Media Informatics /
Graphics & Geometric Computing

Prof. Dr. Marcel Campen
Axel Schaffland, M.Sc.

UNIVERSITÄT OSNABRÜCK

Computergrafik (SS 2018)

Übung 6

fällig Montag **28. Mai**, 14:20

- Geben Sie bei jeder Abgabe alle Ihre Matrikelnummern auf jedem Blatt an.
- Verspätet eingereichte Abgaben können nicht gewertet werden.

Aufgabe 1 (Theorie: WebGL)

(a) (4P) Variablen in GLSL-Shadercode können verschiedener Art sein (siehe Matrix Kopfzeile). Kreuzen Sie in der Matrix jedes Feld an, für das die Aussage der jeweiligen Zeile auf die Variablenart der jeweiligen Spalte zutrifft:

	attribute	uniform	const	varying
Kann im Vertex Shader gelesen werden				
Kann im Vertex Shader geschrieben werden				
Kann im Hauptprogramm geschrieben werden				
Kann im Fragment Shader gelesen werden				
Kann im Fragment Shader geschrieben werden				
Kann im Vertex Shader innerhalb einer Funktion definiert werden				
Kann im Fragment Shader innerhalb einer Funktion definiert werden				
Hat im Vertex Shader grundsätzlich für jeden Vertex den gleichen Wert				
Hat im Fragment Shader grundsätzlich für jedes Fragment den gleichen Wert				

(b) (1P) Ergänzen Sie den Lückentext:

Die finale Position eines Vertex muss im _____ Shader in die eingebaute Variable _____ geschrieben werden.

Die finale Farbe eines Fragments muss im _____ Shader in die eingebaute Variable _____ geschrieben werden.

(c) (1P) Markieren Sie die richtige(n) Antwort(en):
Zwischen Vertex Shader und Fragment Shader findet grundsätzlich statt:

- ☐ ModelView-Transformation
- ☐ Rasterisierung
- ☐ Projektion
- ☐ z-Buffer-Test

(d) (1P) Beschreiben Sie eine mögliche 3D-Szene (in abstrakten Worten; keine numerischen Angaben nötig), die dazu führt, dass beim Rendern eines einzelnen Bildes mit 1 Mio. Pixeln mittels OpenGL der Fragment Shader mehr als 1 Mio. mal ausgeführt wird.

(e) (3P) Markieren und erläutern Sie alle sechs Fehler im folgenden GLSL Code:

```
vec4 v = vec4(1.0, 1.0, 0.0, 0.0);
vec3 w = vec3(2.0, 3.0, 4.0);
vec3 v1 = v.xxx;
vec3 v2 = v.xyzw;
float f = v[1];
vec3 v3 = v.def;
vec3 v4 = v.gb;
vec3 d = dot(w, v);
vec3 e = dot(w.xy, v.yz);
mat2 m = mat2(1.0, 2.0, 1.0, 0.0, 0.1);
v[0] = f;
```

1.

2.

3.

4.

5.

6.

Aufgabe 2 (Praxis: WebGL)

Schauen Sie sich den Code für Übungsblatt 6 an.

(a) (3P) Übergabe von Daten an die GPU:

Ergänzen Sie die Lücke bei `AUFGABE 2a` in der Funktion `initBuffer` derart, dass der darzustellende Würfel in Vertex-Buffer-Objects gespeichert wird.

Die 3D-Positionen der Vertices des Würfels sind im Array `positions` gespeichert, deren Farben in `colors`, die Vertex-Normalen in `vertexNormals`. Die Indizes der Vertices der Dreiecke liegen im Array `indices` vor.

Erstellen Sie die vier GL-Buffer `positionBuffer`, `normalBuffer`, `colorBuffer` und `indexBuffer` mit den entsprechenden Einträgen. Verwenden Sie dazu die WebGL Funktionen `createBuffer`, `bindBuffer` und `bufferData`.

Schauen Sie sich dazu auch den Code ab `MARKER 1` an, da dort definiert wird, wie WebGL den Inhalt der Buffer in die Attribute einlesen soll. Achten Sie darauf, dass Ihr Code mit diesem kompatibel ist (z.B. in Sachen Datentypen).

(b) (3P) Vertex Shader:

Ergänzen Sie die Lücke bei `AUFGABE 2b` im Vertexshader-Code wie folgt. Verwenden Sie dabei die OpenGL Shading Language (GLSL).

Berechnen und setzen Sie die finale projizierte Position des Vertex, basierend auf der Eingabe `aVertexPosition`; die nötigen Matrizen stehen als Uniforms zur Verfügung.

Im Fragment Shader soll anschließend Phong Lighting mit Phong Shading berechnet werden. Dazu benötigt der Fragment Shader eine Position (`vPositon`) und eine Normale (`vNormal`) sowie eine Materialfarbe (`vColor`). Sorgen Sie im Vertex Shader Code dafür, dass dem Fragment Shader diese Informationen (die Sie den Attributen entnemhen können) in den entsprechenden Variablen zur Verfügung steht. Achten Sie dabei darauf, dass Position und Normale in Kamera/View-Koordinaten weitergegeben werden, und dass die Normale Länge 1 hat.

Wenn Sie die Datei nun öffnen, sollte der rotierende Würfel bereits in Tiefschwarz erkennbar sein.

Hinweis: Beachten Sie immer die Dimensionalität der Datentypen (d.h. insbesondere `vec3` vs. `vec4`) und konvertieren Sie wo nötig (z.B. per Multi-Zugriff oder Swizzling, bzw. per `vec4(vec3(1,1,1), 1)`).

(c) (4P) Fragment Shader:

Ergänzen Sie die Lücke bei `AUFGABE 2c` im Fragmentshader-Code wie folgt.

Werten Sie das Phong Beleuchtungsmodell (unter Verwendung von Phong Shading) aus. Die Parameter für das Beleuchtungsmodell sind in den Uniforms `uCAmbient`, `uCDiffuse`, `uCSpecular` und `uShine` hinterlegt. Kamera- und Lichtposition liegen (in Kamera/View-Koordinaten) ebenfalls in Uniforms bereit.

Die Koeffizienten C_a bzw. C_d des Phong-Modells (in Vektor-Schreibweise) erhalten Sie, indem Sie `uCAmbient` bzw. `uCDiffuse` mit der Materialfarbe multiplizieren. Den Koeffizienten C_s erhalten Sie, indem Sie `uCSpecular` mit $(1, 1, 1)$ multiplizieren.

Setzen Sie die Farbe des Fragments anhand des Ergebnisses der Beleuchtungsbrechnung.

Aufgabe 3 (Bonus: Komplexere Beleuchtungsmodelle)

Ersetzen Sie das Phong Beleuchtungsmodell durch ein komplexeres Beleuchtungsmodell Ihrer Wahl. Dies können Beispielsweise das Cook-Torrance-Beleuchtungsmodell oder das Oren-Nayar-Modell sein. Ersetzen Sie den Würfel durch die Kugel aus der vorigen Übung (mit geeigneten Vertex-Normalen), damit sich die Auswirkung des Modells besser erkennen lässt.

Belassen Sie dabei den ursprünglichen Phong-Shader zwecks Bewertung als Kommentar im Code.