden.

Stone-Textur

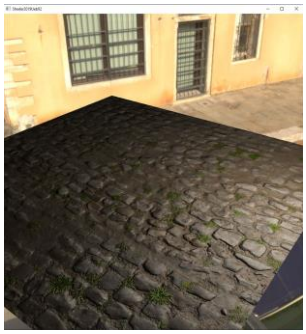

Figure 6 Grid Render

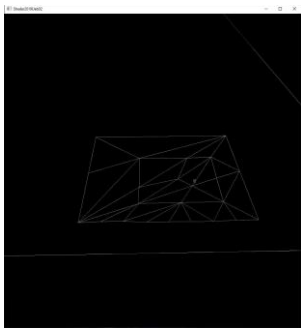

Figure 8 Tessellation Far

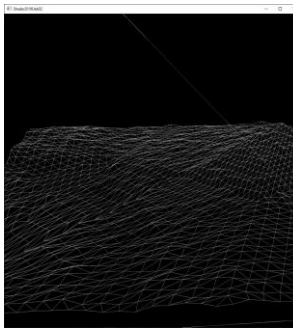

Figure 7 Tessellation Near

Szene

Figure 9 Scene Render

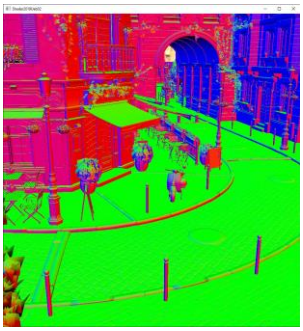

Figure 10 Normalmap im World Space (Tangent to World Space Transformation)

Figure 11 Tessellation mit Faktor durch Abstand und Patch Größe

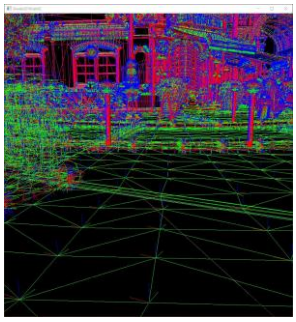

Figure 12 Anzeige des Tangentspaces mithilfe des Geometrie Shader und Geometrie Instancing

Tastatur-Belegung (Vorschlag):

Taste	Funktion
Esc	Beenden des Programms
F1	Toggle Fullscreen
F2	Toggle Tangetspace
F3	Toggle Wireframe
F10	Reload Shader
1	Toggle Normalmapping
2	Toggle Parallaxmapping

3	Toggle Tessellation
H/h	Toggle Hilfe
F/f	Toggle FPS
G/g	Toggle Grid/Szene
L/l	Toggle Licht Visualisierung
N/n	Grid-Auflösung erhöhen
M/m	Grid-Auflösung verringern
U/u	Der Versatz der Vertices entlang der Normale wird erhöht.
I/i	Der Versatz der Vertices entlang der Normale wird verringert.
P/p	Zeitabhängige Animation pausieren
S/s	Shading Umschalten
STRG + LMB	Kamera Rotation
STRG + MMB	Pan Kamera
STRG + RMB (ggf. Maus Wheel)	Zoom Kamera (ggf. Versetzung entlang Orientierung)

Shader (Vorschlag):

Shader (Komponenten)	Beschreibung
LightVisShader *(Vertex & Fragment)*	Zeichnet die Punkt-Lichtquelle als Primitive Geometrie (Cube oder Sphere)
ModelShader *(Vertex & TessellationControl & TessellationEvaluate & Fragment)*	Zeichnet das Modell bzw. das Grid mit den umschaltbaren Techniken Normalmapping, Parallaxmapping und Tessellation mit Displacement.
SkyShader *(Vertex & Fragment)* (Aufg 1)	Zeichnet die Skymap im Hintergrund
TangentShader *(Vertex & Geometrie & Fragment)*	Zeichnet die 3 Achsen des Tangent Spaces, es ist darauf zu beachten, dieselbe Berechnung wie beim ModelShader zu nutzen.
TextShader *(Vertex & Fragment)* (Aufg 1)	Kann Text auf dem Display ausgeben

Tipps:

1) Die Scene besitzt keine Höhemap, aus diesem Grund müssen die davon Abhängigen Techniken (Vertex Displacement & Parallax Mapping) nicht angewendet werden.
2) Es ist im Rahmen dieser Übung erlaubt bedingte Switches (*if else*) im Shader durchzuführen, obwohl dies nicht so effizient ist als verschiedene Shader zu bauen und bei Bedarf zu nutzen.

3) Hochauflösende Texturen (incl. Normal und Displacement-Maps) können hier [1] gefunden werden.
4) Achtet bei allen Matrix Multiplikationen auf die Reihenfolge! Die erste Transformation steht hinten (von rechts nach links). Auch bei einer Multiplikation mit einem Vektor muss die Reihenfolge beachtet werden. v*M führt *nicht* zu einem Compilerfehler, da GLSL *nicht* zwischen Zeilen und Spaltenvektoren unterscheidet.
5) Normal Maps werden unter Umständen so *gebaked (erstellt)*, dass der Nullpunkt nicht unten links, sondern oben links liegt. Berücksichtigt dies bei der Dekodierung der Normalen oder beim Transformieren aus dem Tangent Space, indem Ihr eine Achse invertiert.
6) Denkt daran, die Variable max_vertices im Geometry Shader zu erhöhen, *falls* Ihr mehrere Linien zeichnen wollt. Im Zusammenhang mit Geometry Instancing gibt max_vertices die Anzahl der emittierbaren Vertices pro Instanz an.
7) Wenn die Tessellation Level den Wert 0 haben, wird der Patch verworfen. Setzt diese am besten auf einen Wert >= 1 bevor Ihr anfangt, den Tessellation Evaluation Shader zu implementieren.
8) Lasst Euch nicht von den verschiedenen Patch-Arten (Input, Ouput, Abstract) verwirren. Wir wollen uns auf die Grundlagen konzentrieren. Deswegen haben in dieser Aufgabe alle Patches 3 Vertices.
9) Für die Betrachter anhängige Tessellation kann die Funktion smoothstep nützlich sein. Diese kann auch mit einem höheren Wert für edge0 als für edge1 aufgerufen werden. Ihr könnt jedoch auch beliebige andere Interpolationen wählen.
10) Um den Abstand zur Kamera für die Betrachter abhängige Tessellation zu berechnen, benötigt Ihr die Kameraposition im Weltkoordinatensystem. Diese könnt Ihr berechnen, indem Ihr die inverse View-Matrix mit dem Nullpunkt (0, 0, 0, 1) multipliziert. Matrixinvertierungen werden im Shader vermieden. Sie können im Host vorberechnet werden, um eine höhere Performance zu erreichen. Im Praktikum müsst Ihr auf solche Feinheiten nur bedingt Rücksicht nehmen. Wählt den übersichtlicheren Weg, solange sich dies nicht stark negativ auf die Performanz des Programmes auswirkt.
11) Wenn Ihr die Betrachter abhängige Tessellation implementiert, kommt es zu einem Flackern der Geometrie. Hierfür gibt es Lösungen, die jedoch zusätzlichen Implementierungsaufwand mit sich ziehen. Um dieses Problem zumindest etwas weniger auffällig zu machen, könnt Ihr die Unterteilungsstrategie auf fractional_odd_spacing oder fractional_even_spacing setzen oder die Stärke des Versatzes entlang der Normale vom Abstand abhängig machen. Dies ist jedoch nicht gefordert.
12) Ihr könnt Euch als Hilfe eine Shading Stage im Fragment-Shader ergänzen für Displacement Maps. Achtet darauf ob ihr Height- oder Depthmaps ladet!
13) Eine Normale bzw. Tangente ist ein Einheitsvektor, zur Optimierung kann jeweils die 3. Komponente auf der GPU berechnet werden.
14) Folgende Beispiel können behilflich sein:

    Geometry Shader (zip auf HP), demoTessellation

Weitere (freiwillige) Funktionalitäten:

1) Für die Szene können aus den Normalmaps bzw. Diffuse Maps mit vorheriger Filterung Heightmaps erstellt werden, sodass das Parallax Mapping und das Vertex Displacement im Tessellation Evaluation Shader auch für die Szene durchgeführt werden können. Es kann auch probiert werden, diese zur Laufzeit beim Laden der Texturen zu generieren.
2) Es können verschiedene Arten von Parallax Mapping implementiert werden z.B.
   Reliefe Parallax Mapping & Parallax Occlusion Mapping. Beide könne die Szene optisch verbessern.

[1] https://texturehaven.com/