Computergrafik Praktikum 3

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 erhalten.

Abnahmetermin: 27.04. (Montagsgruppe) 28.04. (Dienstagsgruppe)

Thema Praktikum 1

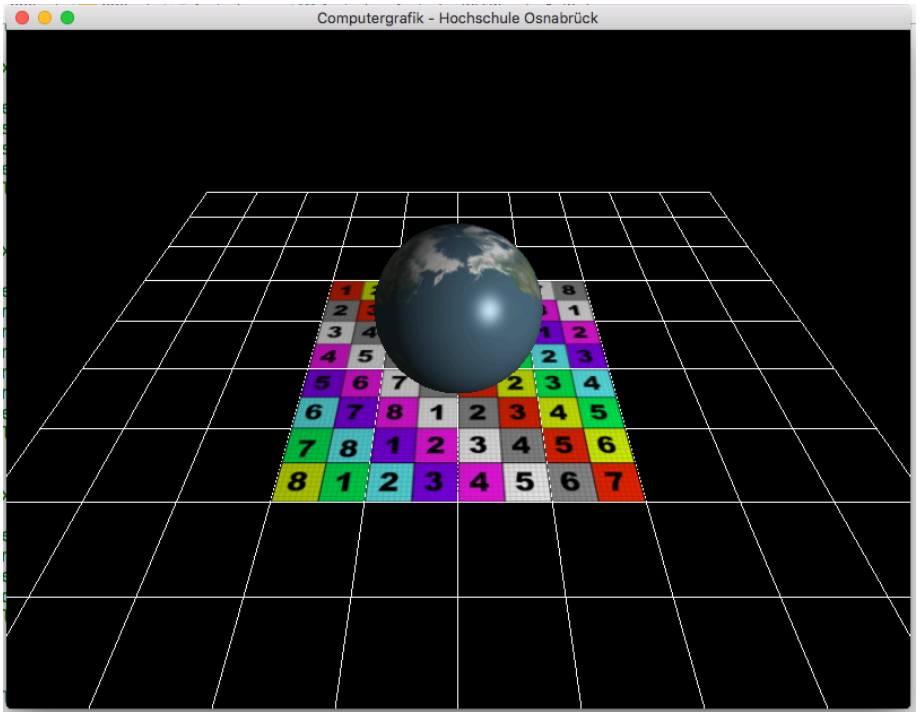
Bei der Bearbeitung der folgenden Aufgaben sollen Sie lernen, 3D-Modelle mit Hilfe von Index- und Vertex-Buffern zu erzeugen und darzustellen. Sie erlernen hierbei das Arbeiten mit unterschiedlichen Vertex-Attributen (Positionen, Normalen & Texturkoordinaten) sowie die effiziente Darstellung mit Hilfe unterschiedlicher Routinen zum Zeichnen. Darüber hinaus lernen Sie unter Verwendung einer externen Bibliothek (assimp), 3D-Modelle aus unterschiedlichen Dateiformaten in Ihre Anwendung zu laden und darzustellen.

Damit Sie das Praktikum bearbeiten können, benötigen Sie das im OSCA-Lernraum zur Verfügung gestellte Visual-Studio-2015- oder XCode-Projekt (beide befinden sich im Archiv prakt3.zip). Die Projekte können nur korrekt ausgeführt werden, wenn Sie die

folgenden Dateien mit Ihren selbstimplementierten Versionen (vorheriges Praktikum) überschreiben:

- vector.h & vector.cpp
- color.h & color.cpp
- rgbimage.h & rgbimage.cpp

Wenn Sie die Dateien ersetzt haben, sollte die Anwendung beim Start wie folgt aussehen:



In dieser Anwendung können Sie die Kamera bewegen, indem Sie eine Maustaste drücken und die Maus gleichzeitig bewegen:

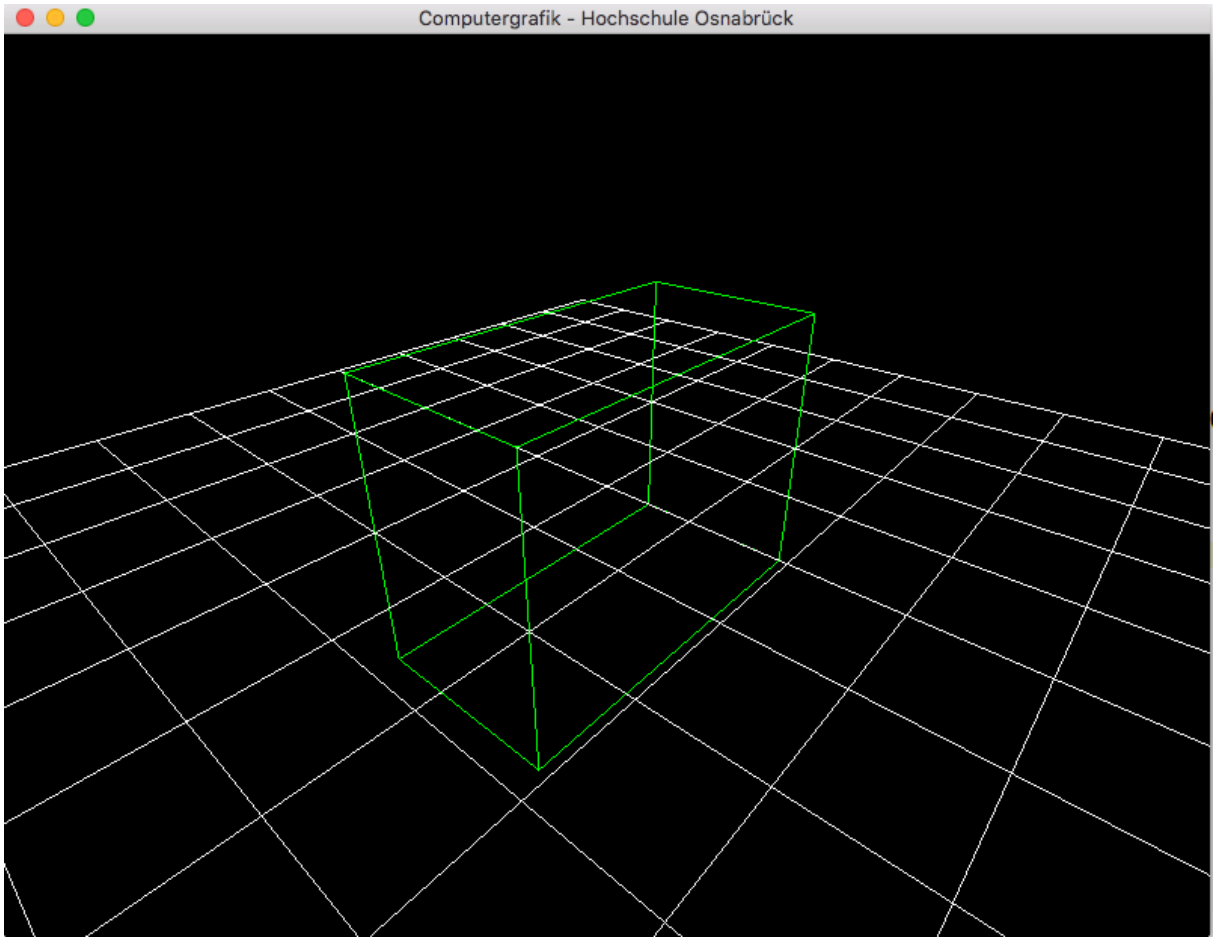
- linke Maustaste➔Kamera rotieren,
- mittlere Maustaste➔Kamera zoomen,
- rechte Maustaste➔Kamera verschieben.

Betrachten Sie den Aufbau des Programms, insbesondere die main.cpp und die Application.h & .cpp. In der Application-Klasse finden Sie eine Liste von Modellen (ModelList Models), die für das Rendering genutzt werden. Im Konstruktor Application::Application() werden drei unterschiedliche Modelle erzeugt (LinePlaneModel, TrianglePlaneModel & TriangleSphereModel) und dem Rendering hinzugefügt. Alle *Model*-Klassen erben von einer gemeinsamen Basis-Klasse (BaseModel) und können so der ModelList hinzugefügt werden. Inspizieren Sie die unterschiedlichen *Model*-Klassen und versuchen Sie, deren Aufbau zu verstehen. Ferner sollten Sie einen Blick auf die Shader-Klassen werfen und deren Aufbau ebenfalls nachvollziehen.

Aufgabe 1 (2 Punkte)

Implementieren Sie den Konstruktor und die draw()-Methode der LineBoxModel-Klasse so, dass ein „Drahtgitter-Modell“ eines Quaders angezeigt wird (die Seitenlängen des Quaders werden dem Konstruktor übergeben).

Befüllen und zeichnen Sie hierfür den als Member-Variable aufgeführten Vertexbuffer (verwenden Sie keinen Indexbuffer).
Wenn Sie die entsprechenden Aufrufe im Application-Konstruktor einkommentieren (exercise 1, siehe unten), sollten Sie die folgende Darstellung erhalten (nach erfolgter Implementierung der LineBoxModel-Klasse):



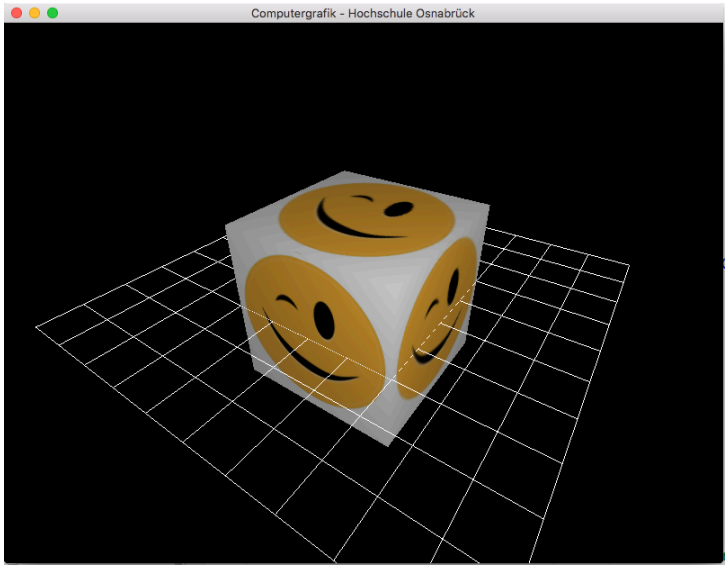
(Application-Konstruktor)

```
// Exercise 1: LineBoxModel
pModel = new LineBoxModel(2,3,4);
pConstShader = new ConstantShader();
pConstShader->color(Color(0,1,0));
pModel->shader(pConstShader, true);
Models.push_back(pModel);
```

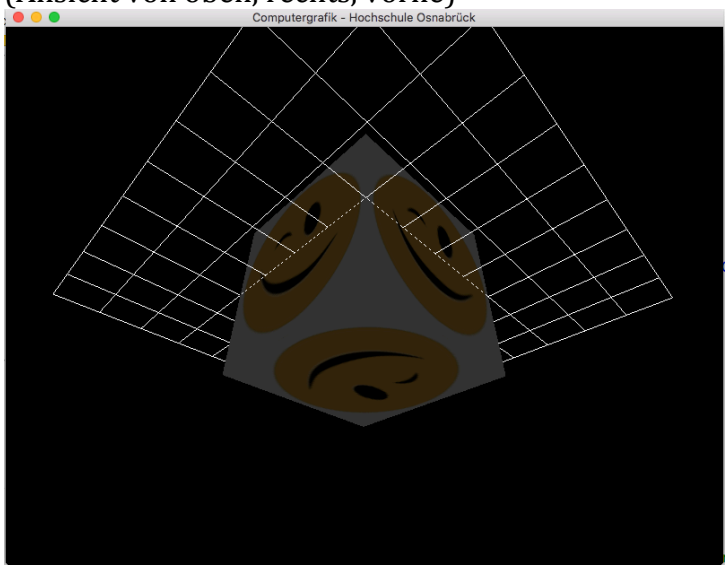
Tipp: Betrachten Sie die Klasse LinePlaneModel, um die Funktionsweise zu verstehen.

Aufgabe 2 (4 Punkte)

Implementieren Sie die Klasse TriangleBoxModel (Konstruktor & draw-Methode) so, dass ein Quader aus Dreiecken erzeugt wird. Jeder Vertex soll außer der Position eine Normale und eine UV-Texturkoordinate besitzen. Die Normalen stehen senkrecht auf den Dreiecken. Die UV-Texturkoordinaten sollen so gewählt werden, dass sich das folgende Bild ergibt (kommentieren Sie hierfür Exercise 2 im Application-Konstruktor ein):



(Ansicht von oben, rechts, vorne)



(Ansicht von unten, links, hinten)

(Application-Konstruktor)

```
pModel = new TriangleBoxModel(4,4,4);
pPhongShader = new PhongShader();
pPhongShader->ambientColor(Color(0.2f,0.2f,0.2f));
pPhongShader->diffuseColor(Color(1.0f,1.0f,1.0f));
pPhongShader->specularColor(Color(1.0f,1.0f,1.0f));
pPhongShader->diffuseTexture(Texture::LoadShared(ASSET_DIRECTORY "smiley.png"));
pModel->shader(pPhongShader, true);
Models.push_back( pModel );
```

Achten Sie darauf, dass die Textur jeweils korrekt ausgerichtet dargestellt wird. Verwenden Sie für die Implementierung zusätzlich zum Vertexbuffer auch einen Indexbuffer, so dass gleiche Vertices nicht mehrmals im Vertexbuffer abgespeichert werden müssen.

Tipp: Betrachten Sie die Klasse TrianglePlaneModel oder TriangleSphereModel, um die Funktionsweise zu verstehen.

Aufgabe 3 (4 Punkte)

Bei dieser Aufgabe sollen Sie die vorgegebene *Model*-Klasse vervollständigen, damit komplette Modelle von der Festplatte in Ihr Programm geladen und anschließend dargestellt werden können. Zum Laden der Modelle benutzen wir die freie Bibliothek Assimp, die unterschiedliche 3D-Dateiformate einlesen kann. Für die Darstellung müssen die Daten jedoch jeweils in Index- und Vertexbuffer umgewandelt werden. Ferner müssen die passenden Materialeigenschaften ausgelesen und dem Phong-Shader übergeben werden. Implementieren Sie hierfür die folgenden Methoden:

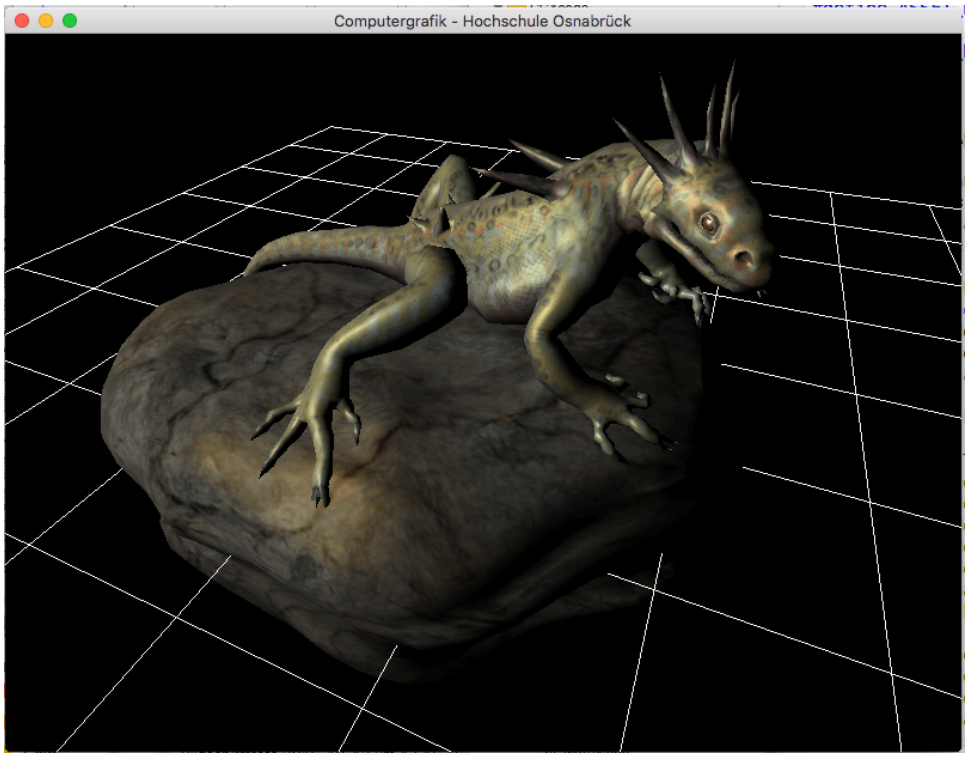
- Model::loadMeshes(const aiScene, bool FitSize)
    - Lädt die Meshes (Polygonale Modelle) aus *aiScene::mMeshes* in *Model::pMeshes* (und *MeshCount*). Hierfür müssen passende Vertex- und Indexbuffer erzeugt werden (Position, Normale & UV-Texturkoordinaten, soweit vorhanden).
    - In dieser Funktion soll auch eine achsenausgerichtete Bounding-Box für das Modell berechnet werden. Implementieren Sie hierfür die Methode *calcBoundingBox*(...) und rufen Sie diese in *loadMeshes*(..) auf.
    - Wenn *FitSize* auf *true* gesetzt wurde, soll das Mesh so skaliert werden, dass es eine sinnvolle Größe hat (z. B. 5x5x5 Einheiten o. ähnliches).
- Model::loadMaterials(const aiScene* pScene)
    - Lädt die Materialeinstellungen von Assimp in *Model::pMaterials* (und *MaterialCount*). Hierfür müssen die passenden Material-Keys von Assimp abgefragt werden (*aiGetMaterialColor* & *aiGetMaterialString* für Texturnamen).
    - Zum Laden von Texturen können Sie auf die Member-Variable *Path* zugreifen, die den Pfad zur Modell-Datei enthält. Texturen können Sie mit Hilfe der Funktion *Texture::loadShared*(...) laden.

Die von Ihnen erzeugten und beschriebenen Daten (*pMeshes*, *MeshCount*, *pMaterials*, *MaterialCount* & *BoundingBox*) werden von den restlichen Methoden genutzt, um das Modell korrekt darzustellen. Versuchen Sie den Aufbau der kompletten *Model*-Klasse zu verstehen, insbesondere den hierarchischen Aufbau via Nodes.

Wenn Sie den folgenden Block im *Application*-Konstruktor einkommentieren, sollten Sie die folgende Darstellung erhalten:

(Application-Konstruktor)

```
pModel = new Model(ASSET_DIRECTORY "lizard.dae");
pPhongShader = new PhongShader();
pModel->shader(pPhongShader, true);
// add to render list
Models.push_back( pModel );
```



Testen Sie Ihre Implementierung mit allen *.dae und *.obj Dateien aus dem Asset-Ordner. Alle Modelle sollten korrekt dargestellt werden.

Eine Dokumentation zur Assimp-Bibliothek finden Sie unter http://www.assimp.org .

Viel Erfolg!