Media Informatics /
Graphics & Geometric Computing

Prof. Dr. Marcel Campen
Steffen Hinderink

UNIVERSITÄT OSNABRÜCK

Computergrafik (SS 2020)

Blatt 6

fällig Sonntag **7. Juni**, 24:00

Verspätet eingereichte Abgaben können nicht berücksichtigt werden.

Aufgabe 1 (Theorie: Shader)

(a) (3P) Variablen in GLSL-Shadercode können verschiedener Art sein (siehe Kopfzeile der Tabelle unten). Kreuzen Sie in der Tabelle jedes Feld an, für das die Aussage der jeweiligen Zeile auf die Variablenart der jeweiligen Spalte zutrifft:

	attribute	uniform	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 des selben Dreiecks 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) Kreuzen Sie die richtige(n) Antwort(en) an:
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) (2P) Markieren und erläutern Sie mindestens sechs der Fehler, die sich im folgenden GLSL Code verstecken:

```
1  vec4 v = vec4(1.0, 1.0, 0.0, 0.0);
2  vec3 w = vec3(2.0, 4.0, 0.0);
3  vec3 v1 = v.xxx;
4  vec3 v2 = v.xyzw;
5  float f = v[1];
6  vec3 v3 = v.gb;
7  vec3 v4 = v.def;
8  vec3 d = dot(w.xy, v.yz);
9  vec3 e = dot(w, v);
10 vec3 r = cross(w, w);
11 mat2 m = mat2(1.0, 2.0, 1.0, 0.0, 0.1);
12 v[0] = f;
```

(f) (2P) Nehmen Sie Folgendes an:

- Sie übergeben drei Vertices und ein einzelnes Dreieck (welches von diesen drei Vertices aufgespannt wird) an OpenGL.
- Der Color-Buffer wurde zuvor komplett auf schwarz gesetzt, der Depth-Buffer ist im Initialzustand.
- Der Vertex Shader wird für die Vertices ausgeführt; dabei ergeben sich für die Variable `gl_Position` die Werte $(0, 0, 0, 1)$, $(1, 1, 0, 1)$, und $(1, -1, 0, 1)$.
- Die main-Funktion des Fragment Shader beinhaltet nur eine Anweisung: `gl_FragColor = vec4(0, 0, 1, 1)`.

Beschreiben Sie präzise, wie das resultierende Bild aussieht. Geben Sie insbesondere an, wie viel Prozent der Bildfläche am Ende nicht mehr schwarz sind.

Aufgabe 2 (Praxis: Shader)

(a) (2P) Übergabe von Daten an die GPU:

Ergänzen Sie die Funktion `createBuffers`, die die Puffer für das Objekt erzeugt. Die Positionen der Vertices befinden sich in `positions`, die Vertex-Normalen in `normals` und die Farben in `colors`. Die Vertex-Indizes der Dreiecke liegen im Array `indices` vor. Erstellen Sie die vier GL-Puffer `positionBuffer`, `normalBuffer`, `colorBuffer` und `indexBuffer` mit den entsprechenden Einträgen. Verwenden Sie dazu die WebGL Funktionen `createBuffer`, `bindBuffer` und `bufferData`. Achten Sie darauf, dass ihr Code kompatibel zu dem Code ist, der die Puffer einliest (ab `Label a`).

(b) (4P) Vertex Shader:

Programmieren Sie den Vertex Shader in der OpenGL Shading Language (GLSL). Berechnen und setzen Sie die finale projizierte Position des Vertex basierend auf der Eingabe `aPosition`; die nötigen Matrizen stehen als Uniforms zur Verfügung.

Der Fragment Shader soll später Phong Lighting mit Phong Shading berechnen können. Dazu benötigt der Fragment Shader eine Position (`vPosition`), eine Normale (`vNormal`) sowie eine Materialfarbe (`vColor`). Sorgen Sie im Vertex Shader Code dafür, dass dem Fragment Shader diese Informationen (die Sie den Attributen entnehmen können) in den entsprechenden Variablen zur Verfügung stehen. Achten Sie dabei darauf, dass Position und Normale in Kamerakoordinaten weitergegeben werden, und dass die Normale Länge 1 hat.

Wenn Sie die Datei nun öffnen, sollte die Kugel bereits komplett schwarz angezeigt werden.

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:

Programmieren Sie den Fragment Shader. Werten Sie das Phong Beleuchtungsmodell ohne Blinn Approximation (unter Verwendung von Phong Shading) aus. In den Uniforms `uLightPos`, `uAmbient`, `uDiffuse`, `uSpecular` und `uShininess` befinden sich die Parameter, die durch die Slider verändert werden können. Die Lichtposition ist dabei bereits in Kamerakoordinaten. Die Lichtfarbe befindet sich im Uniform `uLightColor`. Die Koeffizienten C_a bzw. C_d des Phong-Modells (in Vektor-Schreibweise) erhalten Sie, indem Sie `uAmbient` bzw. `uDiffuse` mit der Materialfarbe multiplizieren. Den Koeffizienten C_s erhalten Sie, indem Sie `uSpecular` mit $(1,1,1)$ multiplizieren.

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

Aufgabe 3 (Bonus: Anderes Lighting / Shading)

Ersetzen Sie das Phong Beleuchtungsmodell und/oder das Phong Shading durch anderes Lighting und/oder Shading Ihrer Wahl. Sie können beispielsweise das Cook-Torrance-Beleuchtungsmodell, das Oren-Nayar-Modell oder Cel Shading implementieren. Wenn Sie dazu weitere Parameter benötigen, können Sie zusätzliche Slider hinzufügen. Mit Hilfe des Uniforms `uBonus` soll zwischen Phong Lighting und Ihrem weiteren Beleuchtungsmodell gewechselt werden können.