

3. Übungsblatt zur Vorlesung Computergrafik im WS 2024/2025

Abgabe bis Montag, 16.12.2024, 08:00 Uhr

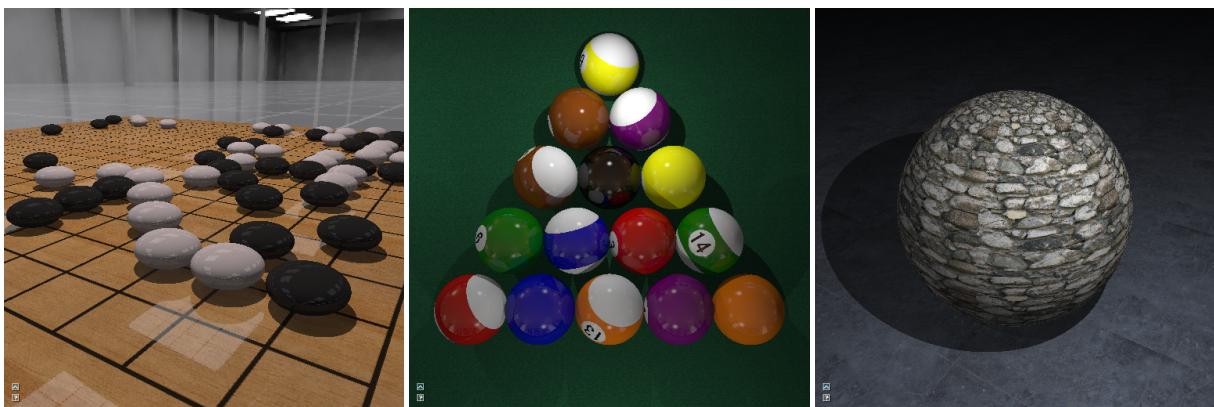
1 Texture Mapping, Aliasing und Transformationen**20 Punkte**

Abbildung 1: Ergebnisbilder des Übungsblattes.

In diesem Übungsblatt sollen Sie den Whitted-Style Raytracer um Texturen und Transformationen erweitern. Texturen helfen dabei, den Detailreichtum der Szene zu erhöhen, ohne die geometrische Komplexität zu vergrößern. Zudem sollen Objekte transformiert (also z.B. skaliert oder rotiert) werden können.

Ein häufiges Problem bei der Verwendung von Texturen ist Aliasing. Dieses Übungsblatt beschäftigt sich mit der Reduktion von Aliasing-Artefakten durch bilineare und trilineare Filterung der Texturen.

Das Weltkoordinatensystem ist ein rechtshändiges Koordinatensystem, in dem die y -Achse „nach oben“ zeigt.

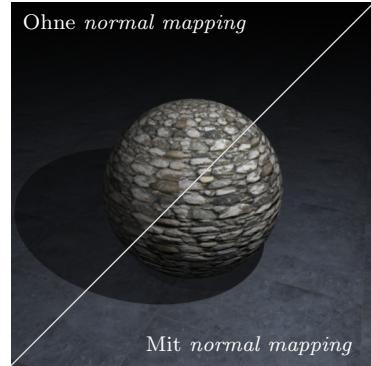
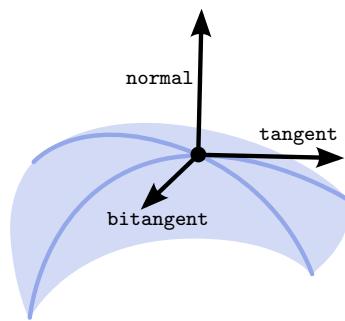
Die Ausgaben einer kompletten Implementierung sind in Abbildung 1 dargestellt. Ihre Implementierung beschränkt sich wieder auf die Datei `exercise_03.cpp`. Lesen Sie bitte das Übungsblatt sorgfältig und machen Sie sich mit dem Code des Frameworks vertraut.

Durch starten des Frameworks mit dem Argument `--create-images` können Sie alle Konfigurationen der Referenzbilder aus `assignment_references` mit ihrer Lösung auf einmal erzeugen. Diese werden in den Ordner `assignment_images` geschrieben. Die Bilder auf dem Übungsblatt dienen nicht als Referenz, sondern nur zur Illustration.

- (4 Punkte)** Diese Aufgabe befasst sich mit Transformationen. Im ersten Teil sollen Sie die Funktionen `transform_direction` und `transform_position` implementieren, die eine Richtung bzw. Position mit einer gegebenen, homogenen Transformationsmatrix `transform` transformieren. Beachten Sie hierbei, dass der Ausgabevektor von `transform_direction` auf Länge 1 normiert sein muss.

Implementieren Sie dann `Object::intersect`. Diese Methode soll den gegebenen Strahl `ray` mit dem jeweiligen Objekt schneiden. Dabei sollen Sie die Transformationsmatrix des Objektes, `Object::transform_world_to_object`, benutzen. Der Strahl soll also zunächst aus Welt- in Objektkoordinaten transformiert werden. Dann soll der Strahlschnitt in Objektkoordinaten durchgeführt werden. Bedenken Sie, dass auch `Intersection::t` nach dem Schnitttest wieder in Weltkoordinaten zurücktransformiert werden muss. Zur Transformation der restlichen Komponenten von `Intersection` und zur Transformation eines Strahls können Sie die Hilfsfunktionen `transform_intersection` bzw. `transform_ray` aus `transform.h` benutzen.

Man kann Strahlschnitte in Objektkoordinaten beispielsweise einsetzen, um mit dem Strahlschnitt für Kugeln auch Ellipsoide darzustellen. Ein Beispiel können Sie in der "Go-Brett"-Szene sehen (Abbildung 2a). Die Spielsteine sind mit transformierten Kugeln modelliert.



- (a) Die "Go-Brett"-Szene. Die Spielsteine sind durch skalierte Kugeln modelliert.
- (b) Ein Tangentenraum, aufgespannt durch Tangente, Normale und Bitangente.
- b) (2 Punkte) In dieser Aufgabe sollen Sie in `transform_direction_to_object_space` eine Richtung `d` aus einem gegebenen Tangentenraum in den Objektraum transformieren. Der Tangentenraum wird definiert durch ein rechtshändiges Orthonormalsystem aus Tangente `tangent`, Normale `normal` und Bitangente `bitangent`. Die drei Vektoren sind im Objektraum gegeben. Eine Darstellung des Tangentenraums finden Sie in Abbildung 2b. Erstellen Sie zunächst mithilfe dieser drei Vektoren eine Rotationsmatrix, die Vektoren aus dem Tangentenraum in den Objektraum transformiert. Rotieren Sie dann die Eingaberichtung `d` mit dieser Matrix. Die Normale entspricht dabei der `y`-Achse.

Stellen Sie sicher, dass die Ausgabe der Funktion auf Länge 1 normiert ist.

Mit dem Tangentenraum lässt sich beispielsweise *normal mapping* durchführen. Normalen werden dabei aus einer Textur ausgelesen. Üblicherweise speichert man die Normalen im Tangentenraum, um unabhängig von der Objektgeometrie zu sein. Ein Beispiel für den Effekt von *normal mapping* sehen Sie in Abbildung 2c.

- c) (2 Punkte) In dieser Aufgabe sollen Sie in der Methode `ImageTexture::evaluate_nearest` einen einfachen Texturzugriff implementieren. Wenn ein Strahl ein Objekt schneidet, wird für den Schnittpunkt eine uv -Koordinate berechnet. Der uv -Raum ist ein von der Texturauflösung unabhängiger Raum, in dem sich die Textur im Einheitsquadrat $[0, 1]^2 \subset \mathbb{R}^2$ befindet. Sie sollen hier die uv -Koordinaten in Texturkoordinaten umwandeln. Für eine W Texel breite und H Texel hohe Textur sollen dabei die Punkte $(u, v) = (0, 0)$ auf $(s, t) = (0, 0)$ und $(u, v) = (1, 1)$ auf $(s, t) = (W, H)$ monoton und linear abgebildet werden. Dazu müssen Sie die Breite und Höhe der Mipmap Stufe herausfinden. Runden Sie anschließend die st -Koordinaten zu ganzzahligen Texelkoordinaten ab. Greifen Sie mit diesen dann über die Funktion `get_texel` auf die Textur zu. Eine Textur wird später eine ganze Reihe von Stufen mit unterschiedlichen

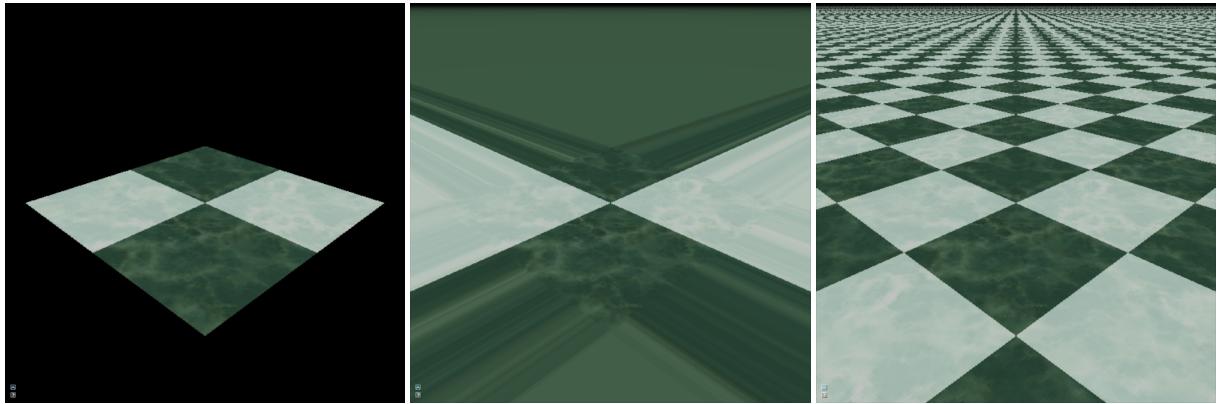


Abbildung 3: Verschiedene Zugriffsmöglichkeiten (Zero, Clamp, Repeat).

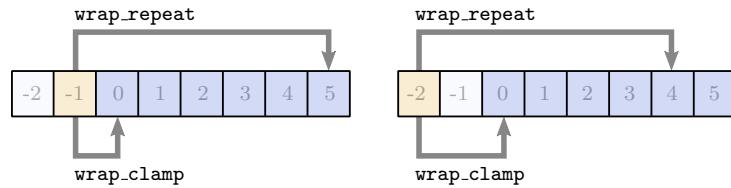


Abbildung 4: *Clamping* und *repeating* für zwei verschiedene Texel außerhalb des Definitionsbereiches mit `size=6`.

Auflösungen haben. Sie erhalten daher den Parameter `level`, der angibt, auf welche Stufe zugegriffen werden soll. Die Stufe 0 ist dabei die Stufe mit der höchsten Auflösung, also in der Regel die ursprüngliche Textur. Wenn Sie diese Methode richtig implementiert haben, sollten Sie die Textur wie in Abbildung 3 (links) sehen, wenn sie den Wrap-Modus in der GUI auf ZERO setzen.

- d) **(2 Punkte)** Die uv -Koordinaten müssen sich nicht im Bereich $[0, 1]^2$ befinden. So kann man z.B. sich wiederholende Texturen realisieren. Die entsprechenden Texelkoordinaten müssen aber vor dem Texturzugriff auf den gültigen Bereich $[0, 1, \dots, W - 1] \times [0, 1, \dots, H - 1]$ abgebildet werden (*wrapping*). Implementieren Sie dazu die Methoden `ImageTexture::wrap_clamp` und `ImageTexture::wrap_repeat`. Diese sollen Überschreitungen des gültigen Bereichs mit *clamping* oder *repeating* auflösen und in den gültigen Bereich bringen (Abbildung 4). In `ImageTexture::wrap_repeat` soll ein gegebener Bereich $[0, \text{size}]$ periodisch fortgesetzt werden, während die Funktion `ImageTexture::wrap_clamp` den Rand des Bereichs kopiert (Abbildung 3).
- e) **(4 Punkte)** Nähert man sich der Textur, wird man irgendwann deren Texel im Bild erkennen können ("*Magnification*"). Um diese Artefakte durch Vergrößerung zu verringern, sollen Sie in dieser Aufgabe bilineare Texturfilterung in `ImageTexture::evaluate_bilinear` implementieren. Interpolieren Sie die Werte der Textur an den angrenzenden Texeln bilinear, wie in Abb. 5 (links) gezeigt. Achtung: Die Werte in der Textur gelten für den Texelmittelpunkt. Beispielsweise ist $(0.5, 0.5)$ der Mittelpunkt des ersten Texels. Für $(s, t) = (1.5, 1.5)$ muss also genau der Wert zurückgegeben werden, der im Texel $(1, 1)$ gespeichert ist.
- f) **(6 Punkte)** Mipmapping kann Aliasing durch Verkleinerung (viele Texel werden auf einen Pixel abgebildet, "*Minification*") verringern. Dabei wird eine Auflösungspyramide für eine Textur erstellt, indem die Textur sukzessiv verkleinert wird und mehrere Texel dabei gemittelt werden. So werden die Texel der Stufen der Auflösungspyramide in Weltkoordinaten betrachtet immer größer. Beim Zugriff auf die Mipmap wählt man Stufen, in denen die Größe des



Abbildung 5: Links: Bilineare Filterung - Beim Zugriff auf den Punkt (s,t) wird bilinear zwischen den angrenzenden Texeln $(0,0)$, $(0,1)$, $(1,0)$ und $(1,1)$ interpoliert. Mitte und Rechts: Bilineare Filterung verbessert die Darstellung von Texturen in Bereichen nahe der Kamera.

Pixelfootprints ungefähr der Größe eines Texels entspricht. Der Pixelfootprint ist dabei die Fläche des Pixels, wenn man ihn auf die Oberfläche des geschnittenen Objekts projiziert.

Im ersten Teil der Aufgabe sollen Sie die AuflösungsPyramide erstellen. Berechnen Sie dazu in der Funktion `ImageTexture::create_mipmap` iterativ die $i + 1$ -te Stufe der Mipmap aus der i -ten Stufe ($i = 0, 1, \dots$), indem Sie Breite und Höhe halbieren und 2×2 Texelblöcke mitteln (Anmerkung: Wenn kein 2×2 Block mehr vorhanden ist, muss man 2×1 bzw. 1×2 Blöcke mitteln). Dies ist schematisch in Abbildung 7 (links) dargestellt. Die 0-te Stufe ist dabei die ursprüngliche Textur. Sie können annehmen, dass Breite und Höhe der Textur Zweierpotenzen sind. Der Vorgang endet, sobald Breite *und* Höhe der $i + 1$ -ten Mipmap-Stufe nur noch einen Texel betragen.

In der Methode `Object::compute_uv_aabb_size` sollen Sie die Größe der achsenorientierten Bounding Box (AABB) des Pixelfootprints im uv -Raum berechnen. Diese dient dann im nächsten Schritt als Größe des Pixelfootprint zur Auswahl der Mipmap-Stufen. Zum Bestimmen der AABB erhalten Sie den Schnittpunkt des Strahls durch den Pixelmittelpunkt `isect` und vier weitere Strahlen durch die Eckpunkte des Pixels. Schneiden Sie zuerst die vier Strahlen mit der Tangentenebene, die durch Position und Normale am Schnittpunkt definiert ist (Abbildung 7, Mitte unten). Speichern Sie die Schnittpunkte im Array `intersection_positions` und

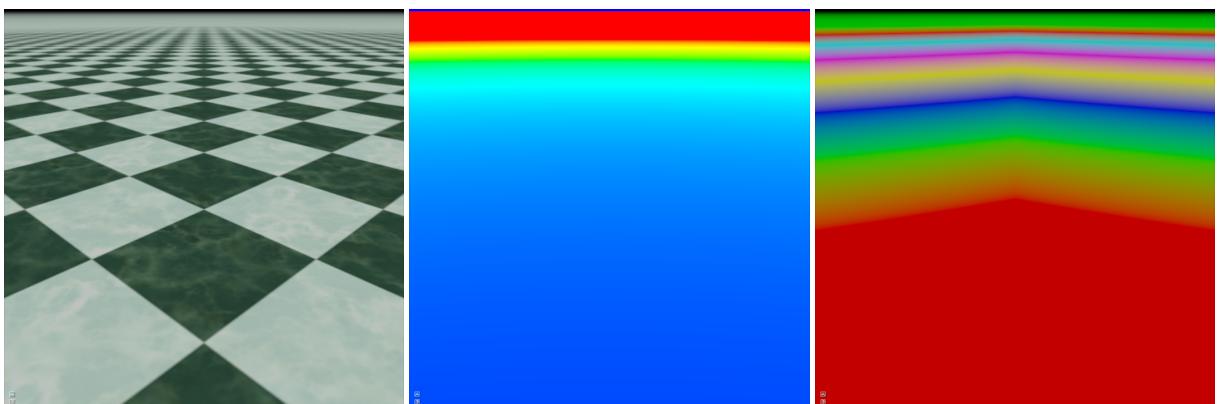


Abbildung 6: Szene mit trilateraler Filterung (links), Visualisierung der Größe des Pixelfootprints (Mitte) und Visualisierung der ausgewählten Mipmap Stufe (rechts).

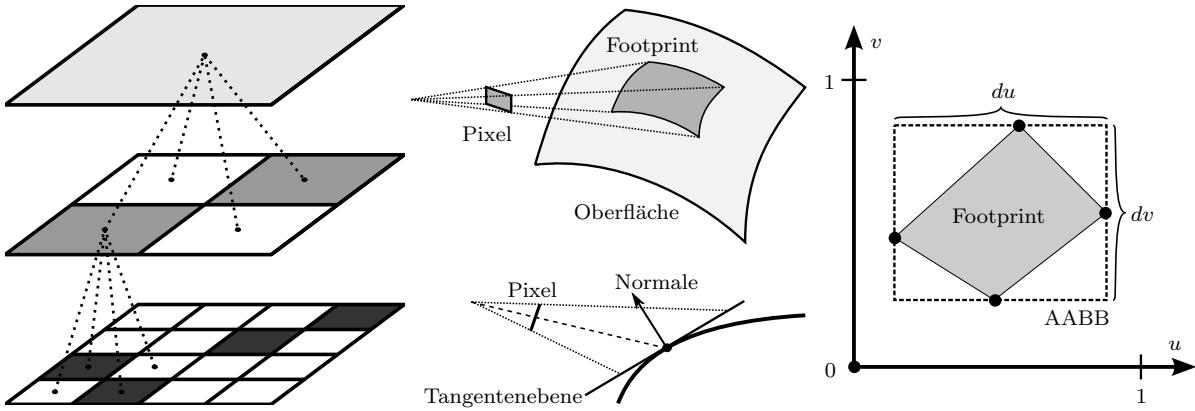


Abbildung 7: Schematischer Aufbau einer Mipmap (links), Illustration des Pixelfootprints (Mitte) und AABB des Footprints in uv -Koordinaten (rechts).

benutzen Sie für die Schnittpunktsberechnung die Funktion `intersect_plane`, welche in `intersection_tests.h` definiert ist. Falls einer der vier Strahlen die Tangentenebene nicht schneidet, so lassen Sie den entsprechenden Eintrag in `intersection_positions` unverändert.

Im nächsten Schritt, der schon implementiert ist, werden für die Schnittpunkte die jeweiligen uv -Koordinaten bestimmt. Berechnen Sie nun die AABB der uv -Koordinaten und geben Sie deren Größe (du, dv), wie in Abbildung 7 (rechts) dargestellt, zurück. In der GUI lässt sich der RenderMode `dudv` einstellen, der bei aktiverter trilinear Filterung die Werte (du, dv) visualisiert (Abbildung 6 Mitte).

Im letzten Teil der Aufgabe sollen Sie in der Methode `ImageTexture::evaluate_trilinear` trilineare Texturfilterung implementieren. Dabei müssen zwei geeignete Mipmap Stufen bestimmt werden, in denen jeweils bilinear gefiltert wird. Anschließend sollen die Ergebnisse der bilinearen Filterung nochmals linear interpoliert werden (daher der Name trilinear). Der Eingabeparameter `dudv` ist die im vorherigen Aufgabenteil berechnete Größe der AABB des Pixelfootprints in uv -Koordinaten. Transformieren Sie diese Größen in den st -Raum (bzgl. Mipmap-Stufe 0) und wählen Sie deren maximale Komponente S . Die Mipmap-Stufen ergeben sich durch ab- und aufrunden des Wertes

$$T = \log_2(S).$$

Verwenden Sie als Gewicht für die lineare Interpolation zwischen den Stufen den Nachkommanteil von T . Führen Sie in den zwei Stufen bilineare Interpolation durch und geben Sie schließlich das trilinear interpolierte Ergebnis zurück. Zum Testen haben wir Ihnen den Filtermodus `Debug Mip` zur Verfügung gestellt (Abbildung 6, rechts).

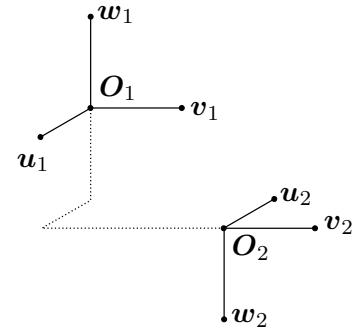
2 Transformationen (Theorie, keine Abgabe)

Gegeben sind zwei Koordinatensysteme mit den Ursprüngen

$$\mathbf{O}_1 = \begin{pmatrix} -2 \\ 1 \\ -1 \end{pmatrix} \quad \text{und} \quad \mathbf{O}_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

und den orthonormalen Basisvektoren

$$\begin{aligned} \mathbf{u}_1 &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} & \mathbf{v}_1 &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} & \mathbf{w}_1 &= \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \\ \mathbf{u}_2 &= \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} & \mathbf{v}_2 &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} & \mathbf{w}_2 &= \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}. \end{aligned}$$



Gesucht ist die *homogene* Matrix $M \in \mathbb{R}^{4 \times 4}$, die Koordinaten eines Punktes \mathbf{P}_1 , gegeben im System $S_1 = (\mathbf{O}_1, \mathbf{u}_1, \mathbf{v}_1, \mathbf{w}_1)$, in das System $S_2 = (\mathbf{O}_2, \mathbf{u}_2, \mathbf{v}_2, \mathbf{w}_2)$ transformiert:

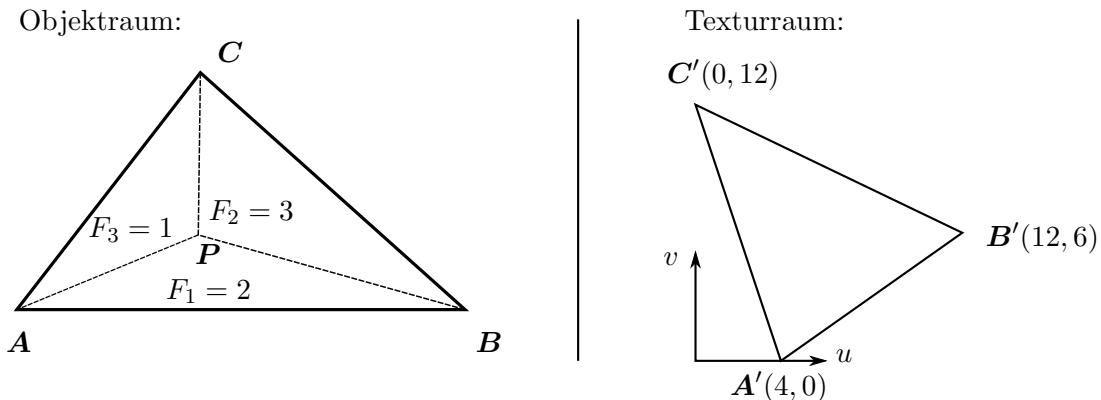
$$\mathbf{P}_2 = M \cdot \mathbf{P}_1.$$

Geben Sie die endgültige Transformationsmatrix M sowie den Rechenweg und die dabei benötigten Matrizen an! Beschreiben Sie stichpunktartig, welche Transformation diese Matrizen beschreiben! Matrixmultiplikationen (sofern vorhanden) müssen Sie nicht ausrechnen.

3 Baryzentrische Koordinaten im Dreieck (Theorie, keine Abgabe)

Gegeben ist ein Dreieck mit den Eckpunkten \mathbf{A} , \mathbf{B} und \mathbf{C} (siehe Abbildung). Den Eckpunkten sind die zweidimensionalen Texturkoordinaten \mathbf{A}' , \mathbf{B}' und \mathbf{C}' zugewiesen. Das Dreieck wird im Punkt \mathbf{P} von einem Strahl geschnitten. F_1 , F_2 und F_3 bezeichnen jeweils die Flächeninhalte der eingezeichneten Teildreiecke.

Berechnen Sie die baryzentrischen Koordinaten λ_A , λ_B und λ_C von \mathbf{P} und daraus die Texturkoordinaten $\mathbf{P}' = (u, v)$ des Punktes \mathbf{P} ! Geben Sie jeweils Ihren Rechenweg an!



Abgabe Laden Sie die Datei `exercise_03.cpp` in Ilias hoch. Der von Ihnen geschriebene Code sollte sich ausschließlich in den dafür vorgesehenen Funktionen in der Datei `exercise_03.cpp` befinden.

Framework Für jedes Übungsblatt stellen wir ein Framework bereit, das Sie im Ilias-Kurs herunterladen können. Das Framework nutzt C++11 und wird unter Linux getestet. Es ist allerdings auch unter Windows mit Visual Studio 2013 lauffähig. Das Framework enthält das Unterverzeichnis `cglib`. Weiterhin gibt es das aufgabenspezifische Unterverzeichnis `03_textures`, in dem Sie Ihre Lösung programmieren. Die Datei `Kompilieren.txt` enthält Informationen darüber, wie Sie das Framework kompilieren können.

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 jeweils 50% der Punkte aus den Praxisaufgaben von Block A (Blatt 1, 5, 6) und Block B (Blatt 2, 3, 4).
- Die theoretischen Aufgaben bedürfen *keiner* elektronischen Abgabe.
- Die Abgabe muss im Ordner `build` mit `cmake .. / && make` in der bereitgestellten VIRTUAL-Box VM¹ 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, insbesondere *nicht* die mitgelieferten Funktionsdeklarationen! Sie können allerdings in den abzugebenden Dateien Hilfsfunktionen definieren und benutzen.
- Sie dürfen sehr gerne untereinander die Aufgaben diskutieren, allerdings muss jede*r die Aufgaben *selbst* lösen, implementieren und abgeben. Plagiate bewerten wir mit 0 Punkten.
- Sie können sich bei Fragen an die Übungsleitung oder das Ilias-Forum wenden.

Alle Übungsleiter `cg_ws2425@lists.kit.edu`

Lucas Alber	<code>lucas.alber@kit.edu</code>
Nathan Lerzer	<code>nathan.lerzer@kit.edu</code>
Vincent Schüßler	<code>vincent.schuessler@kit.edu</code>

¹<https://cg.ivd.kit.edu/lehre/ws2023/cg/downloads/cgvm.zip>