Prof. Dr. Reinhard Klein
Dr. Michael Weinmann
Richard Vock, Sebastian Ochmann

Universität Bonn
Institut für Informatik II
25.06.2018

Sommersemester 2018

Einführung in die Computergraphik
Übung 9

Deferred Rendering

In dieser Übung werden wir uns moderneren Rendering-Konzepten nähern und die Basis für screen-space Rendering-Techniken schaffen.

Nachteile des Forward Renderings

Bisher hatten wir ein relativ fixes Konzept einer Rendering Pipeline. Der bisherige Ansatz - üblicherweise als *Forward Rendering* bezeichnet - basiert auf der Idee eines einzelnen Geometry-zu-Bild-Output Schrittes. Die Nachteile dieses Ansatzes werden am ehesten deutlich wenn wir Szenenbeleuchtung mit vielen Lichtquellen machen wollen. Angenommen wir haben eine Szene mit 200 Lichtquellen und recht komplexer Geometrie. Jeder Vertex (und seine interpolierte Form) muss auf Sichtbarkeit getestet und seine entsprechende Beleuchtung berechnet werden - *unabhängig davon ob er im finalen Output überhaupt zu einem Pixel beiträgt.*

Um jene komplexe Beleuchtungsberechnung von der tatsächlichen Geometrie zu entkoppeln, führt *Deferred Rendering* das Konzept von screen-space Berechnungen ein.

Grundlagen des Deferred Rendering

Gehen wir von einem Setup aus in dem für jeden *Pixel im Output-Bild* die Beleuchtung berechnet werden soll. Offensichtlich brauchen wir dafür die Normalen, vielleicht Positionen oder sogar Materialeigenschaften der Oberflächen welche zum Pixel beitragen. Da in unserem bisherigen Grundkonzept von OpenGL (ohne Transparenz) aber davon ausgegangen wird, dass nur die vorderste Geometrie zu einem Pixel beiträgt, brauchen wir diese Geometrieinformation eigentlich auch nur für jene Oberflächen.

Die grundlegende Idee des Deferred Rendering ist es hierfür ein 2-schrittiges Renderingsetup einzuführen:

1. Im ersten Pass (welcher in eine Texture statt dem Outputbuffer gerendert wird), zeichnen wir die eigentliche Geometrie, wobei die geschriebenen Pixel *die Oberflächeneigenschaften* kodieren welche wir z.B. für die Beleuchtung brauchen. Die entsprechende Ausgabetextur wird i.d.R. als **G-Buffer** (geometry buffer) bezeichnet.
2. Im zweiten Pass rendern wir ein bildschirmfüllendes Viereck. Für jeden Pixel in diesem Viereck lesen wir nur die Geometrieinformation aus dem G-Buffer und berechnen auf Basis dieser (und weiterer) Informationen z.B. die Beleuchtung und damit unseren Output.

Dieses Setup hat den riesigen Vorteil, dass alle Operationen im zweiten Pass *nur einmal pro Pixel* berechnet werden müssen und daher unabhängig von der Komplexität der Szene sind. Ernstzunehmende Nachteile dagegen sind Probleme mit Transparenzeffekten und Antialiasing (beides Themen die wir bisher noch nicht diskutierten). Es sei dennoch zu erwähnen, dass die Mehrheit moderner Renderingsysteme (insbesondere in Spielen) Deferred Rendering Setups sind.

Der G-Buffer

Wie oben angedeutet ist der Kern eines Deferred Renderers die Ausgabetextur des ersten Passes (G-Buffer). Dabei handelt es sich um eine Farbtextur in der Informationen über die Geometrie in Farbkanälen kodiert sind. Aus Performancegründen ist der G-Buffer üblicherweise eine einzelne Textur (abgesehen davon haben wir in WebGL keine Wahl, da der Standard die Ausgabe auf eine Textur beschränkt). Die Kunst ist hier jetzt so viele Informationen wie nötig bei minimalem Genauigkeitsverlust im verfügbaren Platz unterzubringen (heutzutage üblicherweise 4 Kanäle à 32bit).

In erweiterten Systemen enthält der G-Buffer in der Regel weitaus mehr Informationen, was Bitmanipulations-Tricks notwendig macht. So sieht der G-Buffer in *Crysis 3* z.B. wie im Bild unten beschrieben aus:

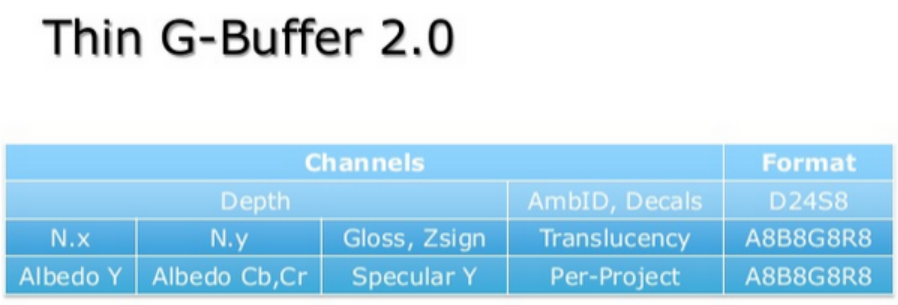

Figure 1: G-Buffer

(die Beschreibung ist von einem Foliensatz welcher für Interessierte sehr zu empfehlen ist).

In unserem Fall betrachten wir einen "einfachen" Fall wo wir Normalen und Positionen zum Zwecke der Beleuchtungsberechnung (Punktlichtquelle mit Phong Modell) brauchen. Dies stellt uns aber bereits vor ein offensichtliches Problem: Positionen (3 floats), und Normalendaten (3 floats, genau genommen 2) stellen bereits 6 floats (strenggenommen 5) dar und wir haben mit unseren 4 Kanälen dafür keinen Platz. Hier bedienen wir uns allerdings eines recht üblichen Tricks: Haben wir die Projektionsmatrix, so brauchen wir eigentlich pro Pixel nur die Tiefe (den z-Wert) um im zweiten Pass die Position berechnen zu können.

Aus diesem Grund schreiben wir folgendermaßen in unseren recht einfachen G-Buffer:

```
gl_FragColor = vec4(normal, gl_FragCoord.z);
```

Hier ist gl_FragCoord.z die Tiefe des aktuellen Fragments normalisiert auf das Intervall $[0, 1]$.

Position aus Fragment-Z

Bleibt die Aufgabe die Position (im eye-space) unserer lokalen Geometrie basierend auf dem gespeicherten z-Wert zu berechnen. Es sei daran erinnert, dass die (Sub-)Transformationskette in OpenGL

—Modelview—> **Eye-Space** —Projektion—> **NDC-Space** —Linear—> **Window-Space**

ist. Wir müssen also irgendwie diesen Prozess umkehren um unsere X, Y und Z-Werte im Eye-Space auszurechnen.

Der erste Schritt ist hier die Transformation des z-Wertes vom Window-Space in den NDC-Space. Dabei handelt es sich um eine einfache Transformation vom $[0, 1]$ in den $[-1, 1]$ Bereich, also

$$z_{ndc} = 2z_{wnd} - 1.$$

Der interessantere Teil ist die Transformation vom NDC-Space in den Eye-Space. Auch dies machen wir zunächst nur für den z-Wert. Es sei daran erinnert, dass die Transformation vom Eye-Space in den NDC-Space durch Multiplikation mit der Projektionsmatrix geschieht, also $z_{ndc} = Pz_{eye}$, wobei P üblicherweise definiert ist als

$$P = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{z_{far}+z_{near}}{z_{near}-z_{far}} & \frac{2z_{far}z_{near}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

In diesem Fall wollen wir aber nur den z-Wert und schauen uns dabei eine allgemeinere Form von Projektionsmatrix an, nämlich

$$P = \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & a & b \\ 0 & 0 & c & 0 \end{pmatrix}.$$

Nach Multiplikation mit P und *perspektivischer Division* durch die w-Koordinate erhalten wir:

$$\frac{az_{eye} + b}{cz_{eye}} = z_{ndc} \tag{1}$$

$$az_{eye} + b = z_{ndc} cz_{eye} \tag{2}$$

$$b = (cz_{ndc} - a)z_{eye} \tag{3}$$

$$z_{eye} = \frac{b}{cz_{ndc} - a}. \tag{4}$$

Bei der Implementation ist darauf zu achten, dass Matrizen in OpenGL als column-major Arrays gespeichert werden, also ist z.B. c im Code sceneProjection[2][3].

Damit erhalten wir den z-Wert im Eye-Space (also nach der ModelView Transformation, wo das Frustum bzw. die Blickrichtung entlang der negativen z-Achse liegt).

Um die letztendliche Position im Eye-Space auszurechnen nutzen wir nun den (ansonsten recht langweiligen) Vertex-Shader. Die Idee ist einen Strahl vom Betrachterpunkt (Ursprung) durch jeden Pixel im Vertex-Shader zu berechnen. Dafür brauchen wir eigentlich nur die Ausmaße der Near-Plane. Sei w_n die halbe Breite der Near-Plane und entsprechend h_n die halbe Höhe, so können wir die Richtung durch jeden Pixel berechnen mittels:

$$d_{eye} = \begin{pmatrix} 2w_n u - w_n \\ 2h_n v - h_n \\ -1 \end{pmatrix},$$

wobei $u, v \in [0, 1]^2$ die Koordinaten auf der Near-Plane normalisiert auf $[0, 1]$ sind. Die z-Koordinate von -1 stellt sicher, dass Interpolationen dieser Richtung ausschließlich innerhalb der Near-Plane stattfinden (was wir gleich benötigen).

Der Trick ist nun folgender: Im Bild unten wäre E unser Betrachterpunkt (Ursprung), P wäre der gewünschte Punkt im Eye-Space und Pz wäre unsere berechnete z-Koordinate im Eye-Space. Unsere Richtung d_{eye} oben zeigt nun in Richtung P und ihre Projektion auf die untere Linie hat die Länge 1 (denn

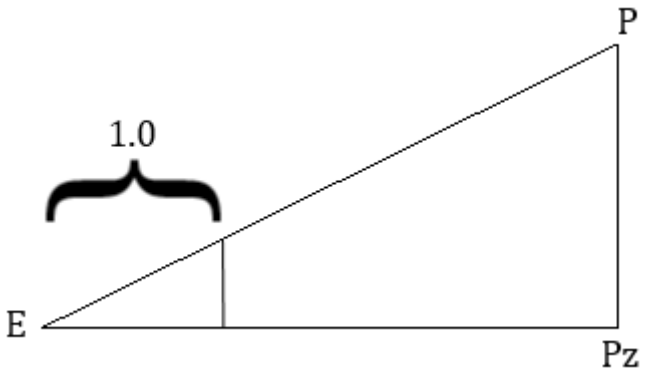

Figure 2: Tiefen-Dreieck

ihr z-Wert ist immer -1). Das heißt dann aber auch, dass $z_{eye}d_{eye}$ gerade unser Punkt P im Eye-Space ist!

Sobald wir dann also unsere Positionen und Normalen im Eye-Space haben können wir genau jene Berechnungen machen die wir vorher direkt im Geometrie-Pass machten, indem wir schlicht unseren G-Buffer benutzen. Das ist natürlich nur der Anfang; wesentlich interessantere Screen-Space Methoden wie z.B. SSAO/SSDO sind ebenfalls einfach zu implementieren sobald wir einen funktionierenden Deferred Renderer haben.