Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 4

Sommersemester 2018

4 Der Rasterisierungsalgorithmus

Aufgabe 4.1

Nachdem die ErzeugeDreiecke Phase beim Rasterisierungsalgorithmus durchlaufen ist, liegen die zu zeichnenden Dreiecke in *Fensterkoordinaten* vor. Trifft man die vereinfachende Annahme, dass geometrische, aus Dreiecken zusammengesetzte Objekte geschlossen sind und immer nur von außen betrachtet werden, kann man vor der Scan-Konvertierungsphase bereits sehr viele Dreiecke vom Zeichnen ausschließen. Man legt eine Konvention bzgl. der Orientierung gemäß Uhrzeigersinn der Dreiecke (engl. "winding order") fest. Dazu betrachten wir die Dreieckseckpunkte v_1, v_2 und v_3. v_1 und v_2 bilden eine gerichtete Kante und teilen die Ebene in zwei Hälften. Liegt nun v_3 in der linken Hälfte der Ebene, ist die *winding order* des Dreiecks gegen den Uhrzeigersinn. Wir legen (beliebig) fest, dass wir solche Dreiecke von der Vorderseite sehen. Liegt v_3 in der rechten Hälfte, sehen wir das Dreieck von der Rückseite und müssen es aufgrund unserer vereinfachenden Annahme nicht weiter betrachten. Das Vorgehen bezeichnet man als "backface culling".

Die Orientierung des Dreiecks kann man leicht ermitteln, indem man das Vorzeichen der Determinante der Eckpunktsmatrix in homogenen Koordinaten berechnet:

$$\det T = \begin{vmatrix} v_{1x} & v_{2x} & v_{3x} \\ v_{1y} & v_{2y} & v_{3y} \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} v_{2x} - v_{1x} & v_{3x} - v_{1x} \\ v_{2y} - v_{1y} & v_{3y} - v_{1y} \end{vmatrix}$$

Für nicht degenierte Dreiecke (bei letzteren sind die drei Eckpunkte kolinear) ist bzgl. unserer Konvention für "front faces" das Vorzeichen positiv und für "back faces" das Vorzeichen negativ.

a.)
Rechnen Sie für die Dreiecke

$$\begin{aligned} T_1 &= \{v_1 = (2,2), v_2 = (4,2), v_3 = (4,4)\}, \\ T_2 &= \{v_1 = (5,5), v_2 = (2,10), v_3 = (10,5)\} \end{aligned}$$

aus, ob sie back faces oder front faces sind. (2 Punkte)

b.)
Der Scan Konvertierungsalgorithmus von Pineda, den wir in der Vorlesung kennengelernt haben, lässt sich auf den gleichen geometrischen Zusammenhang zurückführen. Zeigen Sie, wie die Determinante zweier 2D Punkte v_i, v_{i+1} sowie eines Rasterpunkts $p = (x, y)$ in der Ebene mit den Kantenfunktionen $E_i(x, y)$ aus dem Algorithmus von Pineda zusammenhängt. Nehmen Sie (anders als in der Vorlesung) an, dass die zur Kantenfunktion gehörige Kante gemäß $e_i = v_i - v_{i+1}$ gebildet wird. (3 Punkte)

c.)
Gegeben sei das Dreieck $T = \{v_1, v_2, v_3\}$ mit

$$\begin{aligned} e_1 &= v_1 - v_2 \\ e_2 &= v_2 - v_3 \\ e_3 &= v_3 - v_1 \end{aligned}$$

und mit den drei Kantenfunktionen

$$\begin{aligned} E_1(x, y) &= (x - v_{1x})e_{1y} - (y - v_{1y})e_{1x} \\ E_2(x, y) &= (x - v_{2x})e_{2y} - (y - v_{2y})e_{2x} \\ E_3(x, y) &= (x - v_{3x})e_{3y} - (y - v_{3y})e_{3x}. \end{aligned}$$

Zeigen Sie, dass für einen Punkt (x, y) in der Ebene

$$2A(T) = E_1(x, y) + E_2(x, y) + E_3(x, y)$$

gilt, wobei $A(T)$ die Fläche von T bezeichne. (5 Punkt)

d.)
Baryzentrische Koordinaten hinsichtlich Dreiecken sind Tripel $b(p) = b(x, y) = (\lambda_1, \lambda_2, \lambda_3)$, wobei $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{R}$ Gewichte für die Dreieckseckpunkte sind. Für jeden Punkt (x, y) in der Ebene und das Dreieck $T = \{v_1, v_2, v_3\}$ gibt es ein solches Tripel, sodass gilt:

$$\begin{aligned} \lambda_1 v_1 + \lambda_2 v_2 + \lambda_3 v_3 &= (x, y), \\ \lambda_1 + \lambda_2 + \lambda_3 &= 1 \end{aligned}$$

Bzgl. der Dreieckseckpunkte gilt: $b(v_1) = (1, 0, 0)$, $b(v_2) = (0, 1, 0)$ und $b(v_3) = (0, 0, 1)$.

Zeigen Sie, dass für die Kantenfunktionen aus Aufgabenteil **c.)** und für $b(x, y)$

$$\begin{aligned} \lambda_1 &= \frac{E_2(x, y)}{2A(T)} \\ \lambda_2 &= \frac{E_3(x, y)}{2A(T)} \\ \lambda_3 &= \frac{E_1(x, y)}{2A(T)} \end{aligned}$$

gilt, wobei $A(T)$ die Fläche von T bezeichne. (10 Punkte)

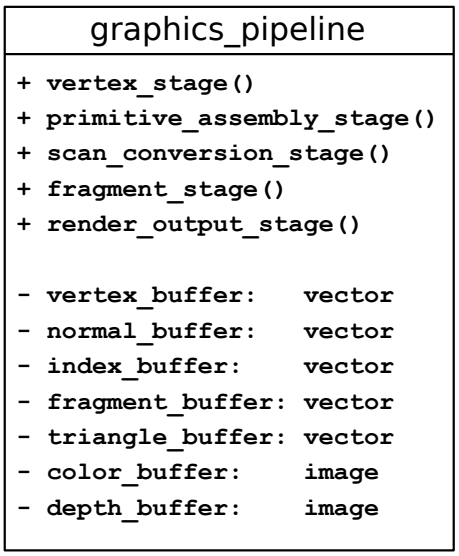
Aufgabe 4.2

Der Algorithmus PRIMAERSTRAHLVERFOLGUNG sollte, abgesehen von Rundungsunterschieden, optisch identische Bilder wie der Algorithmus RASTERISIERUNG generieren. Während beim Algorithmus RASTERISIERUNG Beleuchtung bereits für Fragmente durchgeführt werden muss, wird beim Algorithmus PRIMAERSTRAHLVERFOLGUNG die Beleuchtung *verzögert* durchgeführt. Darüber hinaus ist RASTERISIERUNG ein CRCW Algorithmus. Nennen Sie mögliche Gründe dafür, warum auf GPUs heute sowie früher dennoch der Algorithmus RASTERISIERUNG, und nicht der Algorithmus PRIMAERSTRAHLVERFOLGUNG, implementiert ist. (5 Punkte)

Aufgabe 4.3

Ziel bei dieser Aufgabe ist es, den Rasterisierungsalgorithmus besser zu verstehen, indem wir ihn als CPU Programm implementieren. Als Vorlage dient Ihnen dazu die Datei `rasterization.cpp` aus dem Gerüstprogramm. Das Gerüstprogramm ist deutlich komplexer, als es die Gerüstprogramme aus den vorhergehenden Übungsblättern waren. Teil der Aufgabe ist es auch, sich mit der Programmstruktur vertraut zu machen.

Das Gerüstprogramm implementiert die *logische* Rasterisierungs-Pipeline, die wir in der Vorlesung kennengelernt haben. Kern der Pipeline ist die Klasse `graphics_pipeline`. Das folgende Diagramm zeigt eine vereinfachte Version der Klasse. Das öffentliche Interface stellt Funktionen bereit, die die einzelnen Phasen der Rasterisierungs-Pipeline abbilden.



Der Benutzer der Klasse füllt erst die Input Buffer: *Vertices*, *Indices*, *Normalen*. Daraufhin kann er nacheinander die einzelnen Stages der Rasterisierungs-Pipeline aufrufen. Dreiecks Buffer und Fragment Buffer füllt der Benutzer nicht selbst. Vielmehr werden diese während der einzelnen Stages der Pipeline befüllt. Nach der Primitive Assembly Phase befinden sich Dreiecke in Fensterkoordinaten im Buffer für Dreiecke. Nach der Scan Konvertierungsphase befinden sich Fragmente im Buffer für Fragmente. Die Klasse `graphics_pipeline` speichert außerdem noch den Zustand, z. B. den Alpha-Blending, Tiefentest, oder Backface-Culling Modus. Ist die Pipeline vollständig durchlaufen, sind die Output Buffer mit Bilddaten (Farbe und Tiefe) befüllt. Mit dedizierten Funktionen `read_color_buffer()` kann das Ergebnis aus der Pipeline gelesen werden.

Die Vertex Stage und Fragment Stage der Pipeline haben eine Besonderheit: sie sind programmierbar. Dazu übergibt der Benutzer den Funktionen `vertex_stage()` und `fragment_stage()`

Vertex- und Fragment Shader, die als Input je ein Vertex bzw. ein Fragment erhalten. Vertices und Fragmente dürfen verändert werden. Dies ist mit C++11 Lambda Funktionen implementiert. Diese können auch auf global Verfügbare Parameter ("Uniforme Variablen") zugreifen, dies ist mit Lambda Capturing implementiert. Vertices sind definiert über die Attribute Position und Normale. Diese könenn im Vertex Shader verändert (z. B. transformiert) werden. Im Fragment Shader stehen sie dann als Attribute in interpolierter Form zur Verfügung.

vertex	
+ pos:	vec3
+ normal:	vec3

fragment	
+ x:	int
+ y:	int
+ primitive_id:	int
+ z:	float
+ n:	vec3
+ color:	vec4
+ depth:	float

Im Fragment Shader stehen Ihnen die *Fragmenteigenschaften* `x`, `y` und `primitive_id` zur Verfügung, die Ihnen die Rasterposition des Fragments in Fensterkoordinaten angeben, sowie die ganzzahlige ID des Primitivs, zu dem das Fragment gehört. `z` und `n` sind die über das Dreieck in der Scan Konvertierungsphase interpolierten Tiefen- und Normalen*attribute*, aus denen Sie im Fragment Shader lesen dürfen. *Output Attribute* im Fragment Shader sind `color` und `depth`. `depth` ist, wenn Sie das Attribut nicht verändern, mit dem Wert des Input Attributs `z` initialisiert. Die Fragmentfarbe `color` ist als RGBA Tupel *schwarz und durchsichtig* initialisiert.

a.) Zum Transformieren der Vertices in "Normalisierte Gerätekoordinaten" benötigen wir unter anderem die Matrix zur perspektivischen Transformation aus der Vorlesung. Schreiben Sie dazu die Funktion `perspective()` so um, dass sie aus den übergebenen Werten die Matrix `proj_matrix` mit der "Perspective Method" berechnet und zurück gibt. (2 Punkte)

b.) Die Rasterisierungs-Pipeline wird in der Funktion `main()` ausgeführt. Implementieren Sie dort einen einfachen Vertex Shader, der der Funktion `graphics_pipeline.vertex_stage()` übergeben wird. Der Vertex Shader transformiert alle Vertices in "Normalisierte Gerätekoordinaten". Dazu stehen Ihnen die Matrizen `view_matrix` und `proj_matrix` als uniforme Variablen zur Verfügung. Beachten Sie dabei die Informationen im Appendix zur Benutzung der linearen Algebra Funktionen.
Außerdem transformieren Sie die Normalenvektoren, die mit jedem Vertex gespeichert werden. Normalenvektoren werden nicht in das NDC Koordinatensystem transformiert, sondern nur in Augkoordinaten. Einheitsvektoren transformiert man außerdem auch anders als Punkte: man setzt ihre homogene Koordinate ("w") auf 0 und transformiert sie mit dem invers-transponierten der Modelview Matrix. Auch diese Matrix steht Ihnen im Vertex Shader zur Verfügung:
`view_matrix_it`. (5 Punkte)

c.) Leider ist die Rasterisierungs-Pipeline nicht vollständig implementiert. Die Scan Konvertierungsphase, die eigentlich Dreiecke scan-konvertieren sollte, hat der Entwickler vergessen zu implementieren. Implementieren Sie den Algorithmus von Pineda aus der Vorlesung in der privaten Member Funktion `rasterize()`, die von der Funktion `graphics_pipeline.scan_conversion_stage()` aufgerufen wird. `rasterize()` erhält als Input ein Dreieck *in Fensterkoordinaten*:

triangle
+ v1: vec2
+ v2: vec2
+ v3: vec2
+ z1: float
+ z2: float
+ z3: float
+ n1: vec3
+ n2: vec3
+ n3: vec3

Ihnen stehen die Fensterkoordinaten `v1`, `v2` und `v3` mit Subpixelgenauigkeit zur Verfügung. Außerdem speichert das Dreieck Attribute für Tiefe: `z1`, `z2` und `z3`, sowie Normalen für jeden Eckpunkt: `n1`, `n2` und `n3`. Diese muss Ihr Algorithmus *perspektivisch korrekt* über die Dreiecksoberfläche interpolieren.

Gehen Sie wie folgt vor. Mit der bereits implementierten Funktion `get_bounds()` ermitteln Sie das das Dreieck umschließende Rechteck. Dann iterieren Sie über jede Rasterposition im Rechteck und evaluieren die Pineda Kantenfunktionen für jede der Kanten (`v1 - v2`), (`v2 - v3`) und (`v3 - v1`). Ist `backface_culling` aktiviert, dann interessieren Sie sich für alle Rasterpositionen, für die alle Kantenfunktionen ≥ 0 sind. Ist `backface_culling` deaktiviert, interessieren Sie sich außerdem für alle Rasterpositionen, für die alle Kantenfunktionen < 0 sind. Erzeugen Sie Fragmente für diese, die Sie mit der Rasterposition `(x,y)` und der `id` initialisieren. Fügen Sie die Fragmente dem `fragment_buffer` an, den `graphics_pipeline` als privaten Member speichert. Außerdem müssen Sie das Fragment mit Werten für `z` sowie die Normale `n` initialisieren. Ermitteln Sie diese, indem Sie die Eingabeattribute aus dem Dreieck über die Dreiecksoberfläche interpolieren. Rechnen Sie dazu baryzentrische Koordinaten aus, die sich aus den Kantenfunktionen ableiten lassen (vgl. Aufgabe 4.1 **d.)**). Die Funktion `area(triangle)` ist bereits implementiert und darf von Ihnen benutzt werden. Auch die Funktion zur Interpolation mit baryzentrischen Koordinaten müssen Sie nicht selber implementieren, die lineare Algebra Bibliothek stellt dies bereits zur Verfügung. Verwenden Sie dazu die Funktion `lerp(T attr1, T attr2, T attr3, float bary1, float bary2)`, die Ihnen das interpolierte Attribut vom Typ `T` (z. B. `float` oder `vec3`) zurückgibt. (10 Punkte).

d.) Die Ausgabe, die Sie in der Output Datei sehen, ist noch nicht sehr sinnvoll, denn der Entwickler hat auch die Render Output Phase, bei der u. a. der Tiefentest vorgenommen wird, noch nicht vollständig implementiert. In der Funktion `graphics_pipeline.render_output_stage()` implementieren Sie dazu den Tiefentest. In der Funktion werden bereits die Output Buffer initialisiert. Ihre Aufgabe ist es, über den initialisierten Fragment Buffer zu iterieren. Testen Sie für jedes Fragment `f` aus dem Fragment Buffer den Tiefenwert `depth_buffer(f.x, f.y)`. Ist die mit dem Fragment gespeicherte Tiefe `f.depth` kleiner als der Tiefenwert im Tiefenpuffer, aktualisieren Sie den Tiefenpuffer und den Farbpuffer. Um höhere Genauigkeit bei den nachfolgenden Berechnungen zu erzielen, werden die Farboperationen auf einem temporären Buffer mit 32-bit Floating Point Genauigkeit durchgeführt. Der Tiefentest soll wahlweise auch deaktiviert werden

können. Ist die Variable `depth_test false`, sollte Ihr Fragment immer in den Output geschrieben werden, gleich ob es den Tiefentest "gewinnt" oder nicht. (8 Punkte).

e.) Der bereits implementierte Fragment Shader weist dem Fragment einfach die Farbe weiß zu. Implementieren Sie einen etwas realistischeren Fragment Shader, der *Lambert Shading* berechnet. Ihnen stehen die diffuse RGBA Materialfarbe (`diff`) als uniforme Variable zur Verfügung, sowie ein normalisierter Vektor zu einer Lichtquelle (`L`). Beim Lambert Shading Modell gewichten Sie den diffusen RGB Anteil der Materialfarbe mit dem Skalarprodukt aus Vertexnormale und Vektor zum Licht und speichern die so erhaltene Farbe im Fragment Output. Implementieren Sie ein zweiseitiges Shading Modell - Fragmente, die Sie von der Rückseite anschauen, werden auch beleuchtet. (Für das Skalarprodukt können Sie die lineare Algebra Funktion `dot(vec3,vec3)` verwenden.) (5 Punkte).

f.) Zum Schluss passen Sie die Pipeline so an, dass sie Alpha Blending unterstützt. Dazu müssen Sie die Render Output Phase erweitern. Das Alpha Blending ist der Grund, warum Sie das Compositing in einem temporären Buffer vornehmen. Implementieren Sie Alpha Blending mit dem Over Operator aus der Vorlesung, und mit vormultipliziertem Alpha. Denken Sie daran, Alpha Blending nur durchzuführen, wenn der Alpha Blending Modus aktiviert ist. In Kombination mit Alpha Blending empfiehlt es sich, Tiefentest und Backface Culling auszuschalten.
Um Alpha Blending korrekt zu implementieren, muss die Geometrie vorher tiefensortiert werden. Genau nachdem die Vertices eingelesen wurden, sollten Sie sie tiefensortieren. Gehen Sie wie folgt vor:
Erzeugen Sie eine Kopie der Vertices. Transformieren Sie alle kopierten Vertices in Augkoordinaten. Im Augkoordinatensystem ist die Kamera im Ursprung und entlang der negativen z-Achse orientiert, man kann die Vertices hier also gut tiefensortieren. Wir sortieren aber nicht Vertices, sondern Dreiecke! Gehen Sie davon aus, dass sich die Dreiecke niemals schneiden - dann sortieren Sie die Dreiecke, indem Sie das arithmetische Mittel der Eckpunkte bestimmen. Sie sortieren nicht die Eingabedaten, sondern die kopierten Daten, merken sich aber (mit einer Methode Ihrer Wahl), welche Originalvertices mit welchen transformierten Vertices korrespondieren. Nachdem Sie die sortierten, in Augkoordinaten transformierten Vertices sortiert haben, stellen Sie die neu entstandene Ordnung auch auf den Originaldaten her. (Die Transformation der Originaldaten in Aug- und später in NDC-Koordinaten ist Aufgabe des Vertex Shaders). (10 Punkte).

Appendix
Bei der Implementierung verwenden Sie die Visionaray Bibliothek für Lineare Algebra Funktionen:

```
https://github.com/szellmann/visionaray
```

Teile der Bibliothek sind "Header-Only", Sie müssen nicht dagegen linken. So ist es auch mit dem Linearen Algebra Bestandteil der Bibliothek. Inkludieren Sie `<visionaray/math/math.h>`, und Ihnen stehen alle Funktionen zur Verfügung. Dazu müssen Sie dem Compiler den *Include Pfad* der Bibliothek mitteilen, z. B. mit GCC:

```
g++ rasterization.cpp -I/path/to/visionaray/include -std=c++11 -O3
```

Die Funktionen orientieren sich weitestgehend an der Spezifikation von GLSL. Beispiele finden Sie in der beigefügten .cpp Datei.

Abgabe bitte bis zum 06.06.2018, 22:00h in Ilias.