Karlsruher Institut für Technologie
Lehrstuhl für Computergrafik

Prof. Dr.-Ing. Carsten Dachsbacher
Dipl.-Inform. Johannes Meng, Dipl.-Inform. Florian Simon,
M.Sc. Emanuel Schrade

7. Übungsblatt zur Vorlesung Computergraphik im WS 2016/17

Abgabe bis **Freitag, 10.02.2017**, 11:00 Uhr.

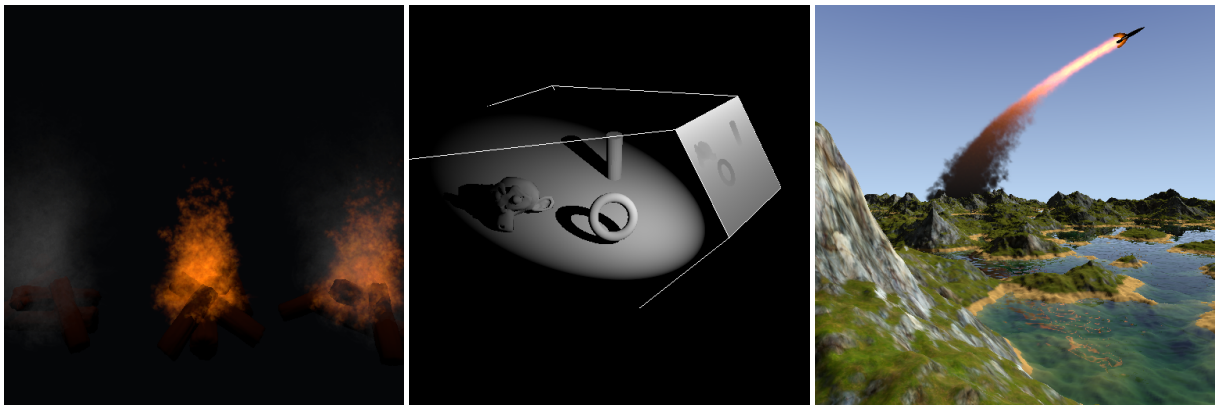

Abbildung 1: Ergebnisbilder des Übungsblattes.

Dieses Übungsblatt beschäftigt sich mit Shadow Mapping und fortgeschrittenen Realtime Rendering-Techniken in modernem OpenGL. Sie müssen in diesem Übungsblatt OpenGL-Shader in GLSL implementieren. Eine ausführliche Dokumentation von OpenGL und GLSL (3.x) finden Sie z.B. unter
`http://docs.gl`,
`https://www.opengl.org/registry/doc/GLSLangSpec.3.30.6.clean.pdf` und
`https://www.opengl.org/sdk/docs/man`.

Die Ergebnisbilder einer kompletten Implementierung sind in Abbildung 1 dargestellt. Lesen Sie bitte das Übungsblatt sorgfältig und machen Sie sich mit dem Code des Frameworks vertraut.

Hinweis: Zur Abgabe Ihrer Lösung sollen Sie ein zip-Archiv mit dem Namen `solution.zip` erstellen, das die von Ihnen bearbeiteten Dateien enthält. Geben Sie nur Dateien ab, die explizit auf dem Übungsblatt erwähnt werden.

Hinweis: Während der Bearbeitung wird bei Ihnen wahrscheinlich die Warnung *Warning: uniform "xyz" not found in shader!* ausgegeben werden. Dies ist normal und tritt auf, wenn eine Uniform-Variable im Shader nicht benutzt wird oder vom GLSL-Compiler wegoptimiert wurde. Nach Abschluss der Implementierung sollten keine Warnungen mehr auftreten.

Hinweis: Wenn Sie Shader implementieren, müssen Sie das Programm nicht nach jeder Änderung im Shader neu starten. Es genügt in der GUI den Button "Reload all shaders" zu drücken.

Hinweis: Sämtliche Winkel werden in der `glm`-Version des Frameworks in Bogenmaß angegeben.

Fehlerbehebung unter Windows: Fehler in der Originalversion von Microsoft Visual C++ 2013 führen zu fehlerhaften ausführbaren Dateien, welche das Bearbeiten einiger Aufgaben

unmöglich machen. Glücklicherweise beheben die offiziellen Updates diese Fehler. Stellen Sie sicher, dass Sie mit der aktualisierten Version arbeiten (Update 4) und aktualisieren Sie ihr Visual Studio ggf. über `Tools` → `Extensions and Updates`, dort in der Sidebar `Updates` → `Product Updates`, wählen Sie `Visual Studio 2013 Update 4`.

1 *Shadow Mapping* — 8 Punkte

In dieser Aufgabe sollen Sie den Shadow Mapping Algorithmus für ein Spotlight implementieren. Das Verfahren läuft in zwei Schritten ab. Zunächst wird die Szene aus Sicht der Lichtquelle gezeichnet und der Tiefenpuffer in einer Textur gespeichert. Im zweiten Schritt kann nun bestimmt werden, ob ein Oberflächenpunkt im Schatten liegt, indem der Oberflächenpunkt in das Koordinatensystem der Lichtquelle transformiert und in die Shadow Map projiziert wird. Durch einen Vergleich des Tiefenwertes des transformierten Punktes mit dem gespeicherten Tiefenwert kann nun festgestellt werden ob der Oberflächenpunkt im Schatten liegt.

Machen Sie sich mit der Implementierung des Algorithmus in `cglib/src/gl/renderer.cpp` in `ShadowmapRenderer::draw` vertraut.

1. Im ersten Teil der Aufgabe sollen Sie die View-Projection Matrix der Lichtquelle in der Funktion `compute_view_projection_light` in der Datei `exercise_07.cpp` berechnen. Diese soll einen Punkt von Weltkoordinaten in den homogenen Clip-Space der Lichtquelle transformieren. Sie können davon ausgehen, dass die Shadow Map Textur quadratisch ist.

2. Implementieren Sie nun im Fragment-Shader `shadowmap.frag` den Zugriff auf die Shadow Map Textur und den Tiefenvergleich. Die Position des Vertex in homogenen Clip-Koordinaten der Lichtquelle ist gegeben in `pos_shadowmap_space`. Transformieren Sie die x, y und z Koordinate so, dass Sie damit auf die Textur zugreifen und die Tiefenwerte vergleichen können. Die Tiefenwerte in der Shadow Map sind im Bereich $[0, 1]$. Modifizieren Sie zur Vermeidung von Selbstverschattung vor dem Tiefenvergleich zusätzlich den Tiefenwert der Shadow Map um den Parameter `shadow_bias`. Überlegen Sie sich, ob und wann Sie die Clip-Koordinaten des Vertex dehomogenisieren müssen.

2 *Blending* — 4 Punkte

In Abbildung 1 sehen Sie die "FirePlace"-Szene. Der Rauch links ist mit Alpha-Blending gezeichnet, das Feuer in der Mitte mit additivem Blending, die Kombination aus Rauch und Feuer rechts mit vormultipliziertem Alpha-Wert. In dieser Aufgabe sollen Sie die entsprechenden Zustände mit `glBlendEquation()` und `glBlendFunc()` setzen. Implementieren Sie dies in den Funktionen `initialize_alpha_blending()`, `initialize_additive_blending()` und `initialize_premultiplied_blending()` in der Datei `exercise_07.cpp`.

Die Vormultiplikation ist notwendig, damit Rauch und Feuer gemischt dargestellt werden können. Dabei wird die Farbe eines Fragments schon im Fragment Shader mit dem Alphawert wie folgt multipliziert:

$$c = (R \cdot \alpha, G \cdot \alpha, B \cdot \alpha, \alpha) \qquad \text{Alpha-Blending}$$
$$c = (R \cdot \alpha, G \cdot \alpha, B \cdot \alpha, 0) \qquad \text{Additives Blending}$$

Wählen Sie hier `glBlendEquation()` und `glBlendFunc()` so, dass der Effekt im Framebuffer jeweils derselbe ist wie in den Beispielen mit rein additivem Blending bzw. reinem Alpha-Blending.

3 *Fortgeschrittene Realtime Rendering-Techniken* **8 Punkte**

In dieser Aufgabe sollen Sie wichtige Elemente fortgeschrittener Realtime Rendering-Techniken wie Displacement Mapping, semi-prozedurale Texturierung, Blending, einfache volumetrische Absorption und Partikeleffekte implementieren. Alle Teilaufgaben werden unabhängig bewertet.

Wichtig: Da die geometrische Komplexität der Testszene für schwächere GPUs zu hoch sein könnte, haben Sie die Möglichkeit, eine Detailverminderung einzustellen. Ändern Sie hierzu den von der Funktion `get_default_landscape_quality_reduction()` in `main.cpp` zurückgegebenen Wert auf 0, 1, 2 oder 3. Der *höchste* Detailgrad ist 0, jeder Schritt viertelt die Komplexität.

Die Landschaft in der Testszene ist prozedural modelliert. Das Framework nutzt ein einfaches heterogenes Modell, um ein Höhenfeld zu erzeugen. Die Implementierung dieses Modells auf der GPU können Sie sich in `heightmap.frag` ansehen. Als Zufallsquelle dient eine einfache Textur, welche bei Programmstart mit Zufallszahlen gefüllt wird.

1. Implementieren Sie zunächst im Vertex Shader `vertex_displacement.vert` Vertex Displacement Mapping. Der Shader erhält von der Anwendung die Höhenfeld-Textur `HeightMap` und einen Höhenskalierungsfaktor `height_scaling`, welcher angibt, wie stark die verarbeitenden Vertices entlang der Y-Achse entsprechend den Einträgen der `HeightMap` verschoben werden sollen. Abbildung 2(a) zeigt den Zusammenhang von Gitterkoordinaten und Texturkoordinaten der Heightmap.

   Die Verschiebung der Vertices beeinflusst auch die Oberflächennormalen. Berechnen Sie deshalb zusätzlich zu den neuen Vertexpositionen aus dem Höhenfeld auch korrekte Normalen. Abbildung 2(b) zeigt, wie sich die Vertexnormale aus den Höhenwerten der 4 direkten Nachbarn im Gitter berechnen lässt.

2. Implementieren Sie nun im Fragment Shader `terrain.frag` eine einfache semi-prozedurale Texturierung der Landschaft in Abhägigkeit des Höhenverlaufs. Wenn Sie diese Teilaufgabe nicht bearbeiten, lassen Sie die Texturierung unverändert, damit eine unabhängige Bewertung der restlichen Teilaufgaben möglich ist.

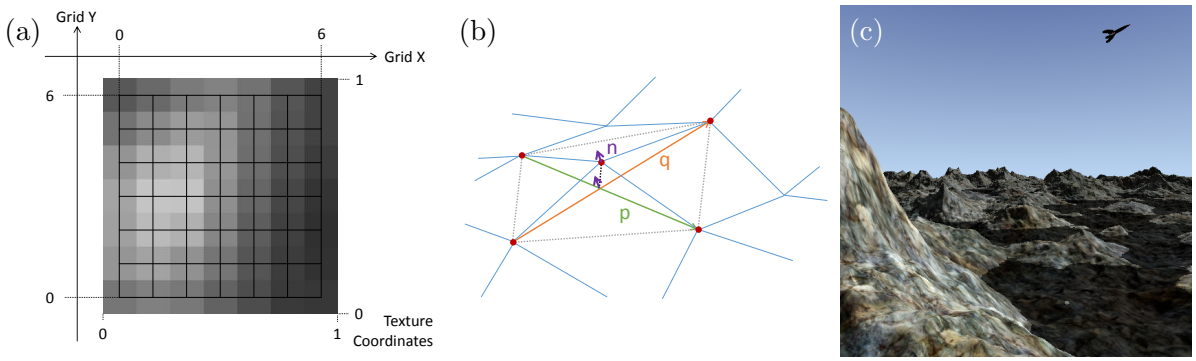

Abbildung 2: (a) Zusammenhang zwischen Heightmap-Texturkoordinaten und Gitterkoordinaten. (b) Berechnung der Vertexnormalen **n** orthogonal zu den Vektoren **p** und **q**, welche die 4 direkten Nachbarn im Gitter verbinden. (c) Ausgabe nach erfolgreicher Verschiebung und Normalenberechnung.

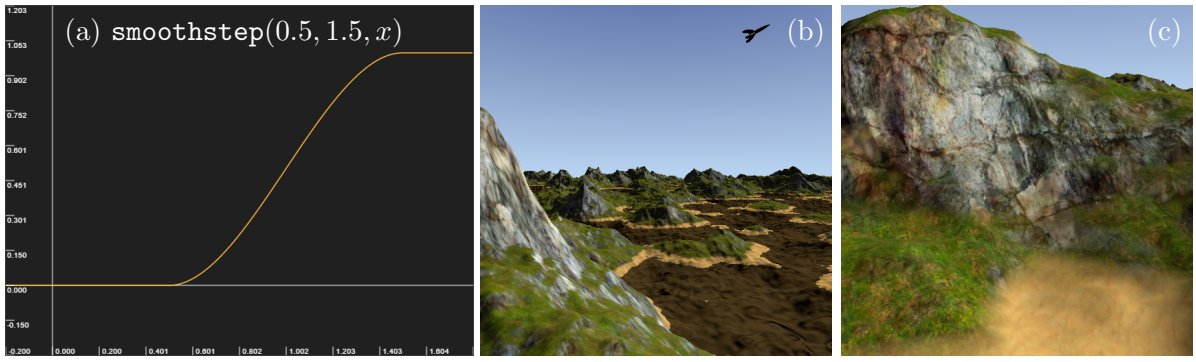

Abbildung 3: (a) Verlauf von `smoothstep(0.5, 1.5, x)`. (b, c) Ausgabe nach erfolgreicher Implementierung der Textur-Übergänge.

Waagrechte Flächen sollen mit Gras bedeckt sein. Die aus der Textur gelesene entsprechende Farbe ist bereits in der Variable `grass_color` verfügbar. Senkrechte Flächen sollen mit Fels bedeckt sein. Die entsprechende Texturfarbe ist bereits in der Variablen `rock_color` hinterlegt. Die Texturzuordnung soll in Abhängigkeit der Steilheit mit weichem Übergang erfolgen. Für weiche Übergänge zwischen vorgegebenen unteren und oberen Schwellwerten lässt sich die GLSL-Funktion `smoothstep` verwenden, welche innerhalb der Schwellwerte eine kubisch hermitesche Interpolation zwischen 0 und 1 durchführt und außerhalb der Schwellwerte konstant die Werte 0 bzw. 1 beibehält. Die Uniform-Variable `min_rock_threshold` gibt den Sinus des Winkels θ zwischen Oberflächennormale und Y-Achse an, bis zu dem die Oberfläche vollständig mit Gras bedeckt sein soll. Zwischen `min_rock_threshold` und `min_rock_threshold` + `rock_blend_margin` soll mit `smoothstep` in Abhängigkeit von $\sin\theta$ ein Blend-Faktor ausgerechnet werden[1], mit dem dann linear zwischen Gras und Fels interpoliert wird. Zur linearen Interpolation können Sie die GLSL-Funktion `mix` verwenden. Steilere Oberflächen außerhalb des Übergangsbereichs sollten dann vollständig mit Fels bedeckt sein.

Bis zu einer bestimmten Höhe über der Wasseroberfläche `water_height + beach_margin` soll die Oberfläche außerdem vollständig von Sand bedeckt sein. Oberhalb soll der Sand mit linearem Verlauf binnen einer Welteinheit vollständig der Fels-Gras-Oberfläche weichen.

3. Implementieren Sie nun ein einfaches Absorptionsmodell für die Lichtausbreitung unter Wasser. Grundsätzlich soll die duchgelassene Lichtmenge exponentiell mit der unter Wasser zurückgelegten Strecke abnehmen. Die Transmissionskoeffizienten σ_t für rotes, grünes und blaues Licht sind in der Uniform-Variable `water_transmission_coeff` gespeichert. Der Anteil transmittierten Lichts in Abhängigkeit der im Wasser zurückgelegten Strecke d ergibt sich als $e^{-\sigma_t d}$.

   Nutzen Sie die in `terrain.frag` in Weltkoordinaten gegebene Position des Landschaftsoberflächen-Fragments `world_position`, die Kameraposition `cam_world_pos` und die ebenfalls in Weltkoordinaten gegebene Wasser-Höhe `water_height`, um die Länge des unter Wasser liegenden Kamerastrahlsegments zu berechnen. Nutzen Sie außerdem die gegebene Sonneneinfallsrichtung `sun_world_dir` (in Lichtausbreitungsrichtung), um die Länge des unter Wasser liegenden Lichtstrahlsegments zu berechnen.

   Multiplizieren Sie abschließend die Variable `color` mit der berechneten Lichttransmission, um die Absorption anzuwenden.

[1] Beachten Sie: $\sin\theta = \sqrt{1 - \cos^2\theta}$, wobei sich $\cos\theta$ mit Hilfe des Skalarprodukts berechnen lässt.

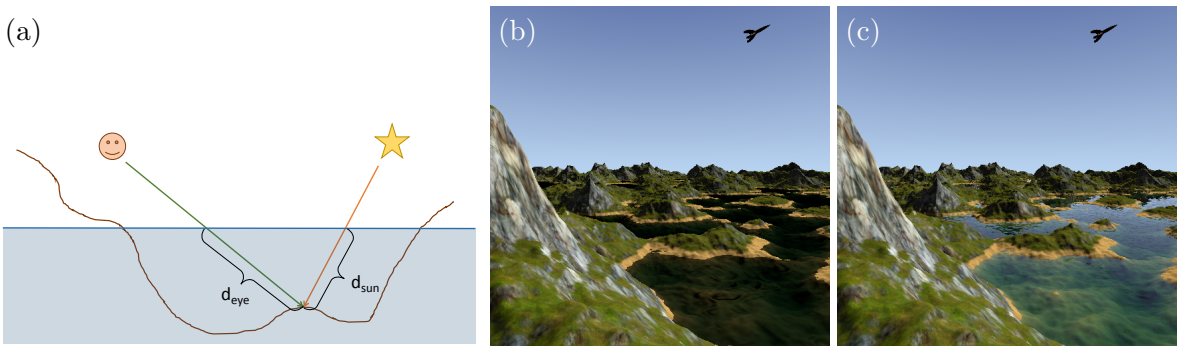

Abbildung 4: (a) Absorption findet auf den unter Wasser gelegenen Segmenten von Kamera- und Sonnenstrahlen statt. (b) Ausgabe nach erfolgreicher Implementierung des volumetrischen Absorptionsmodells. (c) Ausgabe nach erfolgreicher Implementierung von Reflexion, Fresnel-Effekt und Oberflächentransmission.

4. Das Framework rendert eine Reflection Map für die Wasseroberfläche, welche Sie sich über die entsprechende Option in der GUI anzeigen lassen können. Die Reflection Map wird gerendert, indem die Kameratransformation an der näherungsweise planaren Wasseroberfläche gespiegelt wird. Damit enthält sie prinzipiell nur Informationen für eine perfekt planare Spiegeloberfläche, tatsächlich lässt sich aber mit einigen Vorkehrungen auch das Aussehen einer unebenen spiegelnden Oberfläche approximieren. Diese Vorkehrungen können Sie in `water.frag` sehen, wo bereits eine geeignete Reflexionsrichtung `reflect_dir` berechnet wird, mit der Sie das reflektierte Licht in der als Uniform-Sampler-Variable gegebenen `ReflectionMap` nachschlagen können. Transformieren Sie hierzu die gegebene Reflexionsrichtung in den homogenen Texturraum der Reflection Map (gegeben durch die Transformationsmatrix `RVP_to_tex`). Berechnen Sie daraus die dehomogenisierten Texturkoordinaten. Wenn Sie diese Teilaufgabe nicht bearbeiten, lassen Sie die Reflexionsfarbe unverändert auf 0.5, damit eine unabhängige Bewertung der restlichen Teilaufgaben möglich ist.

5. An der Wasseroberfläche überlagern sich das reflektierte und das transmittierte Licht. Zum Zeitpunkt des Zeichnens der Wasseroberfläche steht das nach Volumenabsorption transmittierte Licht bereits im Framebuffer, da zuvor bereits die Landschaft gezeichnet wurde. Die Wasseroberfläche lässt davon jedoch wiederum nur einen Teil durch. Das reflektierte Licht, welches in der vorigen Teilaufgabe berechnet wurde, kommt additiv hinzu, wobei der Anteil des reflektierten Lichts von der winkelabhängigen Reflektivität R abhängt (sog. *Fresnel*-Effekt). Zur Annäherung der winkelabhängigen spekularen Reflektivität nutzen wir Schlick's Approximation (auch *Fresnel*-Faktor genannt):

$$R = R_0 + (1 - R_0)(1 - \cos\theta)^5.$$

Hierbei gibt θ den Winkel zwischen Sichtrichtung und Reflexionsmittelvektor, also in diesem Fall der Normalen der Wasseroberfläche, an. Für R_0 nehmen wir 0.02 an. Der Anteil transmittierten Lichts ergibt sich in unserem Modell nun als Produkt der Wasseroberflächenfarbe `surface_transmission_color` und $(1 - R)$.

Die Wasseroberfläche wird mit den OpenGL Blend States `glBlendEquation(GL_FUNC_ADD)` und `glBlendFunc(GL_ONE, GL_SRC1_COLOR)` gezeichnet. Das heißt, dass in `water.frag` die Fragment Shader Output-Variable `add_color` mit dem Index 0 als Quellfarbe 0 zum Framebuffer addiert wird, während die Output-Variable `mul_color` mit dem Index 1 als Quellfarbe 1 mit den vorigen Werten im Framebuffer multipliziert wird. Nutzen Sie die beiden Variablen, um das beschriebene Reflexions- und Transmissionsmodell umzusetzen.

4 *Bonus: Partikel mittels Geometry Shader* **8 Punkte**

Abschließend sollen Sie noch dafür sorgen, dass die Partikel der startenden Rakete korrekt gezeichnet werden. Wie Sie in `cglib/src/gl/renderer.cpp` in der Methode `ProceduralLandscapeRenderer::drawParticles` sehen können, werden die Partikel als einfaches vorsortiertes Punkt-Array an OpenGL übergeben. Der Vertex Shader `particle.vert` zerlegt jeden eingehenden Punkt in dessen Weltkoordinate und Radius und gibt beide unverändert weiter. Der Geometry Shader `particle.geom` erhält beide als Eingabe und spezifiziert bereits mit `layout(triangle_strip, max_vertices = 4) out;` einen Triangle Strip von 4 Vertices als Ausgabe.

Ihre Aufgabe ist es, aus den eingehenden Punkten Kamera-ausgerichtete Quadrate zu berechnen, auf welche der vorgegebene Fragment Shader `fire.frag` dann Partikel-Texturen für Feuer und Rauch zeichnen kann. Berechnen Sie hierzu die Normale des kameraausgerichteten Quadrats als Richtung von der gegebenen Partikelposition `world_position[0]`[2] zur gegebenen Kameraposition `cam_world_pos`. Die Hochachse des Quadrats erhalten Sie als Kreuzprodukt der Normalen und der mit dem gegebenen Zufallswinkel α (`randomAngle`) gedrehten Achse $(\cos\alpha, \sin\alpha, 0)$. Die Rechtsachse des Quadrats können Sie als Kreuzprodukt der berechneten Hochachse und der Normalen berechnen. Vergessen Sie nicht, die Richtungen zu normalisieren und das Quadrat mit den Seitenlängen 2 `world_radius[0]` zu erzeugen.

Vertices erzeugen Sie im Geometry Shader mit der GLSL-Funktion `EmitVertex`, welche Sie immer dann aufrufen, wenn Sie alle Ausgabevariablen für den nächsten Vertex aktualisiert haben. Um den Triangle Strip abzuschließen, rufen Sie die GLSL-Funktion `EndPrimitive` auf.

[2] Geometry Shader erwarten als Eingabe stets Arrays, selbst wenn diese bei Punkten nur ein Element enthalten.

Abgabe

Laden Sie die Datei `solution.zip` in Ilias hoch. Achten Sie darauf, dass dieses Archiv alle von Ihnen bearbeiteten Dateien enthält: `exercise_07.cpp`, `shadowmap.frag`, `vertex_displacement.vert`, `terrain.frag`, `water.frag`, `particle.geom`

Framework

Wir werden für jedes Übungsblatt ein Framework bereitstellen, das Sie im Ilias-Kurs unter `https://ilias.studium.kit.edu/goto.php?target=crs_607961&client_id=produktiv` herunterladen können. Das Framework nutzt C++ 11 und wird unter Linux getestet. Es ist allerdings auch unter Windows mit Visual Studio 2013 lauffähig.

Sie können das heruntergeladene Archiv unter Linux mit dem Befehl

```
$ unzip archiv.zip
```

entpacken, wobei Sie `archiv.zip` durch den jeweiligen Dateinamen ersetzen müssen.

Grundsätzlich wird das Framework immer ein Unterverzeichnis `cglib` enthalten. Sie dürfen und sollen den Quellcode in diesem Unterverzeichnis lesen.

Je nach Aufgabe wird es auch ein zweites Unterverzeichnis geben. Für das vorliegende Übungsblatt heißt dieses `07_shadow_mapping`. Hier werden Sie Ihre Lösung programmieren.

Die Dateien `VirtualMachine.txt` und `Kompilieren.txt` enthalten Informationen darüber, wie sie die Virtuelle Maschine zur Übung installieren und das Framework kompilieren. Bitte lesen Sie diese Informationen.

Achtung: Abgegebene Lösungen müssen in der VM erfolgreich kompilieren und lauffähig sein, ansonsten vergeben wir 0 Punkte. Insbesondere darf Ihre Lösung nicht abstürzen.

Allgemeine Hinweise zur Übung:

- Scheinkriterien: Sie benötigen 60% der Punkte aus den Praxisaufgaben.
- Die theoretischen Aufgaben bedürfen üblicherweise *keiner* elektronischen Abgabe.
- Die Aufgaben müssen in Ilias bis spätestens **Freitag, 10.02.2017**, 11:00 Uhr abgegeben werden.
- Die Abgabe muss im Ordner `build` mit `cmake ../ && make` in der bereitgestellten VirtualBox VM kompilieren, andernfalls wird die Aufgabe mit 0 Punkten bewertet.
- Da nur einzelne Dateien abgegeben werden, müssen diese kompatibel zu unserer Referenzimplementation bleiben. **Verändern Sie daher wirklich nur die Dateien, die auch abgegeben werden müssen**, bzw. *nicht* die mitgelieferten Funktionsdeklarationen! Sie können allerdings in in den *abzugebenden* Dateien gerne Hilfsfunktionen definieren und benutzen.
- Sie dürfen sehr gerne untereinander die Aufgaben diskutieren, allerdings muss jeder die Aufgaben *selbst* lösen, implementieren und abgeben. Plagiate bewerten wir mit 0 Punkten.

- Wenden Sie sich bei Fragen an einen Übungsleiter. Unsere Büros sind in Gebäude 50.34.

Johannes Meng	Raum 142	meng@kit.edu
Emanuel Schrade	Raum 140	schrade@kit.edu
Florian Simon	Raum 142	florian.simon@kit.edu