Humboldt-Universität zu Berlin
Institut für Informatik
Informatik in Bildung und Gesellschaft

Computer Graphik Praktikums Kurzbericht Wikip3dia

Kornelius Kalnbach[1] Oliver Stadie[2]

February 17, 2010

[1]kalnbach@informatik.hu-berlin.de
[2]o.stadie@googlemail.com

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Grundlagen

1.1 Zielsetzung

Dies ist ein Kurzbericht zum Praktikum des Kurses “Computer Graphik” im Wintersemester 09/10 an der HU-Berlin. Unser Ziel bei diesem Praktikum war es, uns mit einer neuen Technologie, namens WebGL auseinander zu setzen. Diese Technologie soll es ermöglichen, 3D-Anwendungen in einem Browser darzustellen.

Um die Aufgabe etwas zu konkretisieren, nahmen wir uns vor Wikipedia als 3D-Gebäude darzustellen. Dabei sollte jeder Artikel in Wikipedia als ein Raum und jeder Link zu einem anderen Artikel als Tür und Tunnel zu einem weiteren Raum dargestellt werden. Dabei sollten die Räume je nach Anzahl der Links eine variable Größe haben und die Türen mit den Titeln der dahinter liegenden Räume beschriftet werden. Durch diesen Raum sollte dann wie in einem Ego-Shooter hindurch navigiert werden können.

1.2 Quellen

Als Grundlage für unsere Arbeit diente uns das Tutorial auf [learnWebGL]. Hier haben wir insbesondere auf der Grundlage von “Lesson 10: Loading a World, and the Most Basic Kind of Camera” unseren eigenen Code aufgebaut. Eine klare Trennung welcher Teil unseres Programms von uns stammt, ist nicht mehr möglich, da wir uns sehr viel auf die Codebeispiele von [learnWebGL] stützen, diese jedoch oft so weit umgeschrieben haben, dass diese kaum wieder erkennbar sind.

Klarer abgrenzen können wir uns jedoch von der mathematischen Funktionalität, vor allem für Matrizen- und Vektorrechnung.

Die eigentliche geistige Leistung unserer Arbeit liegt jedoch nicht in den einzelnen Funktionen und Codezeilen unseres Programms. Denn so gut wie jedes unserer graphischen Features, war bereits in anderer Form vorhanden und musste nur noch von uns verstanden und integriert werden. Die wirkliche geistige Leistung liegt in eben dieser Integration der Einzelkomponenten zu einem harmonierenden Programm.

2 Grundaufbau des Programms

Unser Programm schreibt, sobald es die benötigen Informationen von Wikipedia hat, die Vertex- und Textur-Koordinaten und die Normalen unseres Meshes in einen ArrayBuffer. Es werden zwar verschiedene Buffer für Lampen, Tunnel, Wände, usw. benutzt, jedoch werden diese immer vollständig in die Buffer geschrieben. Alternativ hätte man hier auch beispielsweise nur eine Tür in den Buffer schreiben können und diese dann durch Translation und Rotation mehrmals rendern können. Die Meshes werden nur einmal beim Laden jedes Raumes initialisiert.

In jedem Renderzyklus werden dann für jeden Buffer einige individuelle Einstellungen gemacht (z.B. Punktlicht und gerichtetes Licht in Tunneln ausschalten, Tiefenbuffertest für Lense Flare deaktivieren, usw.). Dann werden die Buffer an den Vertexshader

gesendet. Dieser verarbeitet die Vertices und sendet die Daten interpoliert an den Fragmentshader weiter. Nachdem die Shader alle Daten gerendert haben, können diese auf dem Bildschirm ausgegeben werden und ein neuer Renderzyklus kann beginnen.

Folgende Ordner, Dateien und Methoden implementieren den oben beschriebenen Teil unseres Programms:

- Die Kommunikation mit der Wikipedia-API erfolgt im Ordner `/wikipedia` und liefert im Wesentlichen für einen gegebenen Artikel-Titel eine Liste mit davon abgehenden Artikel-Titeln (und Bildern, welche nicht in das 3D-Modell integriert wurden) zurück. Für den ersten Raum liefert uns dieser Ordner die Daten für einen zufälligen Raum/Artikel.
- Die Buffer werden in verschiedenen Dateien erstellt:
  - `initLightMeshes()` in `lights.js` generiert und initialisiert die Buffer für Lampen und für den Lense Flare und weitere benötigte Daten für die Punktlichtquellen.
  - `initRoomMeshes()` in `room.js` generiert und initialisiert die Buffer für Wände, Boden, Decke, Türbeschriftungen, Tunnel und Türen.
  - `initContentMeshes()` in `contenObjects.js` generiert und initialisiert die Buffer für Kisten zufälliger Größe und deren Fake Shadows, die gleichmäßig im Raum verteilt werden.
  - `initFakeShadows()` in `world.js` generiert und initialisiert den Buffer des Fake Shadows unter dem Spieler. Generiert auch die runden und eckigen Fake Shadow Texturen selbst (diese werden nicht aus einer Datei geladen).
- Der Renderzyklus spielt sich hauptsächlich in `draw.js:drawScene()` ab. Hier werden für das Zeichnen jedes Meshes die Randbedingungen konfiguriert (z.B. besonderes Alpha-Blending aktiviert) und alle wichtigen Daten an die Shader gesendet (neben den Meshes vor allem die Model-View-Matrix und die Perspective-Matrix).
- `index.html` definiert die HTML-Seite und das HTML-Canvas-Element in dem die Anwendung am Ende zu sehen ist. Hier ist auch der Vertex- und der Fragmentshader in OpenGL® Shading Language (GLSL) definiert. Außerdem ruft `index.html` die erste Methode unseres Programms auf, sobald die Seite fertig geladen ist.

3 Features

In diesem Abschnitt sollen kurz unsere Erfahrungen aus dem Praktikum mit Elementen aus der Computer Graphik aufgelistet und reflektiert werden.

Hier gibt es eine alphabetisch sortierte Kurzbeschreibung unserer graphischen Features und welche Stellen im Code relevant für das jeweilige Feature sind.

Abbildung 1: Alpha-Blending: Obwohl die Texturen des Lense Flares und der Türbeschriftung rechteckig sind, scheinen Wand und Decke dahinter durch.

3.1 Alpha-Blending

... wird in `draw.js:drawScene()` an den benötigten Stellen aktiviert. Im Detail für folgende Meshes: Fake Shadows, Türbeschriftungen, spiegelnder Boden, Lense Flare. Für Lense Flare wird ein spezielles Alpha-Blending verwendet, bei dem die Lichtquelle zum Farbwert hinzu addiert wird, damit der Hintergrund heller wird, statt gemischt zu werden. Siehe auch Abbildung 1 auf Seite 5.

Als Quelle diente hierbei vor allem Lesson 8 aus [learnWebGL].

3.2 Ambient Lighting

Die RGB-Werte werden als uniform Parameter an den Fragmentshader übergeben. Dieser addiert dann diese Werte zu denen jedes Fragments um es entsprechend aufzuhellen. Siehe auch Abbildung 2 auf Seite 6.

Als Quelle diente hierbei vor allem Lesson 7 aus [learnWebGL].

3.3 Backface Culling

... wurde fürs Zeichnen der im Boden gespiegelten Kisten benutzt, um deren Rückseiten nicht zu zeichnen. Wird durch einen einfachen Aufruf in `draw.js:drawScene()` aktiviert:

```
gl.enable(gl.CULL_FACE);
```

3.4 Directional Lighting

Die RGB-Werte und Richtung werden als uniform Parameter an den Fragmentshader übergeben. Dieser addiert dann, je nach Einfallswinkel mit Phong Shading, die übergebenen Farbwerte zu denen jedes Fragments um es entsprechend aufzuhellen. Siehe auch Abbildung 3 auf Seite 7.

Abbildung 2: Ambient Lighting: Die Tunnel zwischen den Räumen werden nicht mit Punktlicht beleuchtet, weswegen das Ambiente Licht (und auch das gerichtete Licht) dort besonders gut zu sehen ist.

Als Quelle diente hierbei vor allem Lesson 7 aus [learnWebGL].

3.5 Fake Shadow

Die Schatten werden einfach als Meshes erzeugt (`world.js:initFakeShadows()`und `contentObjects.js:initContentMeshes()`) und mit halb-transparenten Texturen gerendert (in `draw.js:drawScene()`). Dazu wird Alpha-Blending benutzt. Siehe auch Abbildung 4 auf Seite 7.

3.6 Generierte Texturen

Die Texturen für die Fake Shadows werden in `world.js:initFakeShadows()` generiert. Die Türbeschriftungen werden in `world.js:createRoomForPage()` als eine einzige große Textur generiert, von der immer der passende Teil verwendet wird, da wir nur 32 Texturen zur Verfügung haben, was andernfalls für 500 Türen zu wenig wäre. Siehe auch Abbildung 5 auf Seite 8.

3.7 Lense Flare

In `lights.js:initLightMeshes()` werden die Meshes als 3 orthogonale Quadrate erzeugt und dann jeweils mit einer halb-transparenten Textur gerendert. Dazu muss vor

Abbildung 3: Directional Light: Unterschiedlich helle Wände trotz symmetrisch angeordneter Lichtquellen sind ein Resultat des gerichteten Lichtes.

Abbildung 4: Fake Shadow: Quadratisch unter den Kisten, rund unter dem Spieler.

Abbildung 5: Generierte Texturen (Transparenz in weiß dargestellt): Fake Shadow des Spielers, Fake Shadow der Kisten, Türbeschriftung

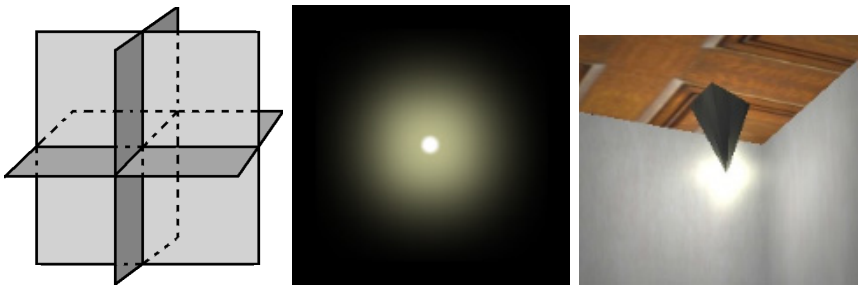

Abbildung 6: Lense Flare: Anordnung des Meshes, Textur (Transparenz ist schwarz dargestellt), Screenshot

dem Rendern (in `draw.js:drawScene()`) der Tiefenbuffertest ausgeschaltet werden, da die Ebenen sich gegenseitig verdecken. Siehe auch Abbildung 6 auf Seite 8.

3.8 Mip Mapping

Beim Initialisieren jeder Textur muss der Grafikkarte nur einmal mitgeteilt werden, dass Mip Mapping für diese aktiviert werden soll. Dies geschieht in `giles.js:handleImageTexture()`:

```
// use mipmap for minification
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);
gl.generateMipmap(gl.TEXTURE_2D);
```

3.9 Nebel

Im Fragmentshader wird jedes Fragment, je nach Entfernung zur Kameraposition, mit einer Nebelfarbe gemischt. Dabei ist die Gewichtung des Nebels bei entfernten Fragments größer, als bei nahen. Siehe auch Abbildung 7 auf Seite 9.

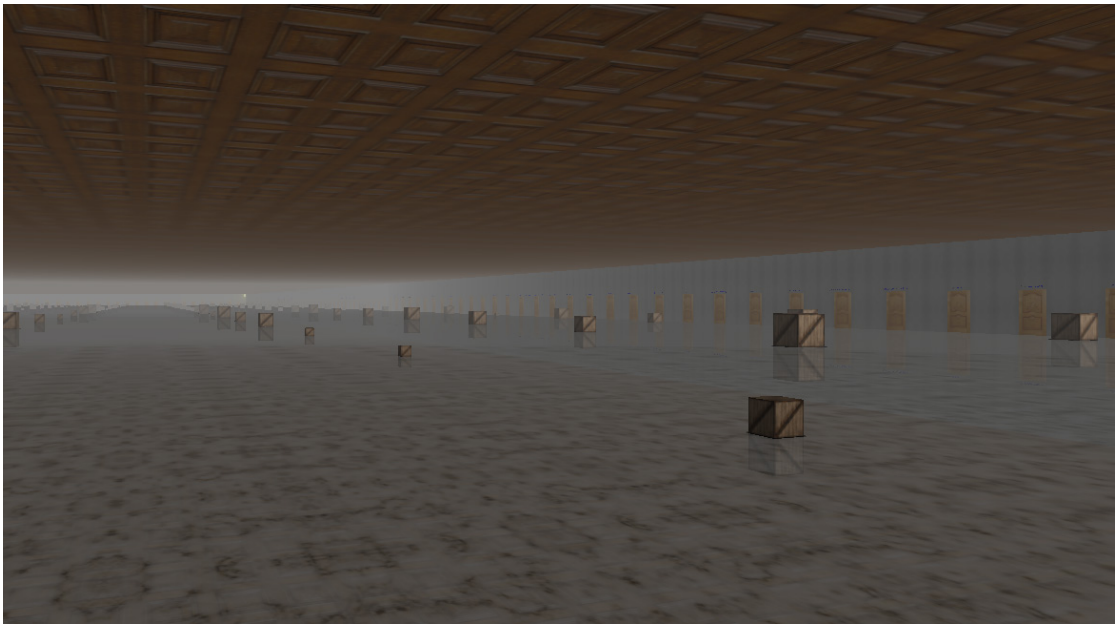

Abbildung 7: Nebel färbt entfernte Fragmente grau ein.

3.10 Phong Shading

Beim Erzeugen der Meshes wird für jeden Vertex auch eine Normale berechnet (diese zeigt immer Orthogonal aus dem Dreieck). Diese Normalen werden bis an den Fragmentshader weitergereicht (und zwischen den Vertices interpoliert). Dort werden sie dann für Directional Light und Point Light benutzt, wobei ein großer Winkel zwischen Normale und Richtung zur Lichtquelle für schwaches Aufhellen und geringer Winkel für starkes Aufhellen sorgt. Siehe auch Abbildung 8 auf Seite 10

Als Quelle diente hierbei vor allem Lesson 13 aus [learnWebGL].

3.11 Point Lighting

Die Positionen und Farben der einzelnen Lichtquellen werden in `lights.js:initLightMeshes()` erzeugt und in Arrays gespeichert. Diese werden dann an den Fragmentshader weitergegeben. Dieser addiert die Lichtwerte dann mittels Phong Shading zu den einzelnen Fragmenten. Jede Lichtquelle hat eine maximale Reichweite und einen linearen Abfall der Lichtintensität, die in dieser Entfernung genau 0 erreicht.

Als Quelle diente hierbei vor allem Lesson 12 und 13 aus [learnWebGL].

3.12 Reflektion

Die Reflektion des Raumes im Marmorboden geht über die Lektionen des Tutorials hinaus. Wir realisieren Reflektion in `draw.js:drawScene()`. da der Stencil Buffer leider nicht zuverlässig funktionierte, mussten wir ohne ihn auskommen. Interessanterweise ist

Abbildung 8: Phong Shading: Beim Boden und der Decke handelt es sich um 2 Meshes aus je 2 riesigen Dreiecken. Das Licht ist zentral im Raum. Mit Gouraud Shading wäre der Boden und die Decke unbeleuchtet, wir interpolieren zwischen den Vertices jedoch die Normalen und nicht die resultierende Lichtintensität.

es aufgrund der Szenengeometrie auch unnötig, die Reflektionen am Rand des Bodens abzuschneiden.

Der Effekt basiert auf dem üblichen Trick, die Szene gespiegelt zu zeichnen:

```
mvMatrix = mvScale(mvMatrix, 1, -1, 1);
```

Zudem benutzen wir Alpha-Blending: Zuerst wird der Boden gezeichnet, dann der Tiefenbuffer umgeschaltet und die Szene gespiegelt. Danach zeichnen wir den Boden nochmals, allerdings mit einem festem Alpha-Wert. Je transparenter diese zweite Schicht gezeichnet wird, desto deutlicher erscheint die Reflektion.

3.13 Shader

Da WebGL auf OpenGL ES 2.0 basiert, haben wir Shader anstatt "festverdrahteter" Graphik Pipelines benutzen. Wir haben genau einen Vertex- und einen Fragmentshader, welche beide in `index.html` definiert sind. Dazu mussten wir die Shadersprache GLSL benutzen.

Literatur

[learnWebGL] Thomas, Giles: *Learning WebGL. ...lessons 'n' links...,* 2009. Internet: http://learningwebgl.com/blog/ [Februar 2010].