Computergrafik Praktikum 6

Allgemeines zum Praktikum

Im Rahmen des Computergrafik-Praktikums lernen Sie, die aus der Vorlesung gewonnenen Erkenntnisse praktisch mit der Programmiersprache C/C++ umzusetzen.

Bei der Konzipierung der Praktika wurde darauf geachtet, dass die verwendeten Bibliotheken sowohl für Windows als auch für OS X erhältlich sind. Sofern Sie die Aufgaben im Medienlabor bearbeiten, steht Ihnen die Entwicklungsumgebung XCode zur Verfügung, mit der Sie Ihre C++-Programme entwickeln können. Wenn Sie die Aufgaben an Ihrem privaten Rechner unter Windows bearbeiten möchten, empfehle ich Ihnen die Entwicklungsumgebung Visual Studio 2013, die Sie als Student/-in über das Microsoft ELMS-Programm kostenlos herunterladen können.

Bitte beachten Sie bei der Bearbeitung der Praktikumsaufgaben, dass Ihre Lösungen für folgende Praktikumsaufgaben weiter genutzt werden müssen. Die Praktikumsarbeiten bauen also ineinander auf. Bearbeiten Sie deshalb die Lösungen sorgfältig und löschen Sie diese nicht, wenn die Aufgabe abgenommen wurde. Achten Sie bei der Entwicklung darauf, dass nur die Schnittstellen in den Header-Dateien stehen, während die Implementierung ausschließlich in den cpp-Dateien steht.

Zum Bestehen des kompletten Praktikums müssen Sie jede Praktikumsaufgabe erfolgreich abschließen und mindestens 80% der Gesamtpunkte für das komplette Praktikum erhalten. Wie viele Punkte Sie für eine einzelne Aufgabe erhalten, hängt von Ihrer Implementierung und Ihren Erläuterungen ab.

Um dieses Praktikum zu bestehen, müssen Ihre entwickelten Teillösungen lauffähig sein (5 Punkte) und Sie müssen die einzelnen Lösungen erläutern können (weitere 5 Punkte). Insgesamt können Sie bei diesem Aufgabenblatt 10 Punkte + 5 Zusatzpunkte erhalten.
Abnahmetermin: 21.06. (Di) & 24.06. (Fr)

Thema Praktikum 6

Bei den vorliegenden Praktikumsaufgaben sollen Sie lernen, wie Sie Shader-Programme entwickeln und auf der Grafikkarte ausführen können. Das Schreiben von eigenen Shader-Programmen erlaubt Ihnen, Ihre Anwendung um neue Darstellungsalgorithmen zu erweitern. In diesem Zusammenhang erlernen Sie den sicheren Umgang mit der OpenGL-typischen Programmiersprache GLSL. Nach der Bearbeitung der Praktikumsaufgaben sollten Sie in der Lage sein, eigene Shader-Programme zu konzipieren, zu implementieren und zu testen.

Für das Praktikum steht Ihnen wieder ein XCode- bzw. Visual-Studio-Projekt zur Verfügung (cgprakt6.zip). Sie müssen in dem jeweiligen Projekt die folgenden Dateien durch Ihre eigenen Implementierungen ersetzen:

- vector.h & .cpp

- color.h & .cpp
- rgbimage.h & .cpp
- model.h & .cpp
- triangleboxmodel.h & .cpp
- lineboxmodel.h & .cpp

Hinweis: Das Projekt enthält eine Klasse zum Zeichnen einer Shadow-Map (ShadowMapGenerator). Diese Klasse zeichnet Modelle (der Model-Klasse) mit einem alternativen Shader, was dazu führt, dass die folgende Warnung beim Rendern der Modelle ausgegeben wird (*Model::applyMaterial()*):

```
if(!pPhong) {
  std::cout << "Model::applyMaterial(): WARNING Invalid shader-type. Please apply PhongShader for rendering models.\n";
      return;
  }
```

Entfernen Sie bitte die Ausgabe in der *Model::applyMaterial()*-Methode, so dass dort nur noch

```
if(!pPhong) {
        return;
}
```

steht, ansonsten ruckelt Ihre Anwendung unnötig.

Aufgabe 1 (2 Punkte)

Die Shader-Dateien „assets/vsphong.glsl“ und „assets/fsphong.glsl“ werden von der *PhongShader*-Klasse genutzt, um Modelle nach dem Phong-Reflexionsmodell anzuzeigen. Verändern Sie den Fragment-Shader so, dass für glänzende Reflexionen die Methode nach Blinn genutzt wird. Die Berechnung des Halbvektors wird in den Vorlesungsfolien leider fehlerhaft beschrieben. Der Fehler in der Berechnung sollte Ihnen leicht auffallen. Korrigieren Sie den Fehler für Ihre Implementation.

Aufgabe 2 (4 Punkte)

Die Implementierung des Phong-Fragment-Shaders (fsphong.glsl) unterstützt leider nur eine Punktlichtquelle. Dies ist für die meisten Anwendungsfälle zu wenig und deshalb soll der Shader so erweitert werden, dass mehrere Lichtquellen unterschiedlichen Typs unterstützt werden (direktionale, Punkt- und Spot-Lichter, siehe Vorlesung 2). Hierfür wurden in den Dateien Lights.h & .cpp bereits die unterschiedlichen Lichttypen als Klassen implementiert. Damit nicht jeder einzelnen Shader-Implementierung alle Lichtquellen separat übergeben werden müssen, wurde die Singleton-Klasse *ShaderLightMapper* entwickelt. Die Klasse verwendet einen so genannten „Uniform Block“, um Speicher auf der Grafikkarte für eine „globale“ Uniform-Struktur anzulegen. In dieser Struktur befindet sich die Beschreibung der Lichtquellen-Daten:

```
const int MAX_LIGHTS=14;
struct Light
{
    int Type;
    vec3 Color;
    vec3 Position;
    vec3 Direction;
    vec3 Attenuation;
    vec3 SpotRadius;
    int ShadowIndex;
};

uniform Lights
{
    int LightCount;
    Light lights[MAX_LIGHTS];
};
```

(Sie finden die Struktur ebenfalls in der Datei „assets/fslightdummy.glsl“.)

Wird die obige Struktur in einem Shader deklariert, so sorgt die *BaseShader*-Klasse dafür, dass die Daten der Lichtquellen-Objekte aus der C++-Anwendung in diese Struktur übertragen werden. Dies geschieht mit Hilfe der *ShaderLightMapper*-Klasse, die das Mapping der Lichtquellen-Objekte zum Shader übernimmt. Hierfür müssen die Lichter, die in der Szene verwendet werden sollen, der *ShaderLightMapper*-Klasse bekanntgegeben werden. Zum Beispiel wie folgt:

```
DirectionalLight* dl = new DirectionalLight();
dl->direction(Vector(0.2f, -1, 1));
dl->color(Color(0.25, 0.25, 0.5));
dl->castShadows(true);
ShaderLightMapper::instance().addLight(dl);
```

Folgende Daten werden durch die Struktur beschrieben:

- **LightCount** – Anzahl der aktiven Lichtquellen.
- **lights[idx].Type** – Typ der Lichtquelle (0=POINT, 1=DIRECTIONAL, 2=SPOT).
- **lights[idx].Color** – Farbe der Lichtquelle in RGB.
- **lights[idx].Position** – Position der Lichtquelle (für DIRECTIONAL uninteressant).
- **lights[idx].Direction** – Ausrichtung der Lichtquelle (für POINT uninteressant).
- **lights[idx].Attenuation** – Koeffizienten für Intensitätsabfall über die Entfernung. Der Intensitätsabfall soll wie folgt berechnet werden: *Intensity=1.0/(Attenuation.x + Attenuation.y*Dist + Attenuation.z*Dist*Dist)*. *Dist* bezeichnet hier den Abstand des Fragments zur Lichtquelle. DIRECTIONAL-Lichtquellen dürfen keinen Intensitätsabfall über die Entfernung aufweisen.
- **lights[idx].SpotRadius** – SpotRadius.x enthält den inneren und SpotRadius.y den äußeren Radius der Spot-Lichtquelle (in Radien umgerechnet).
- **lights[idx].ShadowIndex** – Nur für die Zusatzaufgabe von Bedeutung.

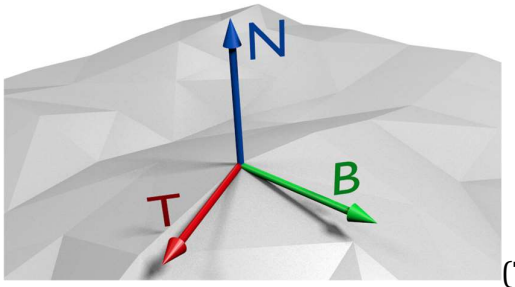
Erweitern Sie den Phong-Fragment-Shader so, dass bis zu 14 Lichtquellen unterschiedlichen Typs korrekt angewendet werden. Hierzu wurde in der *Application*-Klasse eine Beispiel-Szene implementiert, die Sie über die Methode *createScene()* erstellen können. Wenn Ihre Implementierung korrekt ist, sollte die Szene wie folgt dargestellt werden:

(die Szene verwendet 13 Lichtquellen: 1 direktionale Lichtquelle, 6 Punkt-Lichter & 6 Spot-Lichter)

Hinweis: Für das Mapping der Lichtquellen zum Uniform-Block wurde eine entsprechende Struktur mit passendem Alignment in der *ShaderLightMapper*-Klasse deklariert. Die Klasse wurde mit unterschiedlichen Grafikkarten getestet, dennoch ist es möglich, dass Ihr Grafikkarten-Treiber ein anderes Alignment nutzt. Falls dies der Fall ist, wird die Zusicherung „*assert(sizeof(ShaderLightBlock) == BlockSize)*“ im Konstruktor der *ShaderLightMapper*-Klasse fehlschlagen. Für diesen Fall müssen Sie die Struktur *ShaderLightMapper::ShaderLight* und *ShaderLightMapper::ShaderLightBlock* so anpassen, dass deren Größen zur Größe des kompilierten GPU-Blocks passen.

Aufgabe 3 (4 Punkte)

Gegenwärtig werden Texturen vom Phong-Shader nur genutzt, um die diffuse Farbe der Oberfläche zu steuern. In dieser Aufgabe sollen Sie den Phong-Shader so erweitern, dass dieser die Verwendung von Normal-Maps unterstützt. In Normal-Maps werden die Normalen für eine Modelloberfläche im RGB-Raum gespeichert.
Über die folgenden Umrechnungen kann man die X-,Y- & Z-Koordinaten einer Normalen aus der RGB-Farbe berechnen:
X = R*2-1 , Y = G*2-1 , Z = B*2-1
Die X-,Y- & Z-Koordinaten aus der Normal-Map sind im sogenannten Tangentenraum definiert und müssen noch in den Welt-Raum transformiert werden:

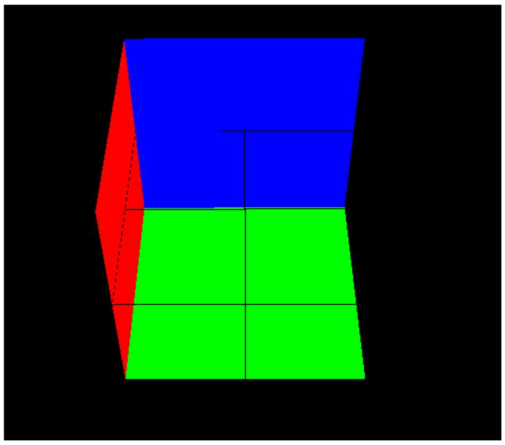


(Tangentenraum)

Um die Transformationsmatrix vom Tangentenraum in den Welt-Raum zu konstruieren, werden zusätzlich zur Normalen noch die Tangente (T) und Bitangente (B) (auch Binormale genannt) benötigt. Diese beiden Vektoren werden von der Import-Library Assimp beim Laden berechnet und können über aiMesh->mTangents und aiMesh->mBitangents abgefragt werden (prüfen Sie vorher mit aiMesh->HasTangentsAndBitangents(), ob die Vektoren vorhanden sind).

Gehen Sie nun wie folgt vor:

1. Erweitern Sie Ihre Model-Klasse:
    a. Lesen Sie die Tangenten & Bitangenten aus und speichern Sie diese im Vertex-Buffer unter Texcoord1 & Texcoord2.
    b. Prüfen Sie, wenn Sie eine Diffuse-Texturdatei mit dem Namen <name>.<ext> (AI_MATKEY_TEXTURE_DIFFUSE) laden, ob eine Normal-Map mit dem Namen <name>_n.<ext> existiert. Wenn ja, laden Sie diese Datei als Normal-Map (erweitern Sie hierfür die Struktur Model::Material).
    c. Übergeben Sie die Normal-Map-Textur in *Model::applyMaterial()* an den *PhongShader* (*normalTexture(..)*)
2. Erweitern Sie den Vertex-Shader um die Eingangsattribute für die Tangente und Bitangente. Die beiden Vektoren müssen im Vertex-Shader wie die Normale in den Welt-Raum überführt und an den Fragment-Shader weitergegeben werden.
3. Erweitern Sie den Fragment-Shader um die Eingänge der Tangente und Bitangente. Die PhongShader-Klasse übergibt die Normal-Map an die „*uniform sampler2D NormalTexture*“-Variable. **Der Sampler muss hinter dem DiffuseTexture-Sampler deklariert werden!**
4. Nutzen Sie die Tangente, Bitangente & Normale, um eine Matrix zu konstruieren (mat3(Tangente,-Bitangente,Normale)). Das Vorzeichen der Bitangente kann in Ihrer Implementierung abweichen. Wenden Sie anschließend die Matrix auf die Normale aus der Normal-Map an. Die Normal-Map nutzt die gleichen Texturkoordinaten wie die Diffuse-Map. Vergessen Sie nicht, die Normalen im Vorhinein vom RGB-Raum in den XYZ-Raum umzurechnen (wie oben beschrieben). Abschließend sollten Sie die Normale noch normalisieren. Die Normale sollte jetzt im Welt-Raum definiert sein.
5. OPTIONAL: Prüfen Sie Ihre Implementierung, indem Sie die transformierte Normale direkt als FragColor ausgeben (achten Sie darauf, dass Alpha=1 ist). Wenn Sie die Methode *Application::createNormalTestScene()* in *Application::Application()* aufrufen, sollten Sie die folgende Darstellung erhalten (die hintere Fläche sollte blau erscheinen, die linke rot und die untere grün):

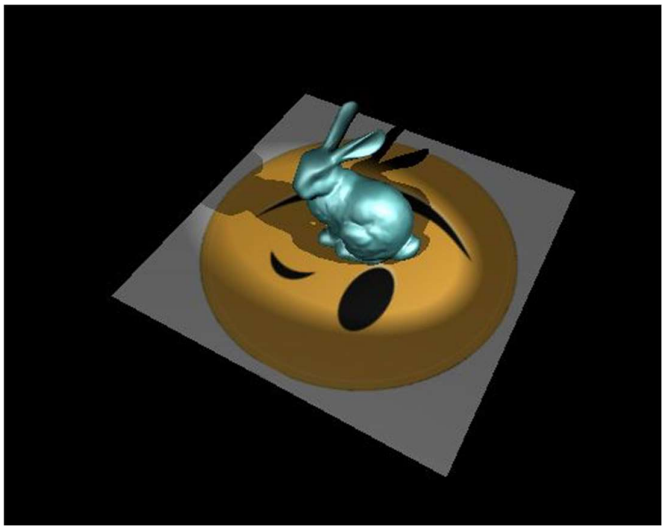


(Erscheinen bei Ihrem Shader die Farben auf anderen Flächen, so haben Sie möglicherweise einen Fehler bei der Normalen-Transformation gemacht.)

6. Rufen Sie in Application::Application() createScene() auf, es sollte sich die folgende Darstellung ergeben (die Oberflächen müssten jetzt wesentlich „rauer“ und strukturierter erscheinen):

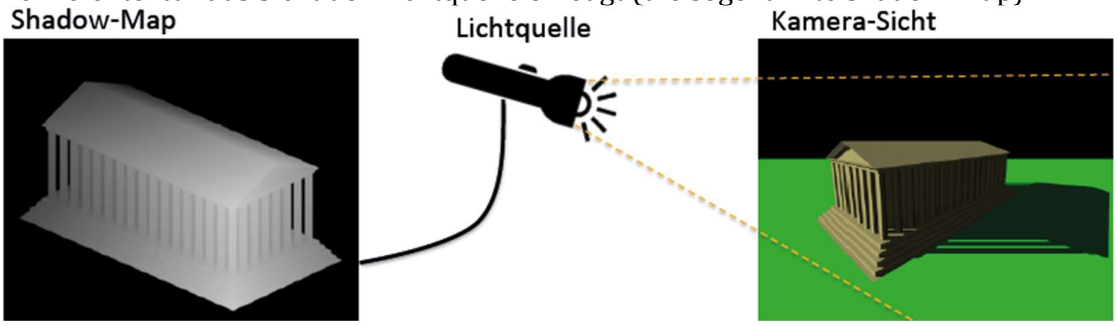
Aufgabe 4 (5 Zusatzpunkte)

Schlussendlich soll die Beleuchtung noch um optionale Schlagschatten erweitert werden:

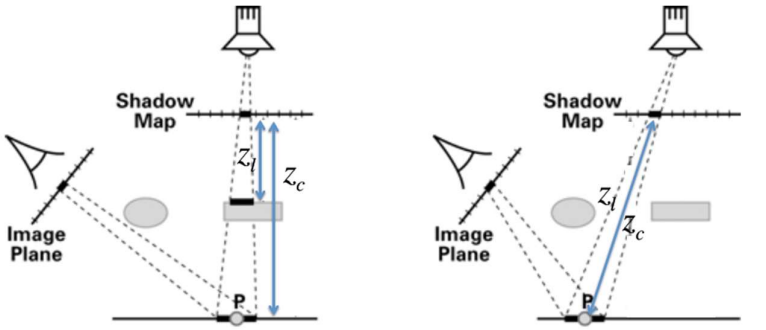


(Das Bunny wird von einer direktionalen Lichtquelle und einer Spot-Lichtquelle beleuchtet. Beide Lichtquellen haben castShadow(true) gesetzt. Sie können die Szene zeichnen, wenn Sie die Methode createShadowScene() im Application-Konstruktor aufrufen.)

Um den Schlagschatten mit Hilfe des Shadow-Mappings zu berechnen, wird für jede direktionale oder Spot-Lichtquelle in der Szene, die castShadow auf true gesetzt hat, eine Tiefentextur aus Sicht der Lichtquelle erzeugt (die sogenannte Shadow-Map):



Die Berechnung der Shadow-Map erfolgt in der Klasse ShadowMapGenerator. Die Shadow-Map wird dann als uniform-sampler2D-Variable an den Phong-Shader übergeben. In diesem muss jetzt geprüft werden, ob ein Fragment im Schatten liegt oder nicht. Hierfür wird das Fragment aus dem Welt-Raum in den Bildraum der Lichtquelle transformiert. Im Sichtraum der Lichtquelle wird nun die Tiefe des Fragments mit der Tiefe in der Shadow-Map verglichen. Ist die Tiefe des Fragments größer als der Tiefenwert in der Shadow-Map, so muss das Fragment offensichtlich im Schatten liegen, da die Lichtquelle diesen nicht sehen kann (blockiert durch vordergründiges Objekt). Die folgende Darstellung verdeutlicht das Prinzip:



$z_l < z_c \Rightarrow$ *Pixel im Schatten* $\quad z_l = z_c \Rightarrow$ *Pixel beleuchtet*

Die Matrix, die die Position des Fragments aus dem Welt-Raum in den Bildraum der Shadow-Map transformiert, wird ebenfalls vom ShadowMapGenerator berechnet. Schauen Sie sich die Methoden ShadowMapGenerator::calcProjection() und ShadowMapGenerator::calcView() an, um das Prinzip der Transformation zu verstehen. Die ShadowMapGenerator-Klasse übergibt die ShadowMap-Daten an die PhongShader-Klasse durch den Aufruf von PhongShader::shadowMap(...). Diese geht wiederum davon aus, dass die folgenden Uniform-Variablen im Fragment-Shader existieren:

```
uniform sampler2D ShadowMapTexture[MAX_LIGHTS];
uniform mat4 ShadowMapMat[MAX_LIGHTS];
```

WICHTIG: Die Shadow-Map-Sampler müssen hinter den Diffuse- und Normal-Map-Samplern deklariert werden.
Sofern eine direktionale oder Spot-Lichtquelle (warum keine Punkt-Lichtquelle?) castShadow=true gesetzt hat, wird in der Light-Struktur der ShadowIndex passend gesetzt (Index zum Element des Arrays ShadowMapTexture & ShadowMapMat). Erzeugt eine Lichtquelle keine Schatten, so ist ShadowIndex=-1.

Um das Fragment vom Welt-Raum in den Bildraum der Shadow-Map zu transformieren, müssen Sie nun die folgenden Schritte vollziehen:

PosSM = ShadowMapMat[i] * vec4(PosWorld.xyz, 1);
PosSM.xyz /= PosSM.w; // perspektivische Teilung vollziehen
PosSM.xy = PosSM.xy*0.5 + 0.5; // Koordinaten von norm. Bildraum [-1,1] in Texturkoordinaten [0,1]
DepthSM = texture(ShadowMapTexture[i], PosSM.xy)
Vergleiche ob DepthSM < PosSM.z ist ➔ Wenn ja, Fragment im Schatten, sonst nicht.

Möglicherweise muss noch ein Bias-Wert eingeführt werden, damit es zu keinen Artefakten an der Oberfläche kommt (sogenannte Surface-Acne).

Hinweis: Möglicherweise funktionieren für die Spot-Lichtquellen die Shadow-Maps nicht auf Ihrem Rechner. Dies kann passieren, wenn der verwendete Grafikkarten-Treiber keine 32-Bit-Fließkomma-Texturen unterstützt. In diesem Fall wird auf 16-Bit-Fließkomma-Texturen zurückgegriffen, die eine unzureichende Genauigkeit aufweisen. In diesem Fall können Sie die Schatten der Spot-Lichtquellen einfach ignorieren.

Wenn Sie Ihre Implementierung korrekt vorgenommen haben, sollte die Szene, die Sie mit createSzene() in Application erzeugen können, wie folgt aussehen (es wurde nur für die direktionale Lichtquelle castShadows auf true gesetzt):

Viel Erfolg!