Universität Basel

Computergrafik 2021

Prof. Dr. Thomas Vetter
Departement Mathematik und Informatik
Spiegelgasse 1, CH – 4051 Basel

Patrick Kahr (patrick.kahr@unibas.ch)
Renato Farruggio (renato.farruggio@unibas.ch)
Martin Ramm (martin.ramm@unibas.ch)

Übungsblatt 1

Ausgabe: 04.03.2021
Abgabe: 18.03.2021

Zu erreichende Punktzahl: 20 (Programmieraufgaben)

Bevor Sie die Übungen lösen, lesen Sie bitte dieses Übungsblatt sowie das Infoblatt aufmerksam durch. Es ist wichtig dass die Abgabe rechtzeitig und wie im Infoblatt beschrieben erfolgt.

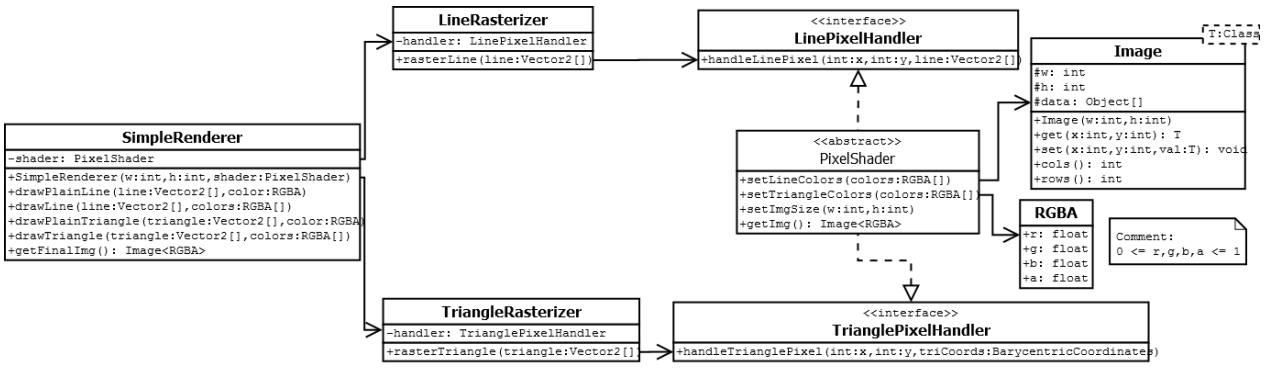
Beachten Sie, dass zukünftige Aufgaben auf solche die mit einem ⋆ gekennzeichnet sind aufbauen können. Es empfiehlt sich also die ⋆ Aufgaben besonders sorgfältig zu lösen. Auch wenn nicht alle Aufgaben dieses Blattes gelöst werden müssen kann die Befragung zum Vorlesungsstoff (Interview) alle Aufgaben abdecken.

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

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

2. 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.



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

(a) Bildoperationen — (1 Punkt)

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. Machen Sie sich auch Gedanken zum Koordinatensystem und zum Definitionsbereich des Bildes. Werden alle Fälle richtig behandelt?

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

Benötigte Dateien: `exercises.Ex1.java`

(b) Bild spiegeln — (1 Punkt)

Nun ergänzen Sie die Methode `flipImageUpsideDown(..)`, welche ein `Image` horizontal in der Mitte spiegelt. Wenden Sie diese Funktion auf das zuvor erstellte Bild an und speichern Sie das gespiegelte Bild.

Benötigte Dateien: `exercises.Ex1.java`

Unit-Tests: `ex1.ImageFlipTest.java`

Aufgabe 2 – Bresenham Algorithmus — 8 Punkte

(a) Algorithmus implementieren — (6 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. Machen Sie von den zur Verfügung gestellten Unit Tests Gebauch, um Ihre Implementierung zu überprüfen.

Hinweis 2: Vergessen Sie das Clipping nicht.

Benötigte Dateien: `rasterization.LineRasterizer.java`

Unit-Tests: `ex1.BresenhamTest.java`

(b) Linien zeichnen — (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 Teilaufgabe (a) aufruft.

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

Benötigte Dateien: `exercises.Ex1.java`

Aufgabe 3 – Baryzentrische Koordinaten

3½ Punkte

Die baryzentrischen (oder auch Dreiecks-) Koordinaten sind sehr hilfreich, wenn es um das Rastern und Einfärben von Dreiecken geht.

Vervollständigen Sie die Methode `getBarycentricCoordinates()` der Klasse `BarycentricCoordinateTransf` 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`

Unit-Tests: `ex1.TriangleTest.java`

Aufgabe 4 – Gefüllte Dreiecke zeichnen

3 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 Verfahren, welches in der Vorlesung vorgestellt wurde. Wenn Sie einen Pixel gefunden haben, der angezeichnet werden muss, rufen Sie analog zur Aufgabe 2 die Methode `handleTrianglePixel(..)` des `trianglePixelHandler`s auf. Diese Methode erwartet die baryzentrischen Koordinaten (`lambda`) des Punktes (x, y) als Übergabeparameter.

Hinweis 1: Um die baryzentrischen Koordinaten des Punktes (x, y) zu erhalten, verwenden Sie die in der Aufgabe 3 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).

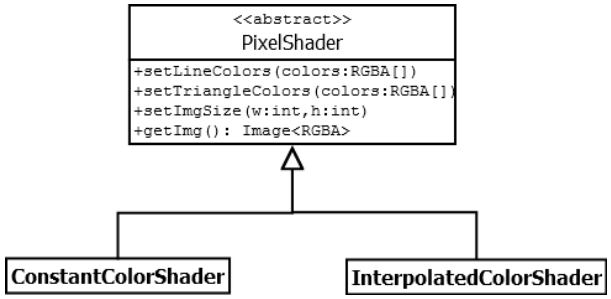
Hinweis 3: Der Unit Test `testPlasteredImage()` in `ex1.TriangleTest` füllt ein ganzes Bild mit Dreiecken, welche sich nicht überlappen. Falls das resultierende Bild eine einheitlich graue Fläche ist, werden die Ränder Ihrer Dreiecke richtig gerundet.

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

Unit-Tests: `ex1.TriangleTest.java`

Aufgabe 5 – Interpolated Shading ⋆

3½ 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.

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` 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` etwas genauer an.

Benötigte Dateien: `shader.InterpolatedColorShader.java`

Unit-Tests: `ex1.InterpolatedColorShader.java`