

## Aufgabenblatt I

### Aufgabe 1: Grundlagen

Machen Sie sich selbständig mit den Grundlagen von OpenGL und vor allem GLSL vertraut. Befassen Sie sich ebenfalls mit der Dokumentation der in der Einführungsveranstaltung erwähnten Qt-Klassen (vor allem `QOpenGLShaderProgram`). Am wichtigsten sind das Verstehen der OpenGL Pipeline unter Einbeziehung der Shader, sowie für das für Aufgabe 2 notwendige Verständnis der verschiedenen Koordinatensysteme, in denen sich eine Szene befinden kann. Ein Tutorial für OpenGL/GLSL gibt es z.B. unter <http://www.opengl-tutorial.org/>. Das Thema der Koordinatensysteme wird innerhalb von Tutorial 3 besprochen. Hinweis: in dem erwähnten Tutorial wird die GLM-Bibliothek für Matrix/Vektor Operationen benutzt. Im Praktikum sollen dafür die entsprechenden Qt-Klassen verwendet werden.

### Aufgabe 2: Objekt/Kamerabewegung

Die wichtigste Aufgabe des Vertexshader ist das Setzen der Position des Vertex. Diese Position muss im Screen Space gegeben sein, also komponentenweise Werte zwischen -1 und 1 annehmen. Ein Modell bzw. dessen Vertices sind aber immer im Model Space gegeben. Das sind in der Regel die Koordinaten, mit denen das Modell erstellt wurde. Wenn man nun ein Objekt in der Welt platziert bzw. bewegt, verändert man seine Koordinaten. Das sind die Weltkoordinaten (World Space). Wenn man sich in 3D eine Szene anschaut, geschieht dies immer durch eine Kamera. Dazu wird die gesamte Szene (also alle Modelle) vor die Kamera gedreht. Die Kamera im Camera Space im Koordinatenursprung. Schlussendlich müssen alle Modelle von der 3D-Welt auf unser 2D-Ausgabemedium (den Monitor) projiziert werden (es gibt verschiedene Projektionsarten, die in der Vorlesung erklärt werden). Jetzt haben wir die Modelle im Screen Space gegeben. Die Transformation zwischen diesen ganzen Koordinatensystemen erfolgt mittels 4-dimensionalen Matrizen:



Das Ziel dieser Aufgabe soll es sein, diese „Pipeline“ zu implementieren. Die Matrizen sind im `CGViewer` vorberechnet. Details zu ihrer Erstellung werden in der Vorlesung und Übung behandelt. Darüber hinaus kümmert sich der `CGViewer` um das komplette Setup der Shader und der Vertexattribute. Ihre Aufgabe ist es, die Matrizen an das Shaderprogramm zu übergeben und den gegebenen Vertexshader (unter `shader/vertex.glsl`) so zu ergänzen, dass die korrekte `gl_Position` ausgerechnet wird. Die Vertexpositionen werden über das Vertexattribut „position“ an den Vertexshader bergeben. Die anderen beiden Vertexattribute „normal“ und „texCoords“ werden noch nicht benötigt. Die Modellmatrix für jedes einzelne Modell bekommen Sie über die Funktion `getTransformations()` der Modellklasse. Die View Matrix ist in der Scene-Klasse durch „`m_view`“ gegeben. Sie wird in der Funktion `setTransformations()` bei jedem Renderdurchgang neu aufgebaut. Ebenso ist die Projektionsmatrix ein Attribut der Scene-Klasse („`m_projection`“). Sie wird in der Funktion `resizeGL(int width, int height)` immer dann neu aufgebaut, wenn sich die Auflösung des Ausgabefensters ändert. Der `CGViewer` verwendet eine perspektivische Projektion mit einem Öffnungswinkel von  $60^\circ$ . Ergänzen Sie daraufhin den Fragmentshader so, dass eine konstante Farbe ausgegeben wird. Wenn Sie alles korrekt implementiert haben, sollten Sie die Silhouetten der geladenen Modelle sehen können, und mit der Maus die Kameraposition bzw. Rotation verändern können.

Hinweise: Im Menü Edit → Reload Shader (oder kurz: CTRL+R) können die Shader neu kompiliert und geladen werden, ohne dass der `CGViewer` neu gestartet werden muss. Fehlermeldungen, die beim Kompilieren der Shader erzeugt werden, werden auf der Konsole ausgegeben.

## Aufgabe 2: Selektion von Objekten

Im CGViewer können die geladenen Modelle frei in der Szene bewegt werden. Dazu wird ein Modell mit einem Doppelklick ausgewählt (siehe `mouseDoubleClickEvent`-Funktion in der `Scene`-Klasse). Wenn das Modell ausgewählt wurde, können verschiedene Transformationen (Verschiebung, Skalierung, Rotation) auf das Objekt angewandt werden. Dass ein Objekt ausgewählt wurde, muss dem Benutzer aber auch visuell mitgeteilt werden.

Zur Zeit besteht das gegebene Shaderprogramm („`m_program`“ in der `Scene`-Klasse) aus dem in Aufgabe 2 erweiterten Vertexshader sowie einem simplen Fragmentshader, der einfach alle Pixel, welche ein Modell darstellen, konstant einfärbt. Das Ziel dieser Aufgabe ist es, einen zweiten Fragmentshader zu schreiben, der den Pixeln einfach eine andere (ebenfalls konstante) Farbe zuweist, wenn der Nutzer das entsprechende Modell ausgewählt hat.

Legen Sie dazu einen weiteren Pointer (`std::shared_ptr`) auf ein `QOpenGLShaderProgram` in der `Scene`-Klasse an (siehe „`m_program`“). Die Initialisierung des Programms erfolgt in der Funktion `reloadShader()`. Diese Funktion wird immer dann aufgerufen, sobald der Nutzer im Menü `Edit` → `Reload Shader` auswählt (sowie beim Starten des `CGViewers`). Sehen Sie sich an, wie das Shaderprogramm „`m_program`“ geladen wird und implementieren Sie die entsprechende Initialisierung für Ihren neuen Shader. Hinweis: das neue Shaderprogramm unterscheidet sich von dem gegebenen Shaderprogramm nur durch den Fragmentshader. Der Vertexshader sollte der gleiche sein. Um zu überprüfen, ob ein Modell ausgewählt wurde, überprüfen Sie beim `render`-Vorgang das Attribut „`selectedModel`“ der `Scene`-Klasse und wählen dann das entsprechende Shaderprogramm aus. In „`selectedModel`“ ist die Position des selektierten Modells im `models`-vector gespeichert. Ist kein Modell selektiert, hat „`selectedModel`“ den Wert `-1`. Wenn Sie alles korrekt implementiert haben, sollten Sie durch einen Doppelklick auf ein Modell dessen Farbe ändern können. Ein weiterer Doppelklick sollte das Modell wieder deselektieren.