

Beispiel für eine Abschlussarbeit mit LATEX

Manfred Brill

12. Juli 2022

Inhaltsverzeichnis

1 Einleitung	3
2 Projektionen	5
2.1 Projektionen und Kameramodelle	5
3 OpenGL Extensions	7
3.1 OpenGL Extensions	7
3.2 GLSL und OpenGL	8
3.2.1 Vertex Processor	8
3.2.2 Fragment Processor	9
3.3 Unser erster <i>GLSL</i> -Shader	10
Literaturverzeichnis	11

Kapitel 1

Einleitung

Ein Beispiel für die Hauptdatei einer Diplomarbeit und die Aufteilung der einzelnen Kapitel in einzelne Dateien, die mit \input in die Hauptdatei `beispiel.tex` integriert werden.

Als Literaturdatenbank wird die Datei `arbeit.bib` verwendet. Die Abbildungen im Text sind sowohl im PNG- als auch im EPS-Format vorhanden. Das Beispiel ist also sowohl mit `latex → dvips → ps2pdf` als auch direkt mit `pdflatex` kompilierbar. Wenn Sie die Endung bei `includegraphics` weglassen entscheidet L^AT_EX selbst, welches Format passt.

Kapitel 2

Projektionen

In diesem Kapitel betrachten wir die virtuelle Kamera in *OpenGL* und verschiedene Parallel- und Zentralprojektionen. Eine gute Quelle ist [Ang03]-

2.1 Projektionen und Kameramodelle

Die Kamera in der Computergrafik können Sie sich als eine Lochkamera oder *Camera Obscura* vorstellen. Diese Metapher enthält die wesentlichen Eigenschaften, die so gut wie alle Kameramodelle in der Computergrafik auszeichnen:

- die Position der Kamera ist gegeben durch die Koordinaten *eines* Punkts in Weltkoordinaten;
- der Bildausschnitt ist rechtwinklig;
- der Schärfebereich der Kamera ist unendlich groß.

Die den Kameramodellen zu Grunde liegende Mathematik wird *Projektive Geometrie* oder *Darstellende Geometrie* genannt. Allerdings haben Sie in der Vorlesung nur einen kurzen Einblick in diese Fächer bekommen.

Ganz entscheidend ist die Unterscheidung zwischen Welt- und Kamerakoordinatensystem. Bevor eine Projektion, also eine Abbildung von drei- in einen zweidimensionalen Raum durchgeführt wird, werden die Koordinaten unserer Objekte mit Hilfe einer Basistransformationsmatrix in das Kamerakoordinatensystem umgerechnet. Die Kamera liegt im Ursprung dieses Koordinatensystems; das macht viele Berechnungen in der Bildsynthese später viel einfacher.

Um zwischen den verschiedenen Koordinatensystemen zu unterscheiden, haben wir vereinbart, dass die Achsen des Kamerakoordinatensystems mit **u**, **v** und **n** bezeichnet werden.

Natürlich ist es möglich, diese drei Richtungen in Weltkoordinaten anzugeben. Aber mit Hilfe des *Projektionsvektors* **p** und einem *View-up-Vektor* kann das Kamerakoordinatensystem berechnet werden:

1. n ist gegeben durch $n = -p$;
2. u ist orthogonal zur Ebene, die durch n und vup gegeben ist:

$$\mathbf{u} = \frac{\mathbf{vup} \times \mathbf{n}}{\|\mathbf{vup} \times \mathbf{n}\|};$$

3. v ist gegeben durch

$$\mathbf{v} = -\mathbf{u} \times \mathbf{n} = \mathbf{n} \times \mathbf{u}. \quad (2.1)$$

Die Formel (2.1) ist ein Beispiel für eine mathematische Formel mit einer Marke, auf die wir mit `eqref` Bezug nehmen können.

Das Original des Utah Teapot befindet sich inzwischen im Computer History Museum; in Abbildung 2.1 auf Seite 6 sehen Sie ein Ausstellungsfoto als Beispiel für eine Abbildung mit Bezug dazu im Text.

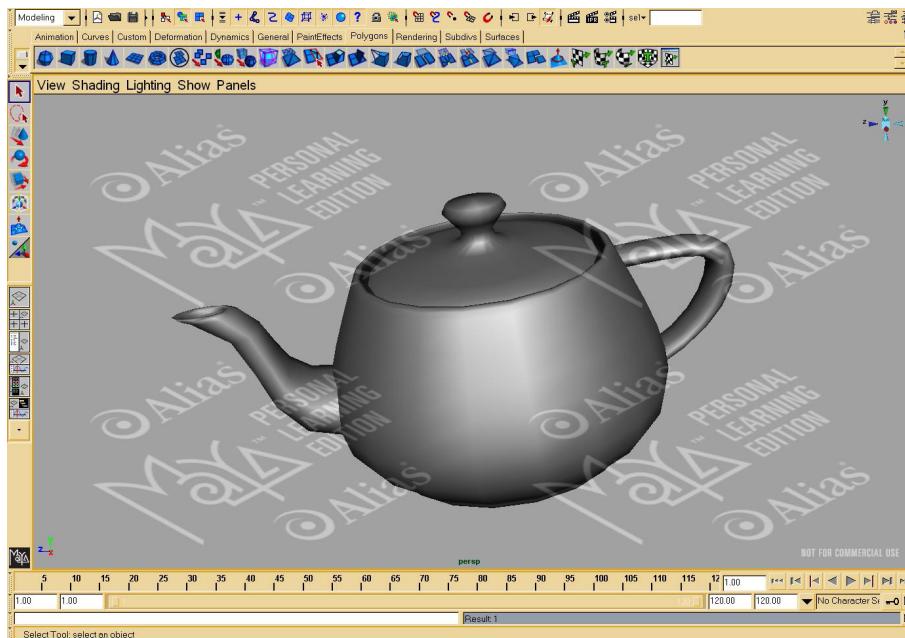


Abbildung 2.1: Der Utah Teapot in Maya

Kapitel 3

OpenGL Extensions

Dieses Kapitel enthält eine Einführung in OpenGL Extensions, wie sie verwaltet und verwendet werden. Mehr dazu finden Sie in [Ros06]

3.1 OpenGL Extensions

Die OpenGL *Extensions* bieten den Hardware-Herstellern die Möglichkeit, spezifische Erweiterungen im Treiber anzubieten. Es gibt inzwischen mehr als 300 solcher Erweiterungen. *Silicon Graphics* ist für das OpenGL Extension Registry zuständig.

Es gibt eine Systematik für die Namensvergabe der *Extensions*. Stammt die Erweiterung von einem Hersteller wie *NVidia* oder *ATI*, dann wird im Namen ein Prefix wie `GL_NV_` oder `GL_ATI_` aufgenommen. Wird eine Erweiterung von mehreren Herstellern angeboten, dann wird der Prefix `GL_EXT_` verwendet. Kommt das ARB zum Ergebnis, dass die Erweiterung eine sinnvolle Erweiterung von *OpenGL* darstellt wird der Prefix `GL_ARB_` vergeben. Dies ist häufig der letzte Schritt, bevor die Erweiterung in einer neuen Version von *OpenGL* zu einer Standard-Funktion wird. Dann fällt der Prefix weg. In Tabelle 3.1 finden Sie die wichtigsten Prefixes.

Tabelle 3.1: Eine Auswahl von OpenGL Extension Prefixes

Prefix	Bedeutung
<code>SGI_</code>	<i>Silicon Graphics</i>
<code>ATI_</code>	<i>ATI</i>
<code>NV_</code>	<i>NVidia</i>
<code>IBM_</code>	<i>IBM</i>
<code>WGL_</code>	<i>Microsoft</i>
<code>EXT_</code>	Herstellerübergreifend
<code>ARB_</code>	Vom ARB übernommen

Wollen Sie unter *Windows* mit *Extensions* arbeiten, dann reicht es nicht mit der `opengl32.dll` zu arbeiten, die Sie im Betriebssystem finden. Auch die Datei `libopengl32.a`, die wir in *Cygwin* verwenden enthält nur eine Version 1.1 von *OpenGL*. Deshalb benötigen wir einen Mechanismus, mit dessen Hilfe wir Pointer auf eine *OpenGL*-Funktion

erhalten, der direkt im Treiber existiert. Die Funktion `wglGetProcAddress()` in der Windows-Implementierung von OpenGL gibt diese Pointer zurück.

Als Beispiel suchen wir nach der Extension `GL_EXT_point_parameters`. Ist diese im Treiber vorhanden, dann erhalten wir einen Pointer auf die Funktion durch

```
#include <windows.h>
#include <GL/glut.h>
#include <GL/glext.h>
// Funktionsprototypen als globale Variable
PFNGLPOINTPARAMETERFEXTPROC glPointParameterfEXT;
...
glPointParameterfEXT =
(PFNGLPOINTPARAMETERFEXTPROC)
wglGetProcAddress("glPointParameterfEXT");
```

Nach dieser Zuweisung in der letzten Zeile können Sie `glPointParameterfEXT` wie gewohnt aufrufen.

3.2 GLSL und OpenGL

Bei der Implementierung eines GLSL-Shaders für eine OpenGL-Applikation haben Sie es mit mindestens zwei unterschiedlichen Dateien zu tun. Zum einen ist dies eine übliche OpenGL-Applikation, die die Shader lädt und aktiviert. Zudem muss in dieser Datei die Parameterübergabe zwischen OpenGL und GLSL definiert werden.

Ein *Vertex-Shader* hat bei uns immer den Suffix `*.vsh`; ein *Fragment-Shader* `*.fsh`. Der Quellcode liegt immer in einem Unterverzeichnis `shader`. Pro *Vertex- und Fragment-Shader* gibt es mindestens die Definition eines Einsprungspunkts; dafür verwenden wir immer eine `main`-Funktion.

3.2.1 Vertex Processor

Der *Vertex Processor* arbeitet auf den Eckpunkten und den dazu gehörenden Attributen. Er ist vorgesehen, um die folgenden Operationen durchzuführen:

- Transformation der Eckpunkte;
- Transformation der Eckennormalen und Normalisierung;
- Berechnung von Textur-Koordinaten;
- Transformation von Textur-Koordinaten;
- Beleuchtungsberechnungen für die Eckpunkte;
- Material und Farbzueweisung für die Eckpunkte.

Eingaben an den *Vertex Processor* werden in *GLSL Attributes* genannt. Das sind Farben und andere Eigenschaften, die zwischen `glBegin()` und `glEnd()` oder mit Hilfe der Funktion `glDrawElements` übergeben werden – prinzipiell alle Eigenschaften, die pro Eckpunkt wechseln können. Es gibt *Built-in attribute values*, die durch Aufrufe von `glColor()`, `glNormal()` oder `glVertex()` erzeugt werden. In *GLSL* greifen wir durch die Variablen `gl_Color` oder `gl_Normal` darauf zu. Darüber hinaus können Sie eigene Attribute definieren. Der *Vertex Processor* hat auch Zugriff auf die *ModelView-Matrix*, die Position von Lichtquellen oder die Position der Kamera. Solche Eigenschaften wechseln nicht von Eckpunkt zu Eckpunkt und werden deshalb in *GLSL* als *Uniform Variables* bezeichnet. Auch für diese Art von Variablen gibt es die Möglichkeit eigene Variable zu definieren und von der Anwendung an den *Vertex Shader* zu übergeben. Eine neue Möglichkeit in *GLSL* ist, dass die Möglichkeit besteht in einem *Vertex Shader* aus dem Texturspeicher zu lesen. Ein *Vertex Shader* wird pro Eckpunkt aufgerufen. Die Ausgabe eines solchen Shaders ist insbesondere die Position des aktuellen Eckpunkts in *Clipping Koordinaten*. Diese Position sollte nach Durchlauf des Shaders in der Ausgabe-Variablen `gl_Position` stehen.

3.2.2 Fragment Processor

Wie der Name schon sagt arbeitet der *Fragment Processor* auf den Fragmenten und den dazugehörenden Daten. Dieser Prozessor ist für die folgenden Operationen vorgesehen:

- Operationen und Berechnungen auf interpolierten Werten,
- Zugriff auf Texturen,
- Anwendung der Texturen,
- Nebel oder
- Farbberechnungen.

Fragment Shader können keine Operationen durchführen, die mehr als ein Fragment auf einmal benötigen. Die primäre Eingabe eines *Fragment Shaders* sind die interpolierten *varying* Variablen, sowohl die eingebauten als auch die von der Anwendung definierten. Innerhalb des Shaders kann mit Hilfe der Variablen `gl_FragCoord` die Position des Fragments festgestellt werden. Mit `gl_FrontFacing` kann abgefragt werden, ob die Werte des Fragments zu einem *Front Face* gehören. Der Shader kann auf alle *Uniform Variables* wie `gl_ModelViewMatrix` oder `gl_FrontMaterial` zugreifen. Dadurch werden Berechnungen ermöglicht, die normalerweise im *Vertex Processor* erfolgen würden.

Die hauptsächliche Ausgabe dieser Shader ist die Variable `gl_FragColor` – die Farbe des Fragments.

3.3 Unser erster *GLSL*-Shader

Jetzt wollen wir einen ersten kleinen *GLSL*-Shader erstellen. Dieser ist ganz unspektakulär, er färbt ein Dreieck. Mit dieser einfachen Funktionalität können wir uns auf die Verbindung zwischen C++-Programm und *GLSL* konzentrieren, bevor wir komplexere Shader implementieren.

Wir wollen ähnlich wie bei Ihrem ersten *OpenGL*-Beispiel ein gelbes Dreieck ausgeben. Allerdings soll die Farbe nicht durch `glColor`, sondern durch Shader definiert werden. Sehr häufig können Sie solche Aufgaben entweder als *Vertex*- oder als *Fragment*-Shader implementieren.

Als erste Lösungsmöglichkeit verwenden wir einen *Vertex Shader*. Die Ecken sollen wie in der Standard-Pipeline transformiert werden. Das könnten wir durch Anweisungen nachbauen, aber einfacher ist die Verwendung der Funktion `ftransform()`. Diese stellt die Funktionalität der *Fixed-Function-Pipeline* zur Verfügung. Bleibt die Definition der Farbe übrig, dafür wird der Wert auf `gl_FrontColor` übergeben. Sie sehen ein erstes Beispiel für die in *GLSL* eingebauten Funktionen für Felder.

```
void main(void)
{
    gl_FrontColor = vec4(1.0, 1.0, 0.0, 1.0);
    gl_Position = ftransform();
}
```

Einen *Fragment Shader* benötigen wir nicht. Jetzt müssen wir den Shader in das *OpenGL*-Programm integrieren. Dazu müssen wir unter Windows alle Pointer auf die Funktionen abfragen; wir gehen im Folgenden davon aus, dass dies bereits durchgeführt wurde.

...

Literaturverzeichnis

- [Ang03] ANGEL, EDWARD: *Interactive Computer Graphics - A Top-Down Approach with OpenGL*. Addison-Wesley, 2003.
- [Ros06] ROST, RANDI: *OpenGL Shading Language, 2nd Edition*. Addison-Wesley, 2006.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, [Vorname] [Name], geboren am [Geburts-Datum und -Ort], ehrenwörtlich,

- dass ich meine Bachelorarbeit/Masterarbeit mit dem Titel
[Titel]
selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe;
- dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

[Ausstellungsort], [Ausstellungsdatum]

[ausgeschriebene Unterschrift]

Sperrvermerk

Die vorliegende Bachelorarbeit/Masterarbeit **[Titel der Arbeit]** enthält zum Teil Informationen, die nicht für die Öffentlichkeit bestimmt sind. Der Inhalt darf daher nur mit der ausdrücklichen schriftlichen Genehmigung des Verfassers und **[Firmenangabe]** an Dritte weitergegeben werden.

Die Arbeit ist nur den Korrektoren sowie erforderlichenfalls den Mitgliedern des Prüfungsausschusses zugänglich zu machen.

[Ausstellungsort], [Ausstellungsdatum]

[Unterschriften]