UNIVERSITÄT BASEL

Prof. Dr. Thomas Vetter
Departement Mathematik und Informatik

Spiegelgasse 1
CH – 4051 Basel

Patrick Kahr (patrick.kahr@unibas.ch)
Clemens Büchner (clemens.buechner@unibas.ch)

Computer Grafik 2019 - Übungsblatt 1

Ausgabe in Woche 1 (21.02.2019).

Vorführung der laufenden Programme im Tutorium Woche 3 (Abgabe 08.03.2019) .

Maximal zu erreichende Punktzahl: 24

Ziele dieses Blattes sind die in der Vorlesung vorgestellten Algorithmen zum Zeichnen von Linien und Dreiecken zu implementieren und die Architektur einer Rendering Pipeline besser zu verstehen.

Den Renderer, den Sie in diesem Blatt komplettieren und auf dem die weiteren Blätter aufbauen werden, kann Dreiecke und Linien zeichnen und ist im Wesentlichen ein 2-Stufen Prozess:

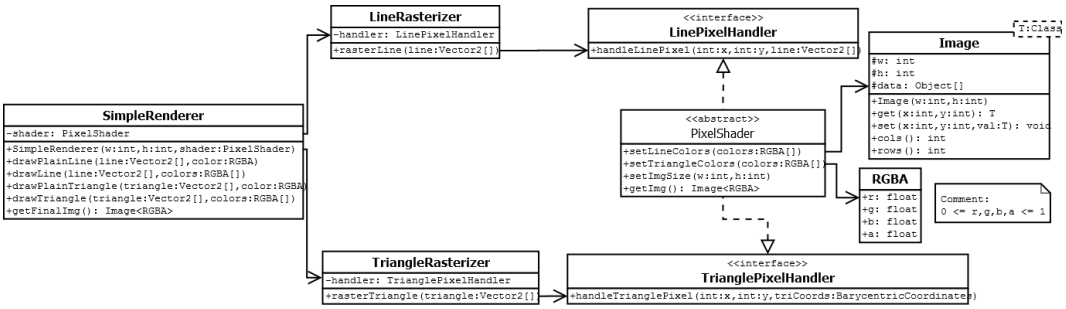
(a) Als erstes wird bestimmt, welche Pixel im zu erstellenden Bild von einem Objekt (Linie oder Dreieck) getroffen werden. Diese Aufgabe wird vom Rasterizer übernommen.

(b) Wenn der Rasterizer einen Pixel "gefunden" hat, färbt der Shader diesen ein. Je nach Art des Shaders können Objekte verschieden eingefärbt werden.

Die Renderer, welche in der Praxis verwendet werden und auch den, welchen Sie bis zum Ende des Semesters implementieren werden, sind weitaus komplexer. Nichtsdestotrotz sind diese beiden Schritte in irgendeiner Form immer Teil einer Rendering Pipeline.

Im soeben beschriebenen Fall wird ein Pixel direkt eingefärbt nachdem er vom Rasterizer identifizert wurde. Dies muss nicht unbedingt so sein. Es ist auch möglich, zuerst alle Objekte zu rastern und die Farbzuweisung erst später für alle Pixel auf einmal vorzunehmen. Solch eine Architektur wird auch als Deferred Shading bezeichnet. Deferred Shading hat den taktischen Vorteil, dass der Rasterizer unabhängig vom Shader ist.

UNI
BASEL



In diesem UML sehen sie eine Klassen-Übersicht der beschriebenen 2-Stufen Pipeline. Wie Sie sehen, rufen dabei die beiden Rasterizer nicht direkt einen Pixel Shader auf, sondern eine Implementationen des Interfaces Line- bzw. TrianglePixelHandler. Eine direkte Variante wäre durchaus denkbar, die Zwischenstufe der Pixel-Handler ist jedoch notwendig, um später das Deferred Shading einfach implementieren zu können.

Aufgabe 1 - Image<RGBA> (2 Punkte)

Ziel dieser Aufgabe ist es sich mit dem image package vertraut zu machen.

Erzeugen sie mit Hilfe des image packages eine neue Image<RGBA> Instanz und zeichnen Sie von “Hand” einige Pixel oder Linien speichern Sie das Bild anschliessend.

- Wie liegt das Koordinatensystem im Bild?
- Was geschieht, wenn man auf Pixel ausserhalb des Images zugreift?
- Was geschieht wenn man die Farbe eines Pixels abruft, wenn dieser vorher nicht eingefärbt wurde?

Hinweis: $0 \leq r, g, b, a \leq 1$

Benötigte Dateien: exercises.Ex1.java

Aufgabe 2 - Image Flip (2 Punkte)

Ergänzen sie die Methode flipImageUpsideDown, welche ein Image horizontal in der Mitte spiegelt. Wenden sie diese Funktion auf das in der Aufgabe 1 erstellte Bild an und speichern sie das gespiegelte Bild.

Benötigte Dateien: exercises.Ex1.java

Aufgabe 3 - Bresenham Linien zeichnen (4 Punkte ⋆)

Implementieren Sie die Funktion bresenham(..) der Klasse LineRasterizer, welche

eine Linie mit Hilfe des Bresenham Algorithmus' rastert. Eine Linie wird durch zwei Vektoren Vector2 repräsentiert (Start- und Endpunkt). Wenn Sie innerhalb des Bresenham-Algorithmus einen Pixel bestimmt haben, rufen sie anschliessend die Methode handleLinePixel() des linePixelHandlers auf, welcher im ConstantColorShader implementiert ist und alle Pixel gleich einfärbt.

Hinweis 1: Die Erklärung des Bresenham Algorithmus auf den Vorlesungsfolien gelten nur für den Fall, dass die Steigung der Linie im Intervall $[-1, 1]$ liegt! Sie müssen jedoch auch die anderen Fälle abdecken.

Hinweis 2: Vergessen Sie das Clipping nicht.

Benötigte Dateien: rasterization.LineRasterizer.java

Aufgabe 4 - Linienbilder generieren (2 Punkte)

Nachdem Sie den Bresenham Algorithmus implementiert haben, zeichnen Sie in der Methode lineRasterExample der Ex1 Klasse Linien mit unterschiedlichen Steigungen um sich zu vergewissern, dass Ihre Linien lückenlos gezeichnet werden. Verwenden Sie dazu die Funktion simpleRenderer.drawPlainLine(Vector2[] line, RGBA color), die im Hintergrund Ihren Algorithmus aus Aufgabe 3 aufruft.

Hinweis: Zeichnen Sie Linien mit unterschiedlich Steigungen, welche auch grösser als 1 sind.

Benötigte Dateien: exercises.Ex1.java

Aufgabe 5 - Baryzentrische Koordinaten (2 Punkte ⋆)

Die baryzentrischen (oder auch Dreiecks-) Koordinaten, sind sehr hilfreich wenn es um das Rastern und Einfärben von Dreiecken geht. Zur Erinnerung: Punkte $\vec{p}$ innerhalb eines Dreiecks $(\vec{a}, \vec{b}, \vec{c})$ lassen sich mit Hilfe der baryzentrischen Koordinaten darstellen als

$$\vec{p} = \lambda_0 \vec{a} + \lambda_1 \vec{b} + \lambda_2 \vec{c}$$

mit

$$0 \leq \lambda_i \leq 1 \mid i \in \{0, 1, 2\} \qquad \text{und} \qquad \sum_{i \in \{0,1,2\}} \lambda_i = 1.$$

Verfollständigen Sie die Methode getBarycentricCoordinates() der Klasse BarycentricCoordinateTransform, welche gegeben der drei Eckpunkten $(\vec{a}, \vec{b}, \vec{c})$ eines Dreiecks für einen Punkt $\vec{p} = (x, y)$ die Baryzentrischen Koordinaten λ_i berechnet.

Benötigte Dateien: utils.BarycentricCoordinateTransform.java

Aufgabe 6 - Gefüllte Dreiecke zeichnen (6 Punkte ⋆)

Implementieren Sie die Funktion rasterTriangle(..) der Klasse TriangleRasterizer, die ein ausgefülltes Dreieck zeichnet. Ein Dreieck wird ebenfalls als Array von Vector2 repräsentiert (diesmal aber natürlich mit drei Einträgen). Es gibt verschiedene Verfahren um Dreiecke zu rastern. Verwenden Sie das ”billige” Verfahren, welches in der Vorlesung vorgestellt wurde. Wenn Sie einen Pixel gefunden haben, der angezeichnet werden muss, rufen Sie analog zur Aufgabe 1 die Methode handleTrianglePixel des trianglePixelHandlers auf. Diese Methode erwartet die baryzentrischen Koordinaten (lambda) des Punktes (x, y) als Übergabeparameter.

Um zu testen ob Ihre Transformation in baryzentrischen Koordinaten und die Rasterung der Dreiecke richtig funktioniert, rufen Sie die statische Methode generatePlasererdImage() in ex1.test.PlasteredImage.java auf. Diese erzeugt eine gleichmässige Fläche aus Dreiecken indem die Farbwerte der einzelnen Pixel addiert werden. Falls Ihre Implementation richtig funktioniert, ist das Bild *plastered.png* eine einheitlich graue Fläche.

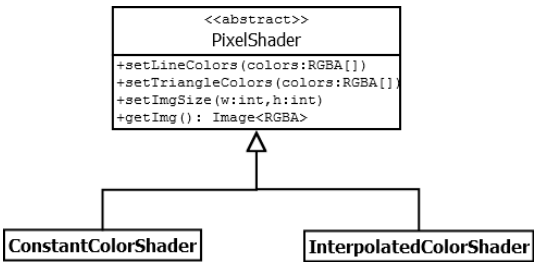
Hinweis 1: Um die barycentrischen Koordinaten des Punktes (x, y) zu erhalten, verwenden Sie die in der Aufgabe 5 implementierte Methode getBarycentricCoordinates()

Hinweis 2: Machen Sie sich in der Aufgabe auch Gedanken zum Clipping. Es kann sein, dass spätere Tests crashen, wenn Sie kein Clipping implementiert haben (ausserdem kann Clipping die Effizienz ihrer Pipeline steigern).

Hinweis 3: Die Methode generatePlasererdImage in der Ex1 füllt das ganze Bild mit Dreiecken, welche sich nicht überlappen. Falls Ihr Bild eine einheitlich graue Fläche ist, werden die Ränder ihrer Dreiecke richtig gerundet.

Benötigte Dateien: rasterization.TriangleRasterizer.java
exercises.Ex1.java

Aufgabe 7 - Interpoliertes Shading (6 Punkte ⋆)



Nun da Sie Linien und Dreiecke rastern können, widmen wir uns in dieser Aufgabe einem etwas fortgeschrittenen Shader. Folgende Methoden der abstrakten Klasse PixelShader müssen implementiert werden:

- handleTrianglePixel(..). Diese Methode soll den Farbwert c für den Pixel an der Position x, y im Bild aus den Eckfarben c_0^t, c_1^t, c_2^t interpolieren. Verwenden Sie dazu die baryzentrischen Koordinaten, die der Funktion übergeben werden. Die benötigten Farbwerte sind in der Membervariable lineColors des PixelShaders gespeichert.
- Analog für die Methode handleLinePixel(..), welche den Farbwert des Pixels an der Position x, y im Bild auf den aus den beiden Farben der Endpunkten, c_0^l und c_1^l, interpolierten Wert setzt:

$$c(x, y) = \lambda_0 c_0^l + \lambda_1 c_1^l$$

Zeichnen und speichern Sie ein Bild welches verschiedenfarbige Linien und Dreiecke enthält um sich zu vergewissern, dass Ihre Implementation korrekt ist.

Hinweis 1: Die Farben der Eckpunkte sind in triangleColors bzw. lineColors der Klasse PixelShader.java gespeichert.

Hinweis 2: RGBA-Farben können sie mittels RGBA.times(factor) mit einem Faktor multiplizieren.

Hinweis 3: Um die Farben in einem Dreieck zu interpolieren, schauen Sie sich die Klasse BarycentricCoordinates.java etwas genauer an.

Benötigte Dateien: shader.InterpolatedColorShader.java