BEISPIEL 5 - RASTERISIERUNG

VU Einführung in Visual Computing, TU Wien, SS 2021 10.05.2021

1 Übersicht

In diesem Beispiel sind drei klassische Computergraphik-Techniken zu implementieren: Line Rasterization, Fill Rasterization und Clipping. Diese wurden in der Vorlesung bereits besprochen und sollen nun in einer praktischen Übung mittels MATLAB implementiert werden.

1.1 Bewertung

Line rasterization :	15 Punkte
Clipping :	15 Punkte
Fill rasterization :	15 Punkte
Gesamt (ohne Bonus) :	**45 Punkte**
Rasterization vektorisiert (Bonus) :	10 Punkte (5 + 5)

1.2 Allgemeine Informationen zur Abgabe

Bitte stellen Sie sicher, dass MATLAB beim Ausführen der abgegebenen Dateien keine Fehlermeldungen ausgibt, das Skript nicht abstürzt oder ähnliches. **Wir können keinen fehlerhaften bzw. nicht ausführbaren Code bewerten!**

Es liegt in Ihrer Verantwortung, rechtzeitig **vor der Deadline** zu kontrollieren, ob Ihre Abgabe funktioniert hat. Sollte der Upload Ihrer Dateien aus irgendeinem Grund nicht funktionieren, posten Sie im TUWEL-Forum (sofern das Problem aktuell noch kein anderer Student gemeldet hat) und schreiben Sie eine E-Mail an **evc@cg.tuwien.ac.at**, damit wir schnellstmöglich darauf reagieren können.

Kontrollieren Sie Ihre Abgabe:

- Werden die hochgeladenen Dateien angezeigt?
- Können die Dateien herunterladen und geöffnet oder im Browser anzeigt werden?
- Sind die richtigen Dateien abgegeben?

Packen Sie für die Abgabe alle MATLAB Dateien des Frameworks in ein ZIP-Archiv.

2 Erinnerung: Graphics Pipeline

In diesem einleitenden Kapitel soll die Graphics Pipeline noch einmal in Erinnerung gerufen, sowie die einzelnen Stellen, die für diese Übung wichtig sind, kurz erklärt werden. Eine visuelle Kurzform der Pipeline ist in Abbildung 1 dargestellt.

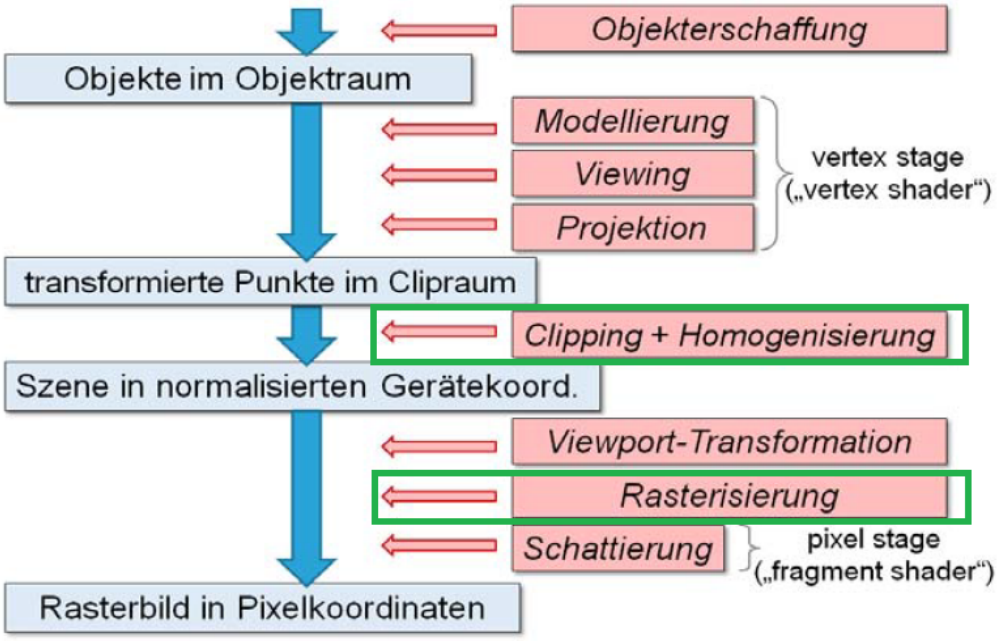


Abbildung 1: Visuelle Kurzform der Graphics Pipeline wie in der Vorlesung besprochen.

Die beiden in grün eingerahmten Schritte werden dabei in dieser Aufgabe selbst implementiert. Zunächst wird eine einfache Variante der Rasterisierung - die *Line Rasterization* - implementiert, um erste visuelle Ergebnisse zu erhalten. Die zweite Aufgabe stellt *Clipping im Clip Space* dar. Zuletzt soll noch die sogenannte *Fill Rasterization* implementiert werden, um nicht nur die Kanten (Edges) sondern auch ganze Flächen (Faces) eines Meshes zu zeichnen.
Da die Rasterisierung nach der Projektion, Homogenisierung und Viewport-Transformation stattfindet, arbeiten wir in diesem Schritt mit echten Pixelkoordinaten der einzelnen Vertices. Abhängig von der Art der Rasterisierung werden Linien bzw. Dreiecke in einen Framebuffer rasterisiert (gezeichnet).

EXKURS: Framebuffer bezeichnet eine abstrakte Darstellung aller Informationen, die notwendig sind, um ein Bild am Bildschirm darzustellen, oft auch der Einfachheit halber als Sammlung von Bildern bezeichnet. Dazu gehört meistens ein sogenannter Color-Buffer, der das eigentliche Bild enthält, aber auch oft ein Depth-Buffer (meist auch Z-Buffer genannt), der Tiefenwerte speichert.

Zwischen Projektion und Homogenisierung befinden wir uns im sogenannten Clip Space, in dem Teile eines Meshes, die sich außerhalb des sichtbaren Bereichs befinden, weggeschnitten werden (Clipping). Der Clip Space ist ein homogener Raum (w-Koordinate $\neq 1$). In diesem Raum wird ein Clipping Volume definiert, bestehend aus 6 Ebenen (oben, unten, vorne, hinten, links, rechts). Dieses Clipping Volume beschränkt den kontinuierlichen Raum auf den im Endeffekt sichtbaren Bereich.

3 MATLAB Rasterization Framework

Für die folgenden Aufgaben wird ein MATLAB Framework zur Verfügung gestellt, das 3D Meshdaten (im .PLY Format, siehe Unterordner *data*) lesen und in eine Datenstruktur, wie in Mesh Datenstruktur beschrieben, speichern kann. Weiters wird eine GUI zur Verfügung gestellt, um nicht nur die rasterisierten Daten anzuzeigen, sondern auch schnell neue Daten laden bzw. zwischen den Rasterisierungsmodi wechseln zu können. Im Folgenden wird auf dieses Framework näher eingegangen, um einen angenehmen Start zu ermöglichen.

3.1 GUI

Um das Framework zu starten, muss lediglich das Skript main_GUI.m ausgeführt werden. Anschließend erscheint ein Fenster wie in Abbildung 2 illustriert. Bei Verwendung von High DPI Scaling (Bildschirmvergrößerung) in Windows 10 kann es zu Artefakten in der GUI Darstellung kommen. Dies kann entweder durch Zurücksetzen dieses Faktors auf 100% behoben werden oder durch Rechtsklick auf MATLAB.exe -> Eigenschaften -> Reiter Kompatibilität -> Aktivieren von *Skalierung bei hohem DPI-Wert deaktivieren*.
Falls das Starten über die Konsole gewünscht ist oder eine zu alte Matlab Version verwendet wird, kann das Framework auch händisch über den Aufruf von main.m ausgeführt werden. Hier müssen die beiden Parameter *model* und *rasterization_mode* wie bei einer Funktion übergeben werden, was z.B. wie folgt aussieht: `main('data/plane.ply', 'line')`

Wichtig: Die zur Verfügung gestellte GUI funktioniert erst ab der Matlab Version 2015.

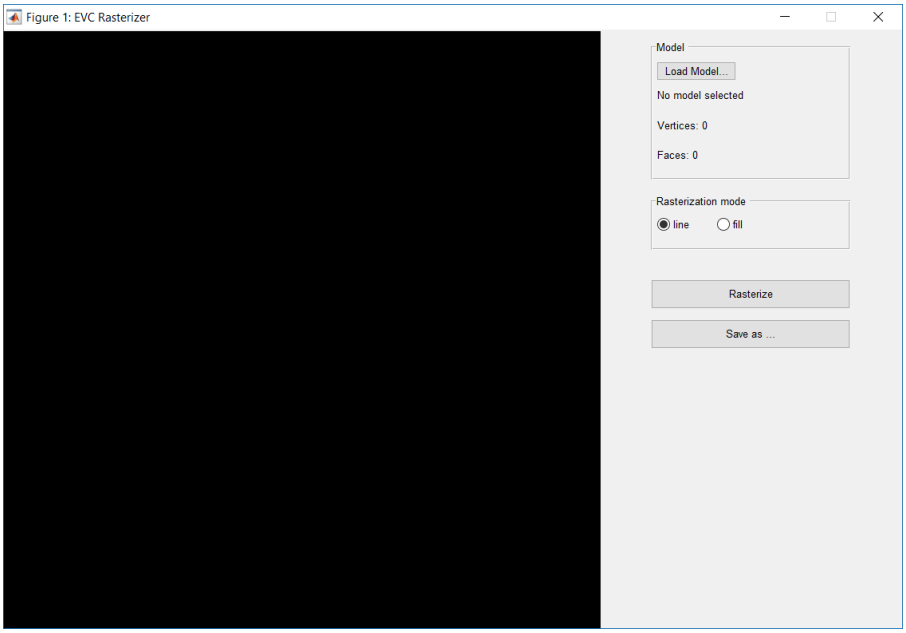


Abbildung 2: GUI nach Start des Frameworks. In der linken Hälfte ist die Anzeige für das rasterisierte Mesh zu sehen, auf der rechten Seite Kontrollelemente um ein neues Mesh zu laden und zwischen den Rasterisierungsmodi hin und her zu wechseln bzw. die Rasterisierung zu starten. Weiters kann das aktuell rasterisierte Mesh als Bild abgespeichert werden.

3.2 Mesh Datenstruktur

Theorie in Vorlesung: *Graphikpipeline + Objektrepräsentation*

Um alle für die Rasterisierung und fürs Clipping notwendigen Informationen zu speichern, wird eine Mesh Datenstruktur zur Verfügung gestellt. Diese speichert eine Liste an Faces (anfangs Dreiecke, nach Clipping zusätzlich Polygone) mit zugehörigen Vertices. Jeder Vertex hat wiederum Information über seine Position im Raum, seine Farbe sowie bei der Rasterisierung die Pixelkoordinaten gespeichert.
Hinweis: Eine detailliertere Beschreibung über den Umgang mit der Datenstruktur in MATLAB ist in der Datei tutorial.m aufgeführt.
Die Klasse *Mesh* erlaubt den Zugriff auf einzelne Faces über die Funktion *getFace(index)*. Dabei wird ein Objekt der Klasse *MeshFace* zurückgeliefert, mit dem wiederum der Zugriff auf einzelne Vertices möglich ist. Die Methode *getVertex(index)* liefert einen *MeshVertex* zurück, der mit den Methoden *getPosition()*, *getColor()* und nach dem Clipping *getScreenCoordinates()* die Information pro Vertex bereitstellt.
Der Zugriff auf die einzelnen Faces/Vertices erfolgt via Indizes. Diese können entweder skalar sein, um auf ein spezifisches Element zugreifen zu können, aber auch vektorwertig. Letzteres ermöglicht den Zugriff auf mehrere Faces/Vertices gleichzeitig. Genaueres hierzu findet sich im tutorial.m.

3.3 Testdaten

Zum Testen der Aufgaben in dieser Übung sind im Ordner *data* Meshes im PLY[1] Format zu finden. Diese Meshes können grundsätzlich für alle Aufgaben herangezogen werden, jedoch gehen Testdaten mit dem Präfix *clipped_* über den sichtbaren Bereich hinaus und würden daher ohne Clipping Fehler produzieren. Die folgende Liste gibt daher einen kurzen Überblick welche Meshes für welche Aufgaben sinnvoll sind:

Line Rasterization	plane.ply star.ply text.ply torus.ply
Clipping	siehe Line Rasterization clipped_star.ply clipped_torus.ply
Fill Rasterization	siehe Line Rasterization siehe Clipping

Hinweis: Zusätzlich zu diesen Daten sind Musterlösungen (Referenzbilder) im Ordner *reference* enthalten. Diese können allerdings nur als grobe Referenz herangezogen werden, da das selbst erzeugte Bild eine höhere Präzision (8 bytes) besitzt, als die Referenzbilder (1 byte).

[1] https://en.wikipedia.org/wiki/PLY_(file_format)

4 Line Rasterization

Theorie in Vorlesung: *Rasterisierung*

Die erste Aufgabe behandelt die Rasterisierung von Linien mit Hilfe des Digital Differential Algorithmus (kurz DDA). Nachdem die Vertex-Positionen verschiedene Schritte der Grafikpipeline durchwandern (Transformation in World/View Space, Perspektivische Projektion, etc.) kommen diese am Ende bei der Rasterisierung an. Dabei werden die auf 2D projizierten Positionen mit Linien verbunden und ein sogenanntes **Wireframe-Modell**[2] des Objektes gezeichnet.

Laden Sie die Dateien linerasterization.m und MeshVertex.m. Die folgende Angabe findet sich auch in Form von (englischen) Kommentaren im MATLAB-File wieder. Die Stellen, an denen Ihre Lösung implementiert werden soll, sind jeweils mit einem *TODO 1* markiert. Sie sollten bei der Implementierung der folgenden Aufgaben stets sicher sein, dass Sie genau wissen, was in welcher Zeile passiert. **Beim Abgabegespräch müssen Sie Ihren abgegebener Code erklären können und selbst wenn die Lösung zu 100% richtig ist, werden nur dann Punkte vergeben, wenn Sie die Abgabe einwandfrei erklären können!**

4.1 Allgemeine Hinweise

Die Methodensignaturen dürfen auf keinen Fall verändert werden!

4.2 DDA Algorithmus

Digital Differential Algorithmus (DDA) ist ein Verfahren um Linien von einem kontinuierlichen Raum (Raum-Koordinaten) in einen diskreten (Pixel-Koordinaten) zu transformieren. In unserem konkreten Fall bedeutet das, dass wir zwischen allen Punkte-Paaren eines Dreiecks eine Linie zeichnen. Die genaue Funktionsweise des Algorithmus wurde bereits in der Vorlesung behandelt bzw. kann auf Wikipedia[3] unter “Einfache Methoden” dies noch einmal nachgelesen werden.
Achtung: Es wird dringend empfohlen auch das nachfolgende Kapitel “Verallgemeinerung auf beliebige Richtungen” zu lesen, da das naive Verfahren für diese Aufgabe nicht ausreichend ist!
Die Implementierung des Algorithmus findet in der Datei linerasterization.m statt. Dabei ist die grundlegende Methode, die für jedes Vertex Paar den Algorithmus zur Rasterisierung aufruft, bereits implementiert. Das Zeichnen selbst allerdings muss noch in der darunter liegenden Funktion *drawLine(...)* implementiert werden. Dieser Funktion werden zwei Vertices übergeben zwischen denen eine Linie in den verfügbaren Framebuffer geschrieben werden soll. Bei nicht ganzzahligen Koordinaten während der Linienrasterisierung muss kaufmännisch mit der Funktion *round(...)* gerundet werden. Die Pixel der Linie können zu Testzwecken mit der Farbe Weiß und der Tiefe 0 gesetzt werden. Schließlich müssen aber auch Tiefe und Farbe zwischen zwei Endpunkten interpoliert werden. Anschließend wird der beschriebene Framebuffer zurückgeliefert. Ein Beispielbild dazu ist in Abbildung

[2] Wireframe-Modell: https://de.wikipedia.org/wiki/Drahtgittermodell
[3] Rasterisierung: https://de.wikipedia.org/wiki/Rasterung_von_Linien

3 dargestellt.

Hinweis: Wie auf Eigenschaften/Funktionen eines Objektes im Framework zugegriffen wird (z.B. Position/Farbe eines Vertex), ist in oberem Kapitel Mesh Datenstruktur beschrieben.

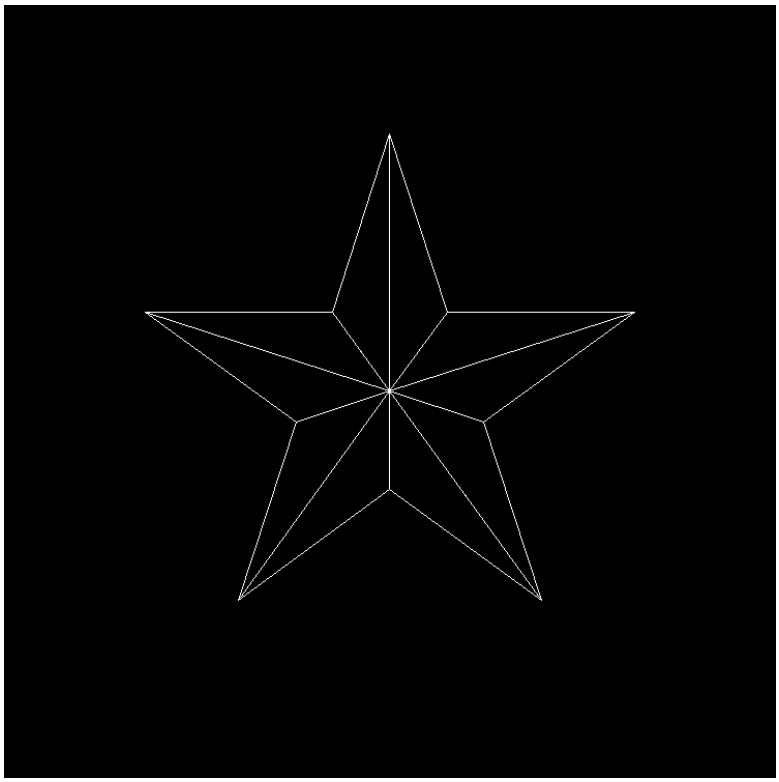


Abbildung 3: Linien von star.ply rasterisiert ohne Farbe.

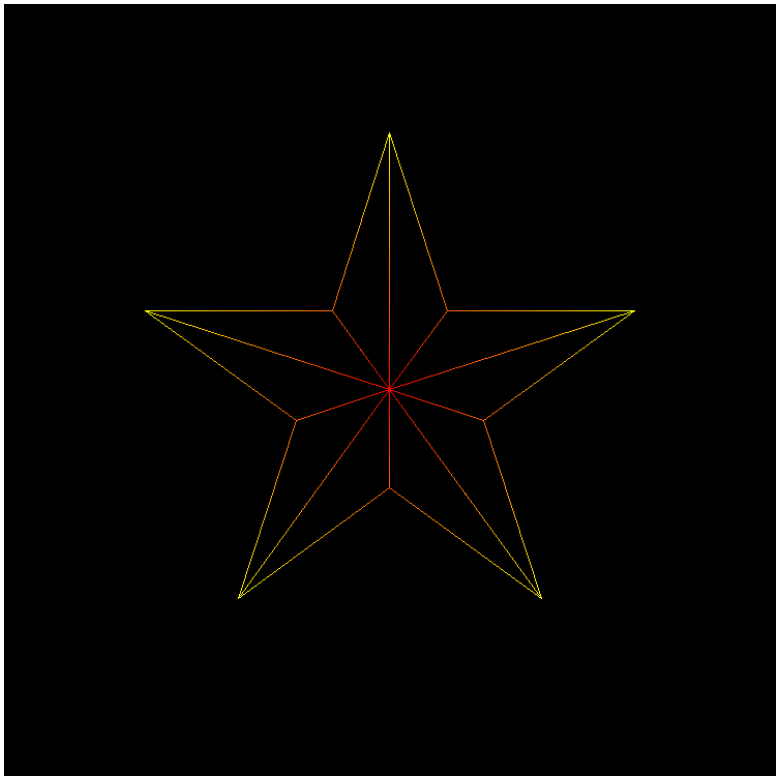


Abbildung 4: Linien von star.ply rasterisiert mit interpolierter Farbe.

4.3 Farb- und Tiefeninterpolation

Da Vertices auch Farbinformation besitzen, ist es sinnvoll, diese mit in die Rasterisierung einfließen zu lassen. Ein Beispiel hierfür ist in Abbildung 4 zu sehen. Ebenso müssen die Tiefenwerte der Vertices interpoliert werden. Haben die zwei Endpunkte (Vertices) einer Linie unterschiedliche Farben oder Tiefenwerte, so wird zwischen diesen linear interpoliert:

$$v_{new} = v_1 \cdot (1 - t) + v_2 \cdot t \tag{1}$$

v_1 und v_2 stellen hier beliebige Datentypen dar (in unserem Fall Vektoren, die Farbe bzw. Position beschreiben). Der Interpolationskoeffizient t gibt an welchen Anteil das Resultat von den jeweiligen Werten haben soll.
ACHTUNG: Dieser Koeffizient t kann ausschließlich Werte aus dem abgeschlossenen Intervall $[0, 1]$ annehmen.
Um diesen Koeffizienten zu berechnen, kann die Strecke zwischen dem ersten Endpunkt und dem aktuell zu rasterisierenden Pixel durch die Gesamtstrecke der Linie dividiert werden. Das Ergebnis beschreibt dann die normalisierte Distanz des aktuellen Pixels vom ersten Endpunkt.
Hinweis: Die zu implementierende Funktion *MeshVertex.mix(...)* finden Sie in der Datei MeshVertex.m. Die Funktion *MeshVertex.mix(...)* kann anschließend verwendet werden um Farb- und Tiefeninformation zwischen den beiden Endpunkten mit dem ermittelten Koeffizienten t zu interpolieren. Die Farbe eines Endpunktes kann mit der Funktion *getColor()* abgefragt werden.

4.4 Bonus - vektorisiert

Da MATLAB Matrix Operationen wesentlich schneller durchführen kann als Schleifendurchläufe (for, while, etc.), macht es in vielen Situationen Sinn, Algorithmen zu vektorisieren. Das bedeutet, dass Operationen, die üblicherweise in einer Schleife durchgeführt werden würden, in eine ganz bestimmte Vektor-/Matrixform transformiert werden und anschließend nur Matrixoperationen angewandt werden. Dies führt teilweise zu unglaublich schneller Performance.
In diesem Bonus-Beispiel geht es darum, die Rasterisierung von Linien (die Funktion *drawLine(...)*) zu vektorisieren. Dafür dürfen **keine** Schleifen (for, while, do..while) und **keine** Funktionen, welche Schleifen emulieren (z.B. arrayfun), verwendet werden! **Hinweise:**

- Die Schrittweiten aller Punkte auf der Linie können in einem Vektor gespeichert werden und anschließend zur Interpolation genutzt werden.
- Bei der Funktion *framebuffer.setPixel()* können auch Vektoren und Matrizen als Inputparameter verwendet werden.

4.5 Hinweise für das Abgabegespräch

Beim Abgabegespräch müssen Sie in der Lage sein Ihren Code zu erklären und Sie müssen genau wissen, was die verwendeten Befehle tun. Auch die Theorie hinter den nachprogrammierten Befehlen

kann gefragt werden und muss erklärt werden können (z.B. Was macht der DDA Algorithmus? Wie wird über eine Linie iteriert? etc.).

5 Clipping

Theorie in Vorlesung: *Clipping und Antialiasing*

In dieser Aufgabe geht es darum, Clipping für Polygone anhand des **Sutherland-Hodgman Algorithmus** zu implementieren. Oft kommt es vor, dass ein Mesh nur teilweise oder gar nicht sichtbar ist und daher ist es sinnvoll, die Rasterisierung auf die Teile einzuschränken, die sichtbar sind. Beim Clipping werden all jene Teile eines Meshes weggeschnitten, die sich außerhalb des Bildschirms befinden. Dies kann im Clip Space getestet werden, indem alle Kanten eines Meshes mit dem Clipping Volumen geschnitten werden. Dabei werden Kanten, die sich ganz außerhalb befinden, weggeschnitten und Kanten, die eine Ebene schneiden, zerteilt. Details zu Clipping und zum Algorithmus finden sich in den Vorlesungsfolien zum Thema *Clipping + Antialiasing*.

Laden Sie die Datei ClippingPlane.m um die Clipping Planes, welche die Grenzen des Clipping Volumens darstellen, zu definieren. Implementieren Sie anschließend in der Datei clip.m die Funktion *clipPlane(...)*, welche ein Face mit einer Plane schneiden soll. Dabei ist zu beachten, dass sich die Anzahl der Facevertices durch das Clipping ändern kann. Dieser Vorgang wird für alle Faces mit jeweils allen Clipping Planes durchgeführt.
Hinweis: Die Vertices eines Face sind in der Datei clip.m in der Matrix *positions* gespeichert. Zwischen je zwei aufeinanderfolgenden Vertices in dieser Matrix (Achtung: auch zwischen dem letzten und ersten Eintrag) besteht eine Kante. Für diese Kanten soll das Clipping durchgeführt werden.

Die dazugehörige Angabe findet sich auch in Form von (englischen) Kommentaren im MATLAB-File wieder. Die Stellen, an denen Ihre Lösung implementiert werden soll, sind jeweils mit einem *TODO 2* markiert. Sie sollten bei der Implementierung der folgenden Aufgaben stets sicher sein, dass Sie genau wissen, was in welcher Zeile passiert. **Beim Abgabegespräch müssen Sie Ihren abgegebenen Code erklären können und selbst wenn die Lösung zu 100% richtig ist, werden nur dann Punkte vergeben, wenn Sie die Abgabe einwandfrei erklären können!**

5.1 Allgemeine Hinweise

Die Methodensignatur darf auf keinen Fall verändert werden!

5.2 Clipping planes

Wie bereits beschrieben, basiert Clipping auf einem Vertex-Plane-Test, d.h. es wird überprüft, ob ein Vertex vor, auf oder hinter einer Ebene (Plane) liegt. Dazu wird im Framework die Klasse ClippingPlane.m zur Verfügung gestellt in der die beiden wichtigen Funktionen *inside(...)* und *intersect(...)* allerdings erst implementiert werden müssen. Außerdem müssen die 6 notwendigen Ebenen in der statischen Methode *getClippingPlanes()* definiert werden. Diese Ebenen sollen das Clipping Volume beschreiben. In unserem Fall ist das Clipping Volume im Clip Space ein Würfel mit der Sei-

tenlänge 2, dessen Zentrum sich im Ursprung befindet.
Eine Ebene $\vec{p}$ wird in Hessescher Normalform[4] dargestellt und kann damit in einem 4-Komponenten Vektor gespeichert werden, wobei die ersten 3 Komponenten dem Normalvektor (Richtung) der Ebene und die letzte der negativen Distanz zum Ursprung entspricht. Dies basiert auf folgender Grundlage:
Alle Punkte $\vec{v}$, die folgende Gleichung erfüllen, liegen auf der Ebene mit Normalvektor $\vec{n}$ und Distanz zum Ursprung d (Im Folgenden wird mit $\cdot$ das Skalarprodukt bezeichnet):

$$\begin{aligned} \vec{v} \cdot \vec{n} - d &= 0 \\ (v_x, v_y, v_z) \cdot (n_x, n_y, n_z) - d &= 0 \end{aligned} \tag{2}$$

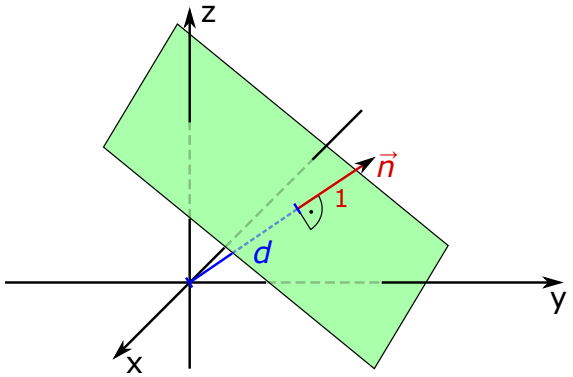


Abbildung 5: Visuelle Darstellung der Komponenten der Hesschen Normalform.[5]

Diese Gleichung kann auch als 4D Skalarprodukt ausgedrückt werden:

$$(v_x, v_y, v_z, 1) \cdot (n_x, n_y, n_z, -d) = 0 \tag{3}$$

Hinweis: Das Skalarprodukt zwischen der Ebene und einem Punkt kann auch als die Distanz des Punktes zur Ebene in Abhängigkeit des Normalvektors interpretiert werden. Beachte die Richtung des Normalvektors! Mit diesem Wissen kann ein gegebener Punkt im Raum einfach gegen diese Ebene getestet werden:

1. Ein Punkt $\vec{v}$ liegt innerhalb einer Ebene $\vec{p}$ wenn: $\vec{v} \cdot \vec{p} < 0$
2. Ein Punkt $\vec{v}$ liegt auf einer Ebene $\vec{p}$ wenn: $\vec{v} \cdot \vec{p} = 0$
3. Ein Punkt $\vec{v}$ liegt außerhalb einer Ebene $\vec{p}$ wenn: $\vec{v} \cdot \vec{p} > 0$

Hinweis: Die Funktion *inside(...)* der Klasse *ClippingPlane* inkludiert auch Punkte, die auf der Ebene liegen!

Wird nun eine Kante mit einer Ebene geschnitten, so möchte man genau den Schnittpunkt berechnen und als neuen Vertex hinzufügen (siehe Sutherland-Hodgman Algorithmus). Der Schnittpunkt kann

[4] https://de.wikipedia.org/wiki/Hessesche_Normalform
[5] Siehe https://de.wikipedia.org/wiki/Datei:Plane_equation_qtl4.svg

als lineare Kombination der beiden Endpunkte $\vec{a}$ und $\vec{b}$ der Kante berechnet werden:

$$((1-t)\cdot\vec{a}+t\cdot\vec{b})\cdot\vec{p}=0 \tag{4}$$

Der Interpolationskoeffizient t soll nun in der Funktion *intersect(...)* wie folgt berechnet werden (diese Formeln ergeben sich aus der Gleichung 4):

$$\begin{aligned} l_a &= \vec{a}\cdot\vec{p} \\ l_b &= \vec{b}\cdot\vec{p} \\ t &= \frac{l_a}{l_a - l_b} \end{aligned} \tag{5}$$

Existiert ein Schnittpunkt, so ist es aufgrund von numerischen Ungenauigkeiten oft sinnvoll, t um einen kleinen Betrag zu verändern (also den Schnittpunkt nach innen zu verschieben).

Dabei wird entweder:

- t um 1×10^{-6} verringert, falls $\vec{a}$ innerhalb der Ebene ist oder
- t um 1×10^{-6} erhöht, falls $\vec{b}$ innerhalb der Ebene ist.

Wichtig: Zusätzlich zur Position soll auch Farbe und Tiefe für den neuen Punkt interpoliert werden.

5.3 Hinweise für das Abgabegespräch

Beim Abgabegespräch müssen Sie in der Lage sein Ihren Code zu erklären und Sie müssen genau wissen, was die verwendeten Befehle tun. Auch die Theorie hinter den nachprogrammierten Befehlen kann gefragt werden und muss erklärt werden können (z.B. Warum benötigt man Clipping? Wie sieht das Clipping Volume aus? etc.).

6 Fill Rasterization

Theorie in Vorlesung: *Polygonfüllen*

Im Normalfall will man nicht die Kanten eines Meshes rasterisieren, sondern seine Oberfläche. Dazu werden in dieser Aufgabe Dreiecke statt Linien rasterisiert.
Der erste Schritt besteht darin, jene Pixel zu finden, die sich innerhalb eines Dreiecks befinden. Dann werden die Farbwerte der Endpunkte abhängig von der Position innerhalb des Dreiecks interpoliert und schließlich im Framebuffer gespeichert.
Die Implementierung ist in den Dateien fillrasterization.m und MeshVertex.m vorzunehmen.

Laden Sie die Dateien fillrasterization.m und MeshVertex.m. Die folgende Angabe findet sich auch in Form von (englischen) Kommentaren im MATLAB-File wieder. Die Stellen, an denen Ihre Lösung implementiert werden soll, sind jeweils mit einem *TODO 3* markiert. Sie sollten bei der Implementierung der folgenden Aufgaben stets sicher sein, dass Sie genau wissen, was in welcher Zeile passiert. **Beim Abgabegespräch müssen Sie Ihren abgegebenen Code erklären können und selbst wenn die Lösung zu 100% richtig ist, werden nur dann Punkte vergeben, wenn Sie die Abgabe einwandfrei erklären können!**

6.1 Allgemeine Hinweise

Die Methodensignatur darf auf keinen Fall verändert werden!

6.2 Bedeckte Flächen identifizieren

Eine einfache Methode jene Pixel zu finden, die sich innerhalb eines Dreiecks befinden, wäre jeden Pixel gegen jede Kante des Dreiecks zu testen. Ist der Punkt innerhalb aller drei Kanten, so ist er auch innerhalb des Dreiecks. Dieser Test ist sehr ähnlich zu dem in Clipping verwendeten Test, ob sich ein Punkt im sichtbaren Bereich befindet. Allerdings werden hier Geradengleichungen statt Ebenengleichungen verwendet. Implementieren Sie folgende Funktionalität in den Funktionen *drawTriangle(...)* und *lineEq(...)* in der Datei fillrasterization.m.

Die Seiten des Dreiecks werden in dieser Aufgabe als **Geradengleichungen** mit Hilfe der (allgemeinen) Form $A \cdot x + B \cdot y + C = 0$ ausgedrückt. In der Folge werden diese mit O_1, O_2 und O_3 bezeichnet. Die Parameter A und B entsprechen den Komponenten der Normalvektoren zu den Kantenvektoren des Dreiecks. Der Parameter C beschreibt jeweils den Abstand der einzelnen Geraden zum Ursprung.

Diese Parameter A, B und C müssen für jede der drei Geraden (Seiten des Dreiecks) berechnet werden. Daraus ergeben sich die Geradengleichungen:

$$
\begin{aligned}
O_1(x,y) &= A_1 \cdot x + B_1 \cdot y + C_1 \\
O_2(x,y) &= A_2 \cdot x + B_2 \cdot y + C_2 \\
O_3(x,y) &= A_3 \cdot x + B_3 \cdot y + C_3
\end{aligned} \tag{6}
$$

Betrachten wir einen Punkt mit den Koordinaten (x, y) dann liegt dieser innerhalb des Dreiecks wenn er für **alle drei** Geradengleichungen die folgende Bedingung erfüllt:

$$
O_i(x,y) \leq 0 \tag{7}
$$

Um die Geradengleichungen für ein Dreieck aufzustellen, berechnen wir zuerst die Kantenvektoren e_i. Die Eckpunkte des Dreiecks werden hier als V_1, V_2 und V_3 bezeichnet.

$$
\begin{aligned}
e_1 &= V_3 - V_2 \\
e_2 &= V_1 - V_3 \\
e_3 &= V_2 - V_1
\end{aligned} \tag{8}
$$

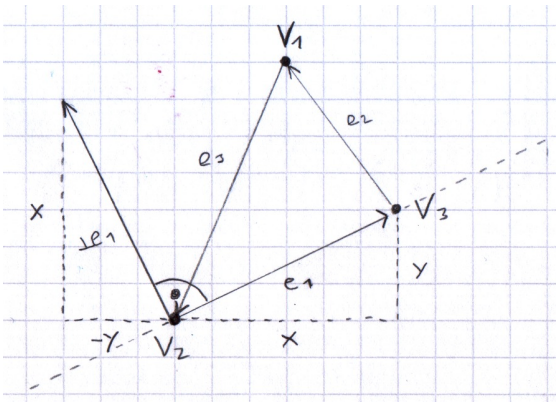
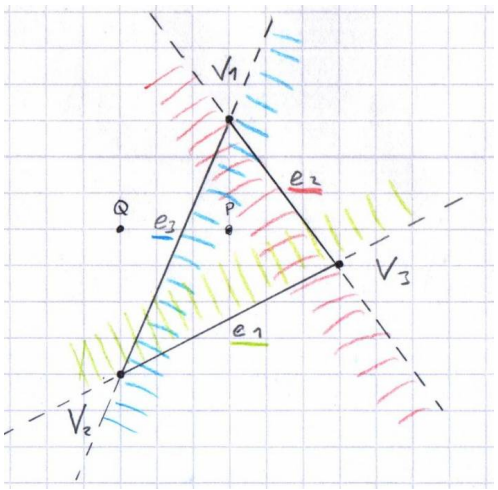


Abbildung 6: Normalvektoren der Kanten.

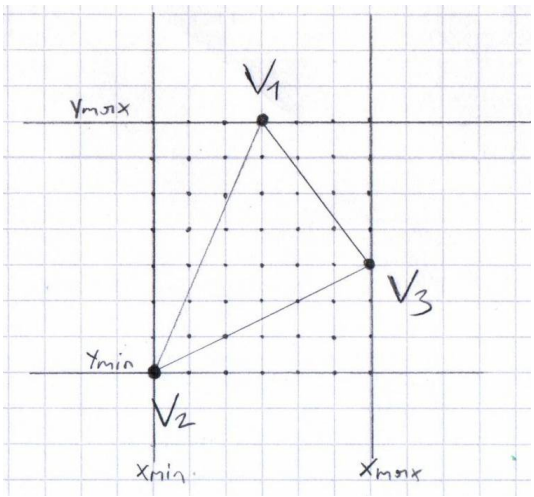
Anschließend wird der Normalvektor zu diesen Kanten ermittelt (siehe Abbildung 6):

$$
\begin{aligned}
e_i &= (x, y) \\
\perp e_i &= (-y, x)
\end{aligned} \tag{9}
$$

Die Normalvektoren stehen orthogonal (senkrecht) zu den jeweiligen Kantenvektoren und entsprechen einer orthogonalen Kante $\perp\ e_i$. Die Komponenten dieser Normalvektoren entsprechen den



(a) Geradengleichungen eines Dreiecks.



(b) Bounding Box eines Dreiecks.

Abbildung 7

Parametern A_i und B_i der Geradengleichungen.

$$\begin{aligned} A_i &= -e_{i,y} \\ B_i &= e_{i,x} \end{aligned} \tag{10}$$

Zuletzt wird noch für jede Gerade deren Abstand zum Ursprung berechnet, indem jeweils ein darauf liegender Punkt in O_i eingesetzt und nach C_i umgestellt wird:

$$\begin{aligned} C_1 &= -(A_1 \cdot V_{2,x} + B_1 \cdot V_{2,y}) \\ C_2 &= -(A_2 \cdot V_{3,x} + B_2 \cdot V_{3,y}) \\ C_3 &= -(A_3 \cdot V_{1,x} + B_3 \cdot V_{1,y}) \end{aligned} \tag{11}$$

Die resultierenden Geradengleichungen sind in Abbildung 7a farblich dargestellt. Nun kann jeder Punkt, wie in Gleichung 7 beschrieben, getestet werden, ob er sich innerhalb oder außerhalb eines gegebenen Dreiecks befindet.
Da für ein Dreieck allerdings nicht jeder Pixel des Framebuffers in Frage kommt, ist es sinnvoll nur jene Pixel zu testen, die möglicherweise innerhalb des Dreiecks sein könnten. Hierzu berechnet man die sogenannte **Bounding Box** des Dreiecks (siehe Abbildung 7b) indem die minimalen und maximalen x- bzw. y-Koordinaten der Eckpunkte V_1, V_2 und V_3 ermittelt werden. Danach werden nur jene Pixel getestet, die sich innerhalb dieser Bounding Box befinden.

6.3 Baryzentrische Koordinaten

Baryzentrische Koordinaten sind ein hilfreiches Werkzeug zur Parametrisierung von Dreiecken. Mit Hilfe von 3 verschiedenen Koeffizienten, die in Summe 1 ergeben, kann jeder Punkt auf der Oberfläche des Dreiecks repräsentiert werden. Die Namenskonvention für diese Koeffizienten ist dabei α, β und γ für die gilt: $\alpha + \beta + \gamma = 1$. Jeder Punkt P auf der Oberfläche kann als Linearkombination der 3 Eckpunkte dargestellt werden: $P = \alpha \cdot V_1 + \beta \cdot V_2 + \gamma \cdot V_3$. Weiters kann mit Hilfe von baryzentrischen Koordinaten die Farbe des Punktes P in der selben Art und Weise ermittelt werden. Diese Koeffizienten sind in Abbildung 8 illustriert.

Die baryzentrischen Koordinaten eines Punktes P können mit den zuvor ermittelten Geradengleichungen berechnet werden:

$$\begin{aligned} \alpha(P) &= \frac{O_1(P)}{O_1(V_1)} \\ \beta(P) &= \frac{O_2(P)}{O_2(V_2)} \\ \gamma(P) &= \frac{O_3(P)}{O_3(V_3)} \end{aligned} \tag{12}$$

Da der Nenner dieser Brüche pro Dreieck konstant bleibt, kann dieser Teil vorberechnet werden (hier exemplarisch

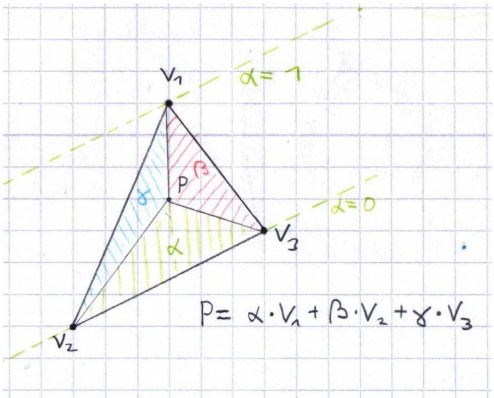


Abbildung 8: Baryzentrische Koordinaten eines Punktes P visualisiert.

für O_1) :

$$f_1 = \frac{1}{O_1(V_1)}$$
$$\alpha(P) = O_1(P) \cdot f_1 \tag{13}$$

Hinweis: Die Interpolation mit Hilfe der baryzentrischen Koordinaten ist in der Funktion *MeshVertex.barycentricMix(...)* in der Datei MeshVertex.m vorzunehmen. Die Funktion *MeshVertex.barycentricMix(...)* kann anschließend verwendet werden um Farb- und Tiefeninformation zwischen den drei Endpunkten mit den ermittelten Koeffizienten α, β und γ zu interpolieren.

6.4 Bonus - vektorisiert

Da MATLAB Matrix Operationen wesentlich schneller durchführen kann als Schleifendurchläufe (for, while, etc.) macht es in vielen Situationen Sinn, Algorithmen zu vektorisieren. Das bedeutet, dass Operationen, die üblicherweise in einer Schleife durchgeführt werden würden, in eine ganz bestimmte Vektor-/Matrixform transformiert werden und anschließend nur Matrixoperationen angewandt werden. Dies führt teilweise zu unglaublich schneller Performance.
In diesem Bonus-Beispiel geht es darum, die Rasterisierung von Dreiecken (die Funktion *drawTriangle(...)*) zu vektorisieren. Dafür dürfen **keine** Schleifen (for, while, do..while) und **keine** Funktionen, welche Schleifen emulieren (z.B. arrayfun), verwendet werden!
Hinweise:

- Die Liniengleichungen können auf alle Pixel der Bounding Box eines Dreiecks gleichzeitig angewendet werden.
- Bei der Funktion *framebuffer.setPixel()* können auch Vektoren und Matrizen als Inputparameter verwendet werden.

6.5 Hinweise für das Abgabegespräch

Beim Abgabegespräch müssen Sie in der Lage sein ihren Code zu erklären und Sie müssen genau wissen, was die verwendeten Befehle tun. Auch die Theorie hinter den nachprogrammierten Befehlen kann gefragt werden und muss erklärt werden können (z.B. Was ist eine Liniengleichung? Was sind baryzentrische Koordinaten? etc.).