Einfuehrung in OpenGL und Shader

Michael Bayer

28. September 2005

Inhaltsverzeichnis

1 OpenGL

1.1 Einführung

OpenGL ist eine plattformübergreifende Umgebung zur Entwicklung von sowohl zwei- als auch dreidimensionalen Grafikapplikationen. Seit seiner Einführung 1992 hat sich OpenGL zur am weitesten unterstützten und weitverbreitetsten API für Grafikprogrammierung entwickelt, und ist auf jeder Art von Endgeräten von Desktopcomputern bis hin zu eingebetteten Systemen zu finden. (siehe auch [1])

Seit 1992 wird der OpenGL-Standard vom *Architecture Review Board (ARB)* überwacht und erweitert. Das ARB ist ein unabhängiges Konsortium bestehend aus führenden Anbietern aus dem Grafiksektor.

1.2 Architektur

OpenGL definiert eine sehr einfache API welche nur grundlegende Funktionalität zur Erstellung von 3D-Applikationen bietet. Das Hostsystem auf dem eine solche Applikation ausgeführt werden soll muss eine Implementation dieser API vorhalten, eine freie Implementation bietet Mesa ([5]), allerdings ist diese nicht offiziell zertifiziert, Mesa hat sich aber trotzdem zumindest auf unixoiden Betriebssystemen als Quasi-Standard etabliert. Die Aufgabe der Implementierenden Bibliothek ist es im Grunde nur die Bibliotheksaufrufe des Benutzers an den ebenfalls vom System bereitgestellten, herstellerspezifischen Treiber weiterzuleiten, welcher diese dann in einer für die jeweilige GPU verständlichen Form an die Grafikkarte weiter gibt. Diese Architektur lässt den Grafikkartenherstellern möglichst große Freiheit in Design und Implementierung der nötigen Funktionen, auch lässt sie dem Benutzer die freie Wahl der implementierenden Bibliothek.

OpenGL arbeitet sowohl auf Bilddaten als auch auf geometrischen Primitiven. Beide Arten von Daten werden über unabhängige, programmierbare Pipelines verarbeitet und erst im letzten Schritt auf der Grafikkarte zusammengefügt und in den Frame-Buffer geschrieben (Siehe Abbildung 1.1).

1.3 Pipeline Schritte

Die Applikation setzt zunächst einige interne Variablen (vgl. 1.4), dann sendet sie Informationen über Vertices und deren Verbindungen untereinander (connectivity) an

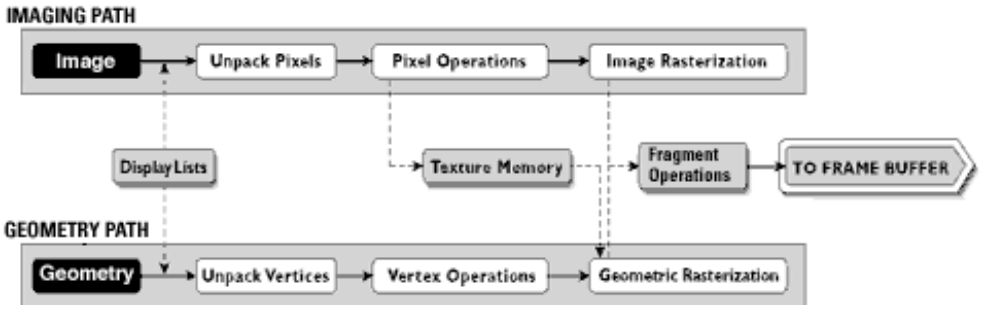

Abbildung 1.1: Die OpenGL Visualisierungspipeline (aus [1])

die Grafikkarte. Jeder Vertex besitzt bestimmte Attribute wie seine Position im Raum, seine Farbe, Texturkoordinaten, Normale usw.

Im ersten Schritt werden diese Attribute zusammengefasst und es werden Operationen auf ihnen ausgeführt, so zum Beispiel Positionstransformationen, per-Vertex Beleuchtungsberechnungen oder die Generierung und Transformation der Texturkoordinaten.

Dann werden aus den Vertices und den Konnektivitätsdaten Primitive zusammengesetzt. Diese werden am Frustum geculled und geclippt, d.h. später nicht sichtbare Primitive werden an dieser Stelle schon aus den Berechnungen entfernt. Dann werden diese Primitive auf die Projektionsebene projiziert und es werden die Positionen der Pixel im späteren Bild berechnet (Rasterisation). Dieser Schritt hat folglich zwei Ausgabe-ströme: die oben beschriebenen Positionen der Pixel und die interpolierten Farbwerte (Fragmente) derer.

Die Fragmenten werden im nächsten Schritt weiter behandelt, zum Beispiel mit einer Textur versehen oder mittels Nebels aufgehellt. Dieser Schritt ergibt einen Farb- und einen Tiefen-wert für jedes Fragment.

Im letzten Schritt wird schließlich aus den Fragmenten und deren Positionsdaten das letztendliche Bild zusammengesetzt, nachdem noch einige Tests auf den Fragmenten stattgefunden haben, wie zum Beispiel Alpha-, Stencil, Depth oder Scissor-Tests.

1.4 Zustände und der Kontext

OpenGL besteht im Grunde aus zwei Gattungen von Funktionen:

Die einen ändern den Zustand von internen Variablen wie zum Beispiel den Transformationsmatrizen, den aktiven Lichtquellen oder der gewählten Textur, die anderen schicken Geometriedaten (Streams) an die Grafikkarte. Letztere werden im Programm durch

```
glBegin(GLenum mode);
```

Listing 1.1: glBegin

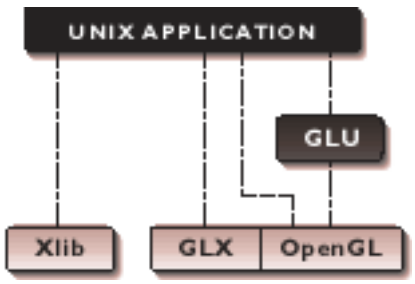

Abbildung 1.2: API-Hierarchie auf einem UNIX System (aus [1])

und

```
glEnd();
```

Listing 1.2: glEnd

zu Blöcken zusammengeschlossen um es dem Treiber zu ermöglichen Optimierungen auf Geschwindigkeit vorzunehmen. Innerhalb dieser Blöcke dürfen keine Änderungen am internen Zustand OpenGLs vorgenommen werden.

Ein vollständiger Satz dieser internen Zustände wird Kontext genannt. Um OpenGL überhaupt nutzen zu können muss für ein Fenster in welches OpenGL zeichnen soll ein solcher Kontext erzeugt und vorgehalten werden. Dies muss ebenfalls vom Hostsystem erledigt werden.

1.5 Bibliotheken

Wie bereits angesprochen bietet OpenGL nur eine sehr eingeschränkte Menge von Methoden mit die alle nur sehr grundlegende Funktionen abbilden. Damit nicht jeder Benutzer aufbauend auf diesen Methoden Standardfunktionalität implementieren muss wurden eine Vielzahl von Bibliotheken entwickelt welche das Benutzen von OpenGL vereinfachen sollen. Auch stellt OpenGL keinerlei Methoden zur Erzeugung von Fenstern, des Kontextes oder zur Behandlung von Benutzereingaben an. Die Abbildungen 1.2 und 1.3 sollen die Hierarchie der APIs auf verschiedenen Systemen verdeutlichen. Im Folgenden sollen drei dieser Bibliotheken vorgestellt werden:

1.5.1 GLU

GLU ist die *OpenGL Utility Library*. Sie soll Grafikfunktionen höherer Ebene wie die Abbildung zwischen Bildschirm- und Weltkoordinaten, die Generierung von Texture-Mipmaps, das Zeichnen von Quadriken sowie Transformationen zur einfacheren Erstellung und Verwaltung von Projektionsebenen und Kamera anbieten. GLU wird wie auch OpenGL selbst vom ARB spezifiziert und ist Teil eines vollständigen OpenGL-Paketes. Bis zur Version 1.2 existierte eine Mesa-eigene Implementation [5], welche allerdings ab Version 1.3 zugunsten der SGI-Implementation [6] aufgegeben wurde.

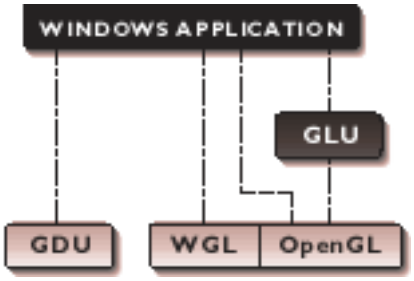

Abbildung 1.3: API-Hierarchie auf einem Windows System (aus [1])

1.5.2 GLut

GLut steht für *OpenGL Utility Toolkit*, und ermöglicht es, Aufgaben wie die Erstellung von Fenstern oder Tastaturein- und -Ausgaben plattformunabhängig zu erledigen. Daneben bietet GLUT einige Funktionen zum Zeichnen einfacher geometrischer Objekte wie Würfel, Zylinder, Kugeln oder der Utah-Teekanne.

GLUT setzt auf OpenGL, GLU und, je nach Betriebssystem, auf den jeweiligen betriebssystemspezifischen Funktionen und Bibliotheken auf, die die Verbindung zu OpenGL herstellen (GLX für X, WGL für Windows und AGL für Mac OS). (aus: [7])

1.5.3 GLew

Die GLew (*The OpenGL Extension Wrangler Library*, [8]) ist eine plattformunabhängig C/C++ Bibliothek die das Laden und Verwalten von Extensions (vgl. 1.6) vereinfacht. Sie bietet effiziente Methoden um zur Laufzeit zu ermitteln welche Extensions vom System unterstützt werden.

1.6 Extensions

Obwohl die OpenGL-Spezifikation eine bestimmte Grafikpipeline definiert bietet sie Herstellern eine einfache Möglichkeit ihre spezifische Implementation um benötigte Features zu erweitern: den *Extension Mechanism*. Dieser Mechanismus ist Fluch und Segen zugleich: Segen weil der Spezifizierungsvorgang durch das ARB zeitraubend ist und OpenGL durch solche Extensions auf dem neuesten Stand der Technik bleiben kann, Fluch weil Hersteller diesen Mechanismus missbrauchen um auch das Letzte an Leistung aus ihrer Hardware herauszuholen und der Benutzer somit solche Features für verschiedene herstellerspezifische APIs separat implementieren muss, was OpenGL ja gerade zu verhindern sucht.

Zum Glück werden Extensions welche sich bei den Benutzern großer Beliebtheit erfreuen und ihre Robustheit im Feld bewiesen haben ständig vom ARB in den Standard übernommen, so dass dieser Mechanismus auch der schnelleren Weiterentwicklung

von OpenGL dient. Bibliotheken wie die GLew vereinfachen zudem den Zugriff und helfen einen Weg durch den Extension-Wald zu bahnen.

Eine komplette Liste der aktuellen Extensions findet man in der *OpenGL Extension Registry* [9].

1.7 Praxis: Initialisierung mit GLut

Der erste Schritt zur funktionierenden OpenGL-Applikation ist die Erzeugung eines Fensters in welches gerendert werden soll, sowie des Kontextes. Wir wollen hierzu das GLut nutzen:

Die Dateien init.h und init.cpp enthalten die wichtigsten Funktionen:

```
void initGL(int argc, char *argv[]);
void resize(int w, int h);
void renderScene();
```

Listing 1.3: init.h

initGL enthält den nötigen Code um ein Fenster zu erstellen, den Kontext zu erzeugen und GLut mitzuteilen welche Callbacks es nutzen soll:

```
void initGL(int argc, char *argv[]) {

    // initialize glut
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(10, 10);
    glutInitWindowSize(800, 600);
    glutCreateWindow("gltut");

    // set global states
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    // set callbacks
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(resize);
    glutKeyboardFunc(handleKey);
}
```

Listing 1.4: init.cpp: initGL

Im ersten Abschnitt initialisieren wir das GLut selbst, erzeugen den Kontext und ein Fenster mit einer Größe von 800x600 Pixel, dem Titel gltutünd der Position (10,10). Der zweite Abschnitt setzt lediglich einige Variablen im Kontext, während im dritten

Abschnitt Callbacks angegeben werden welche GLut zum rendern bzw. während des Idle-Loops, beim Verändern der Größe des Fensters und bei Tastendruck aufgerufen werden sollen.

resize ist die Callbackfunktion die vom GLut aufgerufen wird sobald sich die Größe des Fensters geändert hat.

```
void resize(int w, int h) {
    // calc width/height ratio
    float ratio = 1.0* w / h;

    // set matrix mode
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // set new viewport and perspective
    glViewport(0, 0, w, h);
    gluPerspective(45, ratio, 1, 1000);

    // restore matrix mode
    glMatrixMode(GL_MODELVIEW);
}
```

Listing 1.5: init.cpp: resize

Nachdem das Verhältnis von neuer Breite zu neuer Höhe berechnet wurde wird OpenGL mitgeteilt dass nachfolgende Transformationsoperationen auf der Projektionsmatrix ausgeführt werden sollen und diese mit einer Identitätsmatrix geladen. Hiernach wird der neue Viewport gesetzt und mit der GLU-Funktion gluPerspective die Projektionsmatrix gesetzt. Danach wird der Matrixmodus zurückgesetzt.

renderScene ist die Methode die jeden Frame aufgerufen wird um den Inhalt zu zeichnen.

```
void renderScene() {
    // clear buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // load identity matrix into modelview matrix
    glLoadIdentity();
    // position camera
    gluLookAt(eye[0], eye[1], eye[2],
              cnt[0], cnt[1], cnt[2],
              0.0f,   1.0f,   0.0f);

    // init a light source
    glLightfv(GL_LIGHT0, GL_POSITION, l0pos);

    // draw the teapot
    glutSolidTeapot(1);
```

```
    // swap back and framebuffer
    glutSwapBuffers();
}
```

Listing 1.6: init.cpp: renderScene

Zunächst wird der aktuelle Buffer geleert, dann wird die aktuelle (Modelview) Matrix mit der Identitätsmatrix überschrieben. Nun wird mittels der GLU-Funktion gluLookAt die Kamera positioniert, eine Lichtquelle gesetzt sowie die bekannte Teekanne gezeichnet. Zu guter Letzt werden die Buffer ausgetauscht.

Nun wollen wir diese Funktionen zusammenführen:

```
int main(int argc, char *argv[]) {

    // initialize OpenGL
    initGL(argc, argv);

    // start main loop
    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Listing 1.7: main.cpp

Dies lässt sich nun sehr einfach bewerkstelligen: wir rufen die initGL Methode auf um OpenGL zu initialisieren, danach nutzen wir die GLut-Funktion glutMainLoop um in die Hauptschleife des Programms zu kommen.

Abbildung 1.4: Das erste OpenGL Programm

2 Shader in OpenGL

Shader sind kleine Programme die direkt auf der Hardware der Grafikkarte ausgeführt werden. Generell unterscheidet man zwischen Vertex- und Pixelshadern, wobei erstere - wie der Name impliziert - auf den Vertices operieren und deren Attribute verändern können, und letztere auf den Bildpunkten der fertig gerenderten Szene - in der OpenGL-Terminologie Fragments - operieren und somit deren Farbe ändern können.

Betrachten wir zunächst einmal eine vereinfachte Abbildung der oben besprochenen Visualisierungspipeline - Abbildung 2.1 - wie ordnen sich Shader dort ein?

Die intuitive Antwort auf diese Frage ist auch die Richtige: Vertex Shader greifen im Schritt der Vertex Transformationen, Fragment Shader während der Färbung in die Pipeline ein. Vertex Shader ersetzen die Transformationen des statischen Teils der Pipeline, Fragment Shader ersetzen den Schritt der Einfärbung der traditionellen Pipeline komplett.

Wozu noch statische Transformationen und Färbungen mag man sich fragen, wo man doch theoretisch alles direkt per Shadern machen könnte? Diese Frage ist durchaus berechtigt, denn eigentlich könnten Shader durchaus die traditionellen Pipelines komplett implementieren, die Frage ist aber falsch gestellt: warum nicht?

Durch das Hinzunehmen von Shadern in die Pipelines wird Legacy-Applikationen nicht geschadet, sie funktionieren weiterhin. Der Benutzer der nur Shader nutzen will kann die traditionelle Pipeline einfach ignorieren.

Es gibt viele Möglichkeiten Shader in OpenGL zu nutzen, drei sollen an dieser Stelle kurz vorgestellt werden:

2.1 ARB Programs

Das ARB hat Mitte 2002 zwei eigene Spezifikationen zu Vertex- beziehungsweise Fragment-Shadern veröffentlicht: die *ARB_vertex_program* und *ARB_fragment_program* Extensions. Diese definieren jeweils eine Art Pseudo-Assembler den alle konformierenden GPUs sprich Treiber implementieren müssen.

Diese beiden Assembler sind sehr mühsam zu benutzen da sie fast keine Features bieten die man von Hochsprachen gewöhnt ist. Aber sie werden von fast allen Herstellern (und vor allem von Ati und NVIDIA) unterstützt. Außerdem zeigen sich hier typische Konzepte der inneren Funktion von Grafikkarten.

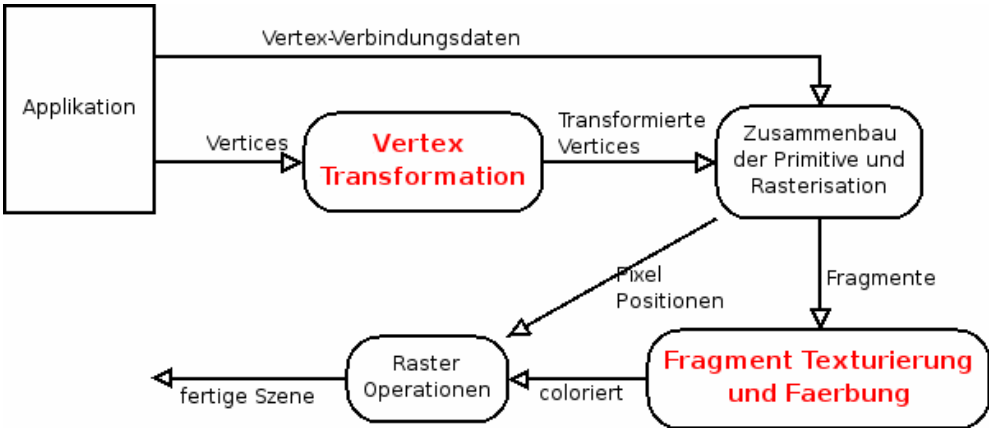

Abbildung 2.1: Vereinfachte Darstellung der OpenGL Visualisierungspipeline

2.2 Cg - C for Graphics

Cg steht für *C for Graphics* und ist eine von NVIDIA ([10]) entwickelte Hochsprache. Cg basiert - wie der Name schon andeutet - auf C und nutzt beinahe identische Syntax zur Deklaration von Funktionen und Variablen. Erweitert wurde diese Syntax allerdings um Methoden zur einfachen Behandlung von komplexen Datentypen die für Grafik-Anwendungen typisch sind.

Viele Entwickler sehen in Cg die beste Alternative zur Programmierung von Shadern, allerdings werden die hierfür nötigen Extensions fast ausschließlich von NVIDIA-Karten unterstützt.

Erwähnenswert ist diese Sprache dennoch da sie weit verbreitet ist, und da der Autor zur Entwicklung der Shader in diesem Dokument ein Programm aus dem *Cg Toolkit* ([11]) nutzte, den cgc (*Cg Compiler*). Aufgrund der robusten Codebasis des NVIDIA-Treibers ist es diesem Compiler möglich verschiedene Eingabe-sprachen wie GLSL oder eben Cg in verschiedenste Ausgabesprachen zu wandeln.

2.3 GLSL - OpenGL Shader Language

Die *OpenGL Shader Language* basiert auf ANSI C, wurde aber ebenfalls um Vertex- und Matrix-Typen erweitert, um einen einfacheren Umgang mit den typischen Operationen in Grafik-Applikationen.

GLSL ist Teil der OpenGL 2.0 Spezifikation, allerdings unterstuetzen noch nicht viele Systeme diese Version vollstaendig. Abhilfe schaffen die beiden Extensions *GL_ARB_fragment_shader* und *GL_ARB_vertex_shader*.

GLSL wird ebenfalls von den meisten Herstellern unterstuetzt, und ist deswegen der wahrscheinlichste Kandidat fuer eine Standardsprache fuer Shader in OpenGL-Applikationen.

3 ARB Program Shader

Mit dem bis jetzt erläuterten Wissen ist es uns nun möglich die ersten Shader zu schreiben. Hierzu müssen wir unser Programm noch um einige Funktionen erweitern:

3.1 Praxis: Extensions mit GLew

Um Shader in OpenGL nutzen zu können sind vielfach Extensions nötig. Folglich ist es nötig die Fähigkeiten des Hostsystems zu testen. Hierzu wollen wir den GLew (1.5.3) nutzen, dazu erweitern wir die initGL Funktion um den einige Zeilen die der Initialisierung dienen:

```
    // init GLew
    GLenum err = glewInit();
    if (err != GLEW_OK) {
        ERR("glewInit returned err: " << glewGetErrorString(err));
    } else {
        LOG("using GLEW version: " << glewGetString(GLEW_VERSION));
    }
```

Listing 3.1: init.cpp: initGL

glewInit initialisiert den GLew, falls dieses fehlschlägt geben wir eine Fehlermeldung aus und beenden das Programm, ansonsten schreiben wir die GLew-Version auf die Konsole. Nun können wir testen ob das Hostsystem die für uns interessanten Extensions unterstützt. Dazu bietet der GLew eine Vielzahl von Möglichkeiten, unter anderem definiert er für jede Extension ein Makro das einfach auf Wahrheit getestet werden kann, mehr dazu später.

3.2 Praxis: Laden von Shadern

Unser Programm soll zur Laufzeit beliebige Shader aus Textdateien laden können um nicht jedes mal wenn wir einen Shader ändern unser Programm neu compilieren zu müssen. Hierzu definieren wir eine Funktion die den Dateinamen des zu ladenden Shaders in einem std::string entgegen nimmt und einen char* zurückgeliefert der den Inhalt der Textdatei enthält.

```
char* readTextFile(const std::string& fn);
```

Listing 3.2: textfile.h

Die Einzelheiten der Implementation sollen dem Leser hier erspart werden, sie befindet sich in der Datei textfile.cpp.

Die Endung der zu ladenden Textdateien haben keinerlei inhärente Bedeutung, wir werden uns für dieses Dokument auf folgende Erweiterungen einigen:

.avp - ARB Vertex Program

.afp - ARB Fragment Program

.avs - ARB Vertex Shader (GLSL)

.afs - ARB Fragment Shader (GLSL)

Nun definieren wir eine weitere Funktion welche das eigentlich Laden der Shader übernehmen soll.

```
void loadShader(const std::string& fn);
```

Listing 3.3: shader.h: loadShader

Diese Funktion nimmt einen std::string entgegen der den Dateinamen enthält, versucht die Endung der Datei zu finden und ruft dann die entsprechende Funktion zum Laden des Shaders auf. Auch hier werden wir die Implementationsdetails nicht erörtern, der geneigte Leser mag sie in der Datei shader.cpp nachlesen.

3.3 ARB Vertex Programs

Wie bereits erwähnt nutzt die ARB_vertex_program-Extension eine Art Assemblersprache, welche in der Spezifikation (auffindbar in der Extension Registry [9]) definiert wird.

Um ARB Vertex Programs zu laden wird - wie bereits oben beschrieben - innerhalb von *loadShader()* die Funktion *loadVertexProgram()* aufgerufen:

```
// check for ARB_vertex_program extension support
if (!GLEW_ARB_vertex_program) {
    ERR("system doesn't support ARB_vertex_program extension!");
}

char* pgmstr = readTextFile(fn);
unsigned int program;
// bind shader
glGenProgramsARB(1, &program);
glBindProgramARB(GL_VERTEX_PROGRAM_ARB, program);
// load shader
glProgramStringARB(GL_VERTEX_PROGRAM_ARB,
                   GL_PROGRAM_FORMAT_ASCII_ARB,
```

```
                    strlen(pgmstr),
                    pgmstr);
// check for errors
if (glGetError() == GL_INVALID_OPERATION) {
    int errorpos;
    const char* errorstr;

    glGetIntegerv(GL_PROGRAM_ERROR_POSITION_ARB, &errorpos);
    errorstr = (const char*)glGetString(GL_PROGRAM_ERROR_STRING_ARB);

    ERR("error while loading ARB_vertex_program from " << fn << " at pos " <<
}

// save handle
AVPHandle = program;
AVPLoaded = true;

// free text memory
delete[] pgmstr;
```

Listing 3.4: shader.cpp: loadShader

Zunächst wird mittels des von GLew gesetzten Makros überprüft ob die Extension *ARB_vertex_program* unterstützt wird. Danach wird der Shader in ein char-Array geladen und auf dem Stack Platz für einen unsigned int geschaffen, welcher mittels *glGenProgramsARB* zum Speichern eines Handles für das zu ladende Programm verwandt wird. *glBindProgramARB* bewirkt dass das soeben generierte Programm gebunden wird, ähnlich wie bei Texturen. Schließlich wird der Shader mittels *glProgramStringARB* geladen, und zwar in das gerade gebundene Handle.

Mittels *glGetError* wird nun geprüft ob beim Laden ein Fehler auftrat, sollte das der Fall sein wird der Fehlerstring ausgelesen, ausgegeben und die Anwendung etwas unsanft beendet.

3.4 ARB Fragment Programs

Die Assemblersprache der ARB_fragment_program-Extension ist in weiten Teilen Identisch zu der der ARB_vertex_program-Extension.

Auch das Laden von ARB Fragment Programs läuft nahezu identisch ab, weswegen an dieser Stelle kein Sourcecode gezeigt werden soll. Es werden lediglich einige OpenGL-Konstanten verändert, genaueres kann in der Datei shader.cpp nachgelesen werden.

3.5 Praxis: Binden von Shadern

Um ARB Shader auch nutzen zu können ist es nötig OpenGL mitzuteilen welche Shader auf den aktuellen Stream angewandt werden sollen. Dies geschieht ganz ähnlich dazu wie man Texturen bindet. Hierzu definieren wir eine weitere Funktion, *bindShaders()*:

```
void bindShaders() {
    // bind and enable ARB_vertex_shader?
    if (AVPLoaded) {
        glBindProgramARB(GL_VERTEX_PROGRAM_ARB, AVPHandle);
        glEnable(GL_VERTEX_PROGRAM_ARB);
    }
    // bind and enable ARB_fragment_shader?
    if (AFPLoaded) {
        glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, AFPHandle);
        glEnable(GL_FRAGMENT_PROGRAM_ARB);
    }

}
```

Listing 3.5: shader.cpp: bindShaders

In dieser wird geprüft ob Vertex- oder Fragment- Programs geladen sind und je nach Ergebnis werden *glBindProgramARB* und *glEnable* mit den jeweiligen Parametern aufgerufen. Diese Funktion rufen wir nun innerhalb von *renderScene()* aufgerufen, und zwar bevor wir den teapot zeichnen.

4 GLSL

Auch für GLSL Shader muss der Programmcode um einige Funktionen erweitert werden:

4.1 GLSL Vertex Shader

GLSL Shader werden zwar nicht in Pseudo-Assembler sondern in einer Hochsprache verfasst, aber auch sie werden aus Textdateien geladen.

Das Laden von GLSL-Shadern gestaltet sich etwas komplizierter als das Laden von ARB Vertex Programs:

```
// check for ARB_vertex_shader extension support
if (!GLEW_ARB_vertex_shader) {
    ERR("system doesn't support ARB_vertex_shader extension!");
}

// create handle
AVSHandle = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
// read program
char* pgmstr = readTextFile(fn);
const GLcharARB* pgms = pgmstr;

// generate shader source
glShaderSourceARB(AVSHandle, 1, &pgms, NULL);
// compile shader
glCompileShaderARB(AVSHandle);

// free text memory
delete[] pgmstr;
```

Listing 4.1: shader.cpp –loadVertexShader

Nachdem wir überprüft haben ob der geladene OpenGL-Treiber die *ARB_vertex_shader* Extension unterstützt wird ein Program-Handle vom Typ *GLhandleARB* erzeugt, und zwar mittels *glCreateShaderObjectARB*, welchem der Typ des zu erzeugenden Handles mitgegeben wird, in diesem Fall ein Vertex Shader Handle. Das eigentliche Laden der Textdatei gestaltet sich identisch zum Laden von ARB Shadern. Sobald der Shader-Source im Speicher ist wird dieser mittels *glShaderSourceARB* an das Handle gebunden und mittels *glCompileShaderARB*

kompiliert. Hiernach wird brav der Speicher für den Source wieder freigegeben. Zur Fehlerbehandlung später mehr.

4.2 GLSL Fragment Shader

Ähnlich wie bei ARB_fragment_programs ist auch bei ARB_fragment_shadern die Sprache und der Ladevorgang weitgehend identisch. Deswegen soll dem Leser auch hier eine langwierige Erläuterung des hierzu notwendigen Sources erspart bleiben.

4.3 Datentypen und Variablen

GLSL bietet verschiedene Möglichkeiten Variablen zu definieren, diese können folgende Typen annehmen:

4.3.1 Datentypen

Folgende simple Datentypen werden angeboten:

- float
- bool
- int

Diese Typen entsprechen den von OpenGL definierten Typen GLfloat, GLint und GLbool und verhalten sich wie erwartet.

Des weiteren können 2-, 3- oder 4-Komponentige Vektoren dieser Typen definiert werden:

- vec2, vec3, vec4
- bvec2, bvec3, bvec4
- ivec2, ivec3, ivec4

Quadratische Matrizen von Fließkommazahlen können in drei verschiedenen Größen definiert werden:

- mat2
- mat3
- mat4

Für den Umgang mit Texturen kann auf vier spezielle Typen zurückgegriffen werden:

sampler1D - für eindimensionale Texturen

sampler2D - für zweidimensionale Texturen

sampler3D - für dreidimensionale Texturen

samplerCube - für Cubemaps

Arrays werden in GLSL genau so definiert wie in C, können allerdings nicht bei der Deklaration initialisiert werden.

Auch Strukturen können definiert werden, auch hier ist die Syntax mit der C-Syntax identisch:

```
struct foobar {
    float foo;
    vec3  bar;
};
```

Listing 4.2: Strukturen in GLSL Shadern

4.3.2 Variablen

Einfache Variablen können wie gewohnt deklariert werden

```
float foo, bar;
int baz = 5;
bool bla = false;
```

Listing 4.3: einfache Variablen in GLSL

Allerdings gibt es in GLSL - anders als zum Beispiel in C - keine automatischen Typecasts beim initialisieren von Variablen, stattdessen wird stark auf das Konzept von Konstruktoren zurückgegriffen, dem von C++ nicht unähnlich:

```
float a = 2;                    // falsch - kein automatisches typecasting
int b = 2;                      // korrekt
float c = float(b);             // korrekt - c = 2.0f
vec2 d = vec2(.5f, 1.0f);       // korrekt
vec3 e = (d, c);                // korrekt
```

Listing 4.4: Strukturen in GLSL Shadern

Wie leicht zu sehen ist gibt es eine weite Reihe von Konstruktoren, dies gilt insbesondere für Matrizen. Diese sollen allerdings an dieser Stelle nicht alle aufgeführt werden.

Für Variablen gibt es außerdem einige Qualifier die hier kurz erläutert werden sollen:

const - Kompilierzeitkonstante, kann nicht geändert werden

uniform - globale Variable welche von der Applikation an den Shader übergeben wird, kann in Vertex und Fragment Shadern gelesen, aber nicht geschrieben werden.

attribute - globale Variable welche von OpenGL an den Vertex Shader übergeben wird, kann nur in Vertex Shadern gelesen, aber nicht verändert werden.

varying - Variable zum Austausch interpolierter Daten vom Vertex Shader zum Fragment Shader.

4.4 Praxis: Linken von GLSL Programmen

Im Gegensatz zu ARB Programs müssen die Handles von GLSL Shadern nach dem kompilieren gelinkt werden, und zwar zu einem weiteren *GLhandleARB*, dem eigentlichen Programm welches dann auf der Grafikkarte ausgeführt werden kann. Dies geschieht in einer eigenen Funktion - *linkShaders* - welche nach dem Laden der Shader aufgerufen wird:

```
// genarate shader program handle
if (AVSHandle || AFSHandle)
    ShaderHandle = glCreateProgramObjectARB();
// attach vertex shader
if (AVSHandle)
    glAttachObjectARB(ShaderHandle, AVSHandle);
// attach fragment shader
if (AFSHandle)
    glAttachObjectARB(ShaderHandle, AFSHandle);
// link program
glLinkProgramARB(ShaderHandle);

// print info logs
printInfoLog(AVSHandle);
printInfoLog(AFSHandle);
printInfoLog(ShaderHandle);
```

Listing 4.5: shader.cpp –linkShaders

Sobald festgestellt wurde dass zumindest ein GLSL Shader geladen wurde wird - diesmal mittels *glCreateProgramObjectARB* ein Handle erstellt, an welches mit der Funktion *glAttachObjectARB* die jeweiligen Shader gebunden werden. Schließlich wird mit *glLinkProgramARB* das Programm erzeugt. Anschließend wird mit der Funktion *printInfoLog* zu jedem Handle das so genannte Info Log ausgegeben:

```
void printInfoLog(GLhandleARB obj) {
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
```

```
    glGetObjectParameterivARB(obj, GL_OBJECT_INFO_LOG_LENGTH_ARB,
            &infologLength);

    if (infologLength > 0) {
        infoLog = (char *)malloc(infologLength);
        glGetInfoLogARB(obj, infologLength, &charsWritten, infoLog);
        LOG("Infolog for " << obj);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```

Listing 4.6: shader.cpp –printInfoLog

Hier wird mittels der Funktion *glGetObjectParameterivARB* zum übergebenen Handle die Länge des entstandenen Info Logs ermittelt. Dann wird Speicher dafür reserviert und das Log mit *glGetInfoLogARB* in diesen geschrieben. Nachdem das Log ausgegeben wurde wird der reservierte Speicher wieder freigegeben.

4.5 Binden von GLSL Shadern

Schließlich müssen wir noch dafür sorgen dass unsere GLSL Shader auch angewandt werden. Hierzu müssen wir lediglich die Funktion *bindShaders* anpassen:

```
void bindShaders() {
    // bind and enable ARB_vertex_shader?
    if (AVPLoaded) {
        glBindProgramARB(GL_VERTEX_PROGRAM_ARB, AVPHandle);
        glEnable(GL_VERTEX_PROGRAM_ARB);
    }
    // bind and enable ARB_fragment_shader?
    if (AFPLoaded) {
        glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB, AFPHandle);
        glEnable(GL_FRAGMENT_PROGRAM_ARB);
    }

    // use GLSL?
    if (ShaderHandle)
        glUseProgramObjectARB(ShaderHandle);

}
```

Listing 4.7: shader.cpp: bindShaders

Hier wurde lediglich der letzte Block hinzugefügt welcher, nachdem überprüft wurde ob ein gültiges Program Handle vorliegt, mittels der Funktion *glUseProgramObjectARB* OpenGL anweist dieses auf den aktuellen Stream anzuwenden.

4.6 Kommunikation Applikation und Shader

GLSL bietet der Applikation mehrere Möglichkeiten mit dem Shader zu kommunizieren, allerdings ist dies immer eine ein-Weg-Kommunikation, der Shader kann also keine Berechnungen an die Applikation zurückgeben. Natürlich gibt es hier Ausnahmen - so gibt es zum Beispiel durchaus Kanäle zum Debuggen - auf diese soll aber im Rahmen dieses Dokumentes nicht eingegangen werden.

Die einfachste Möglichkeit aus der Applikation mit dem Shader zu kommunizieren ist einfach über den OpenGL-Kontext (state). Dieser steht dem Shader in vollem Umfang zur Verfügung, allerdings ist das nicht immer der sinnvollste Weg, zudem der Kontext keine beliebigen Felder hat, und das missbrauchen von nicht benötigten Feldern des Kontextes würde wohl von den meisten Programmierern aus äußerst unsauber empfunden werden.

Glücklicherweise erlaubt GLSL die Definition von Benutzerdefinierten Variablen die die Applikation schreiben und auf diese Weise mit dem Shader kommunizieren kann. Hierzu werden zwei der Qualifier welche in 4.3.2 beschrieben wurden verwendet, nämlich *attribute* und *uniform*.

Uniform-Variablen zeichnen sich dadurch aus dass sie innerhalb eines Streams - sprich innerhalb eines *glBegin()* und *glEnd()* Blockes - nicht geändert werden können, dies impliziert auch dass sie nicht geeignet sind Vertexattribute zu beinhalten. Sie sind deshalb nur geeignet Werte zu halten welche über einen Stream, eine komplette Szene oder gar einen kompletten Frame konstant bleiben.

Attribute-Variablen hingegen können jederzeit geändert werden, allerdings ist das natürlich kostspieliger da der geladene Treiber keinerlei Optimierungen der Stream-Daten vornehmen kann. Auch können sie nur innerhalb eines Vertex Shaders gelesen werden da sie sich auf Vertex-Informationen beschränken und folglich im Fragment Shader nutzlos wären.

Ein anderer Weg auf dem die Applikation mit dem Shader kommunizieren kann ist die benötigten Daten in Texturen zu laden, da diese nicht unbedingt als Bilddaten vom Shader interpretiert werden müssen. Tatsächlich kann man so sehr einfach verschiedene Effekte erzeugen, zum Beispiel kann ein Vertex Shader welcher die Bilddaten als Höheninformation interpretiert sehr einfach ein Heightfield darstellen, oder ein Fragment Shader kann die Textur als Bump- oder Normalmap interpretieren.

4.7 Praxis: Kommunikation Applikation und Shader

Nun wollen wir wie in 4.6 beschrieben eine Uniform-Variable von unserer Applikation an den Shader weitergeben. Dazu muss erst die Position der Variablen im Speicher gefunden werden, dabei müssen wir darauf achten dass dies erst geschieht nachdem wir den Shader übersetzt und gelinkt haben, bei einigen Treibern ist es sogar notwendig dass der Shader dazu gebunden ist.

Ziel ist es dem Shader eine Variable vom Typ GLfloat zukommen zu lassen welche innerhalb einer Sekunde von 0.0f auf 1.0f steigt um dann erneut bei 0.0f zu beginnen. So ist es dem Shader möglich sein Verhalten über die Zeit unabhängig von der Ablaufgeschwindigkeit der Applikation zu verändern. Zu diesem Zweck definieren wir eine neue Funktion - *sendTime()* - und rufen diese in *renderScene()* nach dem Binden der Shader mittels *bindShaders* auf:

```
void sendTime() {
    // determine locate of uniform variable timeElapsed
    GLint loc = glGetUniformLocationARB(ShaderHandle, "timeElapsed");
    // some very basic error handling
    GLenum err = glGetError();
    if (err != GL_NO_ERROR)
        LOG("loc: " << loc << " err " << err << " " << gluErrorString(err)
    // calculate a float value between 0.0f and 1.0f
    GLfloat sec = (GLfloat)((clock() % CLOCKS_PER_SEC))/CLOCKS_PER_SEC;
    // pass it to the shade
    glUniform1fARB(loc, sec);
}
```

Listing 4.8: shader.cpp –sendTime

Die Variable welche wir dem Shader zur Verfügung stellen soll *timeElapsed* heißen. Die Ermittlung der Position dieser Variablen geschieht mittels der Funktion *glGetUniformLocationARB*, welche ein Handle nimmt in welchem die Variable deklariert wurde und einen nullterminierten String der den Namen der Variablen enthält. Nach einiger - zugegebenermaßen sehr rudimentärer - Fehlerbehandlung berechnen wir mittels der in *time.h* definierten Funktion *clock* den Wert welchen wir an den Shader übermitteln wollen und sende ihn schließlich mittels der Funktion *glUniform1fARB* an den Shader.

Auf diese und ähnlich Weise können auch andere Typen von Variablen gesetzt werden, das gezeigte soll aber an dieser Stelle ausreichen.

5 Shader

Nachdem der Boxcode zum Laden und Binden von Shadern nun fertig ist können wir endlich unsere eigenen Shader schreiben und ausführen. Alle Shader in diesem Kapitel sind stark von [13] inspiriert.

5.1 Monocolor Shader

Der wohl einfachste Shader ist der welcher die Vertices unverändert lässt und einfach alle Fragmente auf eine bestimmte Farbe setzt.

Der komplette Source dieses Shaders ist in den Dateien im Verzeichnis *shader/monocolor* zu finden. Ausgeführt wird er mittels

./gltut shader/monocol/monocol.avp shader/monocol/monocol.afp respektive

./gltut shader/monocol/monocol.avs shader/monocol/monocol.afs

5.1.1 ARB Program Monocolor Shader

Zunächst soll die Umsetzung dieses Vorhabens mittels der ARB Programs Extensions gezeigt werden:

```
!!ARBvp1.0
PARAM mvp[4] = { state.matrix.mvp };

DP4 result.position.x, mvp[0], vertex.position;
DP4 result.position.y, mvp[1], vertex.position;
DP4 result.position.z, mvp[2], vertex.position;
DP4 result.position.w, mvp[3], vertex.position;

END
```

Listing 5.1: monocolor.avp

ARB Vertex Programs werden immer mit *!!ARBvp* gefolgt von einer Versionsnummer eingeleitet. Danach wird hier eine Variable namens *mvp* bekannt gemacht welche vom Typ *PARAM* ist und somit eine Variable die von OpenGL vorbelegt wird. Variablen sind immer Vierkomponenten-Vektoren bestehend aus Floats. In diesem Fall handelt es sich um ein Array von vier dieser Vektoren die mit den Komponenten der ModelviewProjection-Matrix - dem Ergebnis der Multiplikation der Modelview- mit der Projektions-Matrix - vorbelegt werden.

Mit dem Befehl *DP4* wird ein Punktprodukt von zwei Vierkomponenten-Vektoren erzeugt und im ersten Operanden gespeichert.

Die Variable *vertex* enthält den Vertex auf dem der Shader gerade ausgeführt wird, die Variable *result* ist der Ergebnisvertex.

In diesem Shader wird die Position des Vertex einfach mit der ModelviewProjection-Matrix (ab jetzt kurz MVP genannt) multipliziert und in result gespeichert.

Das zugehörige Fragment Program gestaltet sich folgendermaßen:

```
!!ARBfp1.0
PARAM c[1] = { { 0, 1 } };
MOV result.color, c[0].xxyy;
END
```

Listing 5.2: monocolor.afp

ARB Fragment Programs werden immer mit *!!ARBfp* eingeleitet, auch hier gefolgt von einer Versionsnummer. Nachdem eine Variable namens *c* mit konstanten Werten vorbelegt wurde wird die Farbe des Ergebnisfragmentes *result* auf 0.0, 0.0, 1.0, 1.0 - also blau - gesetzt.

5.1.2 GLSL Monocolor Shader

Nachdem wir den Shader jetzt mittels der ARB Programs umgesetzt haben wollen wir ihn auch in GLSL schreiben:

```
void main() {
    vec4 v = vec4(gl_Vertex);

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.3: monocolor.avs

Es ist leicht ersichtlich dass GLSL deutlich eingängiger als der Pseudo-Assembler der ARB Programs ist.

GLSL lehnt sich stark an C an, somit heißt der Einsprungspunkt auch hier *main*, ist allerdings als *void* definiert. Der Vertex auf dem der Shader ausgeführt wird heißt hier *gl_Vertex*, die Position des Ergebnisvertexes heißt *gl_Position*.

Nun wird hier eine Variable vom Typ *vec4* definiert und mit der Position von gl_Vertex belegt. Die Ergebnisposition ergibt sich einfach aus der Multiplikation der MVP mit diesem Vektor.

Der Source des zugehörigen Fragment Shaders ist ähnlich einfach:

```
void main() {
        vec4 color = vec4(0.0, 0.0, 1.0, 1.0);

        gl_FragColor = color;
```

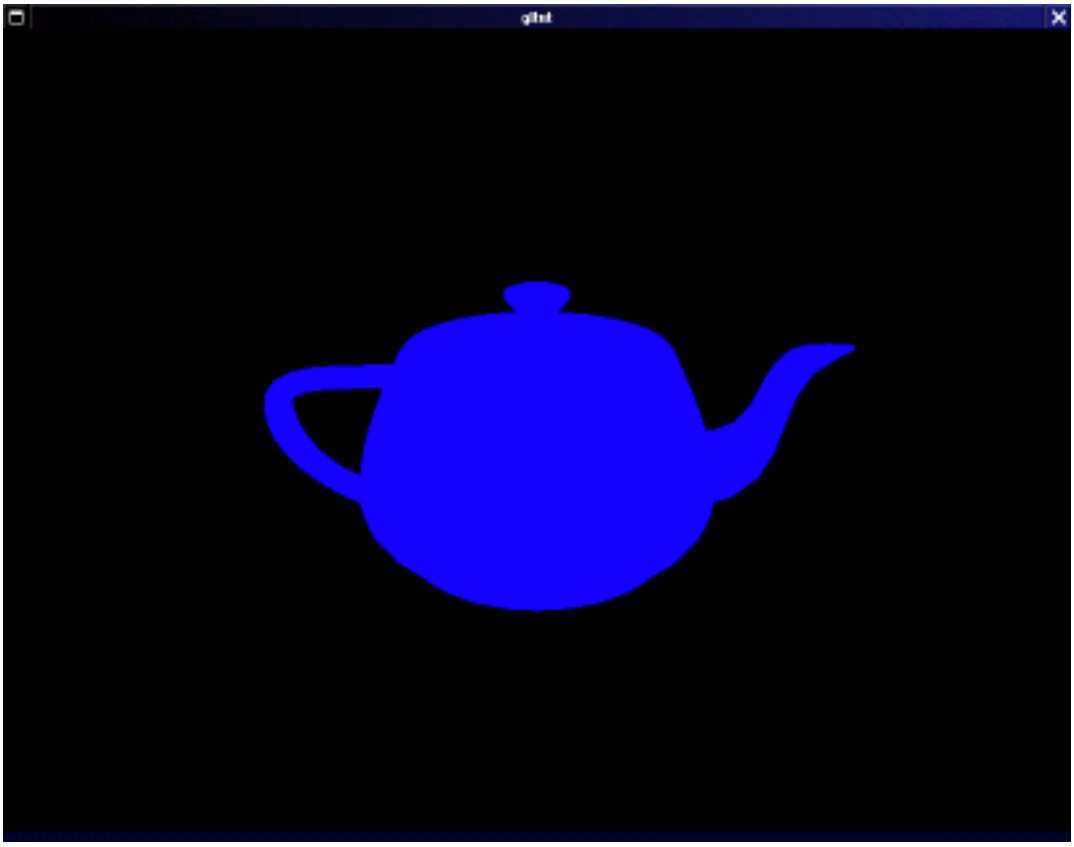

Abbildung 5.1: Der monocolor Shader in seiner gesamten Pracht

```
}
```

Listing 5.4: monocolor.avs

Auch hier heißt der Einsprungspunkt main, die Ausgabefarbe des Fragmentes auf dem der Shader ausgeführt wird heißt *gl_FragColor*. Wieder wird eine Variable definiert, allerdings wird sie hier einfach mit konstanten Werten für blau belegt. Die Ergebnisfarbe ergibt sich einfach aus diesen Werten.

5.2 Flat Shader

Unser erster nächster Shader soll nun auch Vertices im Stream verändern, das war ja der Sinn der Übung. Die einfachste Änderung ist das flachdrücken des Teapot, sprich alle Koordinaten einer Achse auf 0 zu setzen.

Der Source dieser Shader liegt im Verzeichnis *shader/flat*, ausgeführt wird er mittels ./gltut shader/flat/flat.avp shader/monocol/monocol.afp beziehungsweise mit ./gltut shader/flat/flat.avs shader/monocol/monocol.afs für die GLSL Version

5.2.1 ARB Program Flat Shader

Auch hier wollen wir zuerst den schweren Weg gehen und den Shader mittels der ARB Vertex Programs umsetzen:

```
!!ARBvp1.0
PARAM mvp[4] = { state.matrix.mvp };
TEMP v;

MOV v.x, vertex.position.x;
MOV v.y, 0.0;
MOV v.z, vertex.position.z;
MOV v.w, vertex.position.w;

DP4 result.position.x, mvp[0], v;
DP4 result.position.y, mvp[1], v;
DP4 result.position.z, mvp[2], v;
DP4 result.position.w, mvp[3], v;

END
```

Listing 5.5: flat.avp

Dieser Shader gestaltet sich schon deutlich komplizierter als sein Vorgänger. Nach dem üblichen Header und der Bekanntmachung der MVP wir eine Variable *v* definiert die wieder zur Berechnung der Ergebnisposition eingesetzt wird. Nun wird jede Komponente von v mit der entsprechenden Komponente aus der Vertexposition belegt, bis auf eine, welche einfach auf 0.0 gesetzt wird. Anschließend wird auf bekannte Weise die Ergebnisposition aus v und der MVP berechnet.

Als Fragment Shader nutzen wir einfach den monocolor Shader aus dem vorherigen Kapitel.

5.2.2 GLSL Flat Shader

Die Implementation dieses Shaders in GLSL ist wieder deutlich eingängiger:

```
void main() {
    vec4 v = vec4(gl_Vertex);

    v.y = 0.0;

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.6: flat.avs

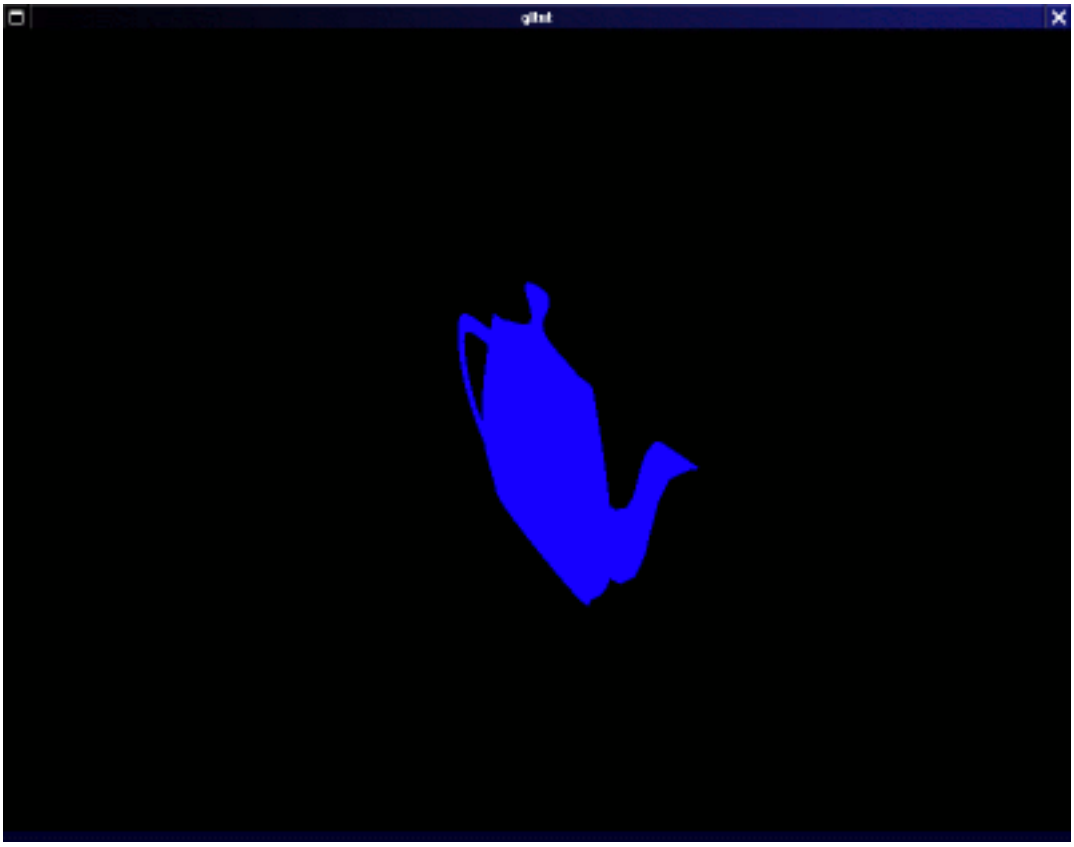

Abbildung 5.2: Der flat Shader

Der Unterschied zum Vertex Shader aus dem vorherigen Kapitel liegt hier einfach darin dass dem Vektor v aus dem die Ergebnisposition berechnet wird eine Komponente auf 0.0 gesetzt wird bevor wir ihn mit der MVP multiplizieren.

Auch hier verwenden wir wieder einfach den monocolor Fragment Shader aus dem vorherigen Kapitel. Abbildung 5.2 zeigt das Ergebnis.

5.3 Flag Shader

Nun wollen wir den Vertex Shader noch etwas interessanter gestalten, indem wir die zu verändernde Komponente der Position der Vektoren nicht nur auf eine Konstante setzen, sondern sie nach einer Sinusfunktion verschieben, was den Eindruck einer Fahne erzeugen soll.

5.3.1 GLSL Flag Shader

Die Umsetzung dieses Vorhabens ist mit der GLSL kein Problem, da sie für alle wichtigen geometrischen Funktionen eigene Methoden vordefiniert:

```
void main() {
    vec4 v = vec4(gl_Vertex);

    v.y = sin(5.0*v.x)*0.2;

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.7: flag.avs

Die Änderungen am Shader des letzten Kapitels nehmen sich wieder sehr gering aus, die zu verändernde Komponente des Vektors wird einfach mittels der Funktion sin() in Abhängigkeit vom Wert der ersten Komponente gesetzt. Die beiden eingeführten Faktoren (5.0 und 0.2) dienen lediglich dazu die Veränderung deutlicher zu machen.

Das resultierende Bild lässt den gewünschten Effekt zwar erkennen, aber leider nur undeutlich. Dies liegt daran, dass in der Szene trotz der von uns gesetzten direktionalen Lichtquelle die Beleuchtung komplett fehlt, da wir die fixed Function Pipeline mit unseren Shadern umgangen haben, folglich müssen wir uns jetzt selbst um die Beleuchtung kümmern.

5.4 Toon Shader

Beleuchtung bedeutet dass die einzelnen Fragmente eines Polygons je nach Lichteinfall in der Farbe verändert werden. Hierzu ist es notwendig den Winkel zu bestimmen in dem das Licht der Lichtquelle auf das Fragment trifft. Dies soll anhand eines ersten einfachen Versuches eine korrekte Beleuchtung zu simulieren demonstriert werden, dem Toon Shader:

5.4.1 GLSL Toon Shader

Um den angesprochenen Winkel zu berechnen muss die Richtung bekannt sein aus der das Licht kommt. Dies geschieht hauptsächlich im Vertex Shader:

```
varying vec3 lightDir,normal;

void main()
{
        lightDir = normalize(vec3(gl_LightSource[0].position));
        normal = gl_NormalMatrix * gl_Normal;

        gl_Position = ftransform();
}
```

Listing 5.8: toon.avs

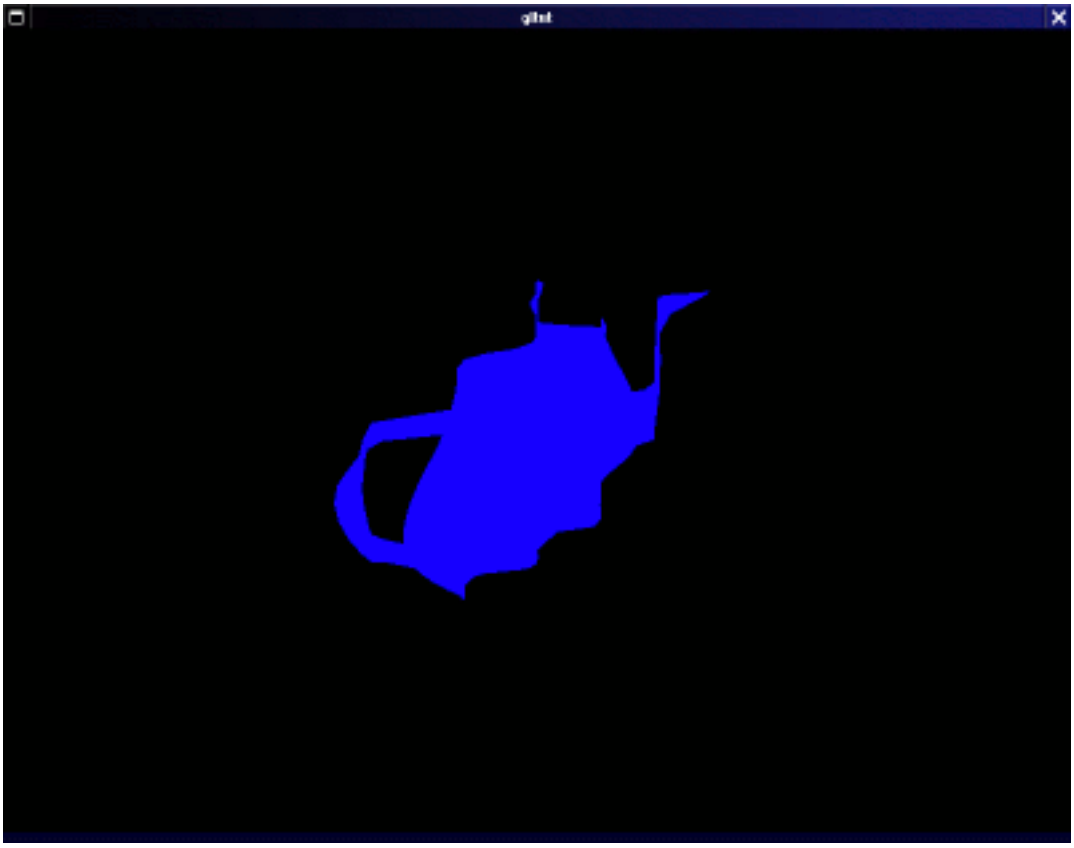

Abbildung 5.3: Der flag Shader setzt eine Komponente nach einer Sinusfunktion

Dazu definieren wir zwei *varying* Variablen (vgl. Kapitel 4.3.2), eine um die Richtung des einfallenden Lichtes zu speichern (lightDir), und eine für die Normale des Vertex auf welchem operiert wird (normal). Die von der Applikation definierten Lichtquellen hält OpenGL für uns in einem Array namens gl_LightSource vor. Die Richtung des Lichtes einer direktionalen Lichtquelle ist einfach ihre Position; um sicher zu gehen normalisieren wir diese also und errechnen so lightDir. Die Vektornormale stellt OpenGL für uns in der Variablen *gl_Normal* zur Verfügung, diese multiplizieren wir mit der ebenfalls von OpenGL gestellten *gl_NormalMatrix*. Nun muss noch die Ergebnisposition des Vertex berechnet werden, hierzu wollen wir die Funktion *ftransform* nutzen, welche einfach die Funktionalität der fixed Function Pipeline nachbildet.

Die eigentliche Beleuchtung findet dann im Fragment Shader statt:

```
varying vec3 lightDir,normal;

void main()
{

        float intensity;
        vec4 color;

        // normalizing the lights position to be on the safe side

        vec3 n = normalize(normal);

        intensity = dot(lightDir,n);

    if (intensity > 0.99)
                color = vec4(0.9, 0.9, 1.0, 1.0);
        else if (intensity > 0.95)
                color = vec4(0.7, 0.7, 1.0, 1.0);
        else if (intensity > 0.5)
                color = vec4(0.3, 0.3, 0.6, 1.0);
        else if (intensity > 0.25)
                color = vec4(0.2, 0.2, 0.4, 1.0);
        else
                color = vec4(0.1, 0.1, 0.2, 1.0);

        gl_FragColor = color;
}
```

Listing 5.9: toon.afs

Hier machen wir zuerst die varying Variablen lightDir und normal bekannt. In main deklarieren wir dann eine Variable welche die Intensität des einfallenden Lichtes abhängig vom Einfallswinkel des Lichtes halten soll, und eine Variable *color* welche zur Berechnung der Ergebnisfarbe genutzt wird.

Abbildung 5.4: der Toon Shader, ein simpler Ansatz der Beleuchtung

Da normal eine interpolierte Variable ist normalisieren wir diese nochmal und speichern das Ergebnis in *n*. Dann berechnen wir die Intensität aus dem Punktprodukt der interpolierten Lichtrichtung und n. Anschließend setzen wir einfach abhängig von dieser Intensität die Farbe des Fragmentes in 5 Stufen. Das Ergebnis (./gltut shader/toon/toon.avs shader/toon/toon.afs, siehe Abb. 5.4) sieht schon deutlich plastischer aus als zum Beispiel der monocolor Shader.

5.5 Flagge, die 2.

Fuhren wir nun den eben geschriebenen Toon-Shader zusammen mit dem Flag-Shader aus Kapitel 5.3 aus ist das Ergebnis ziemlich unbefriedigend - das liegt daran dass der Flag-Shader die Variable normal nicht setzt. Das wollen wir ändern.

5.5.1 GLSL Flag Shader mit Normalen

Die Änderungen zum ersten Flag Shader nehmen sich wieder sehr gering aus:

```
varying vec3 lightDir,normal;
```

```
void main()
{
        lightDir = normalize(vec3(gl_LightSource[0].position));

    vec4 v = vec4(gl_Vertex);
    v.y = sin(v.x);

    vec3 n = normalize(vec3(1/cos(v.x), 1.0f, 0.0f));
    normal = gl_NormalMatrix * n;

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.10: flagnormals.avs

Im Grunde übernehmen wir nur die Variablen lightDir und normal aus dem Toon Shader, normal wird allerdings anders belegt. Führt man den Toon Fragment Shader jetzt mit diesem neuen Vertex Shader aus

./gltut shader/flag/flagnormals.avs shader/toon/toon2.afs

sieht das Ergebnis schon deutlich besser aus. Die jetzt noch vorhandenen Fehler kommen daher, dass GLUT während dem Zeichnen des Teapot leider den Kontext verändert, was sich auch auf die gl_NormalMatrix auswirkt. Das soll uns nicht weiter stören, schließlich wurden wir um eine Flagge zu zeichnen normalerweise nicht auf den teapot zurückgreifen.

5.6 Lighting

So - nachdem wir jetzt einen einfachen Ansatz der Beleuchtung implementiert haben wollen wir uns daran machen ein komplettes Beleuchtungsmodell zu implementieren wie es die fixed Function Pipeline bietet. Mehr noch - wir wollen nicht nur pro Vertex sondern sogar pro Pixel Beleuchten, einfach weil der Aufwand nur geringfügig höher ist da unsere Fragment Shader sowieso auf jedem Pixel ausgeführt werden. Algorithmen zum per Pixel Lighting sind vielfach beschrieben, ich halte mich in diesem Dokument an [3]. Allerdings werden wir uns auf direktionales Licht beschränken:

5.6.1 GLSL per Pixel Lighting

Zu erst der Vertex Shader:

```
varying vec4 diffuse, ambient;
varying vec3 normal, lightDir, halfVector;

void main() {
```

```
    // transform normal to eye space
        normal = normalize(gl_NormalMatrix * gl_Normal);

    // normalize light direction
        lightDir = normalize(vec3(gl_LightSource[0].position));

        // normalize the halfVector to pass it to the fragment shader
        halfVector = normalize(gl_LightSource[0].halfVector.xyz);

        // compute diffuse and ambient colors
        diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
    ambient = gl_FrontMaterial.ambient * (gl_LightSource[0].ambient + gl_Light

    // transform position
        gl_Position = ftransform();

}
```

Listing 5.11: perpixel.avs

Die Variablen normal und lightDir sollten noch aus den letzten Shadern bekannt sein, neu ist *halfVector*. Diese Variable dient der Übergabe des normalisierten Halbvektors - dem Vektor der genau zwischen Vektor vom Augenpunkt zum Vertex (eye vector) und dem Lichtvektor liegt - an den Fragment Shader. Der Halbvektor wird zur Berechnung des spekulaeren Lichtanteils (specular component) genutzt, eine Vereinfachung des Phong-Beleuchtungsmodelles welche zu erst beschrieben in [4]. Des weiteren deklarieren wir vier-komponentige Vektoren *diffuse ambient* die jeweils den diffusen und den ambienten Anteil des Lichtes in diesem Vertex speichern sollen.

Im eigentlichen Programm berechnen wir zunachst die Normale, die Lichtrichtung und den eben beschriebenen Halbvektor. Dann wird der diffuse Lichtanteil berechnet nachdem wir die diffuse Farbe des Materiales mit dem diffusen Anteil der Lichtquelle multiplizieren. Der ambiente Lichtanteil berechnet sich folglich ganz nach Phong aus der ambienten Farbe des Materiales multipliziert mit der Summe der ambienten Lichtanteile der Lichtquelle und des globalen Lichtmodells welches von der Applikation vorgegeben wird.

Schliesslich transformieren wir nur noch die Position des Vertexes auf bekannte Weise.

Der Fragment Shader ist der bis jetzt komplizierteste:

```
varying vec4 diffuse, ambient;
varying vec3 normal, lightDir, halfVector;

void main() {
        vec3 n, halfV, viewV, ldir;
        float NdotL, NdotHV;
```

```
    vec4 color = ambient;

        n = normalize(normal);

        // compute the dot product between normal and ldir
        NdotL = dot(n,lightDir);

        if (NdotL > 0.0) {
                halfV = normalize(halfVector);
                NdotHV = max(dot(n,halfV),0.0);
                color += gl_FrontMaterial.specular * gl_LightSource[0].specula
                color += diffuse * NdotL;
        }

        gl_FragColor = color;
}
```

Listing 5.12: perpixel.afs

Nachdem wir die im Vertex Shader geschriebenen Variablen bekannt gemacht haben kommen wir zur eigentlichen main-Funktion. Dort deklarieren wir einige lokale Variablen welche wir im weiteren Verlauf brauchen werden und da wir die varying Variablen nicht im Fragment Shader beschreiben koennen kopieren wir die interpolierte Normale und ambiente Farbe in ebenfalls lokale Variablen.

Nun da die Vorarbeit geleistet ist kommen wir zur eigentlichen Fragmentfarbe. Dazu berechnen wir zunachst das Punktprodukt zwischen der Normalen und der Lichtrichtung. Falls das Punktprodukt groesser ist als 0 muessen wir mehr tun als nur die ambiente Farbe zu setzen: In diesem Fall addieren wir zum ambienten Lichtanteil die nach Blinn berechneten spekulaeren und die diffusen Farben.

Fuehren wir nun diese beiden Shader auf unserem teapot aus (./gltut shader/lighting/perpixel.avs shader/lighting/perpixel.afs) ist das Ergebnis (Abb. 5.5) doch schon recht ansehnlich.

5.7 zurueck zur Flagge

Nun wollen wir das so eben gelernte auf unsere Flagge anwenden:

5.7.1 GLSL Flag Shader mit per Pixel lighting

Das Erstellen des Vertex Shaders fuer dieses Vorhaben besteht eigentlich nur darin dass wir den Flag Shader aus 5.5 um die noetigen Komponenten aus dem letzten Kapitel (5.6), namentlich dem Halbvektor und die interpolierten Farbanteile, erweitern.

```
varying vec4 diffuse, ambient;
varying vec3 normal, lightDir, halfVector;
```

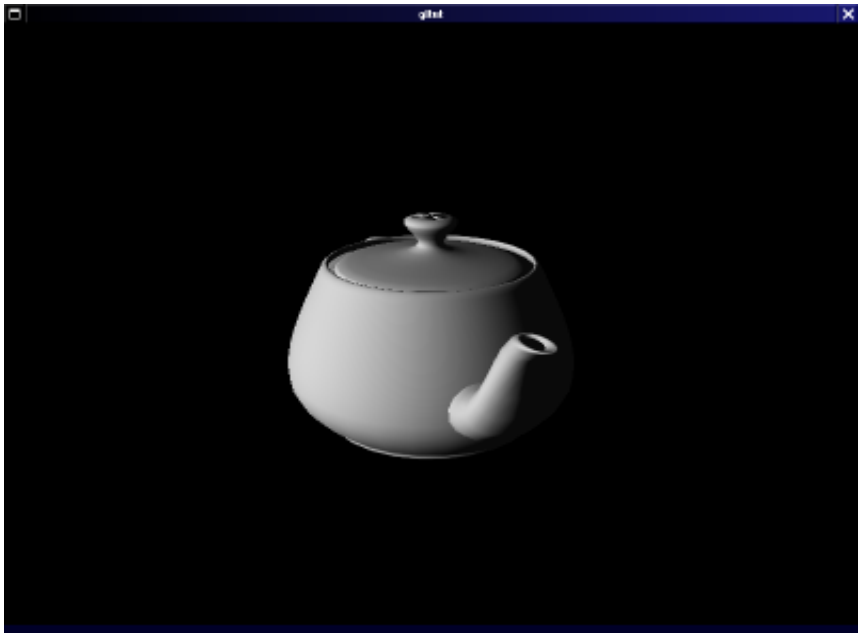

Abbildung 5.5: Per Pixel Lighting

```
void main()
{
    // normalize light direction
        lightDir = normalize(vec3(gl_LightSource[0].position));

    vec4 v = vec4(gl_Vertex);
    v.y = sin(v.x);

    vec3 n = normalize(vec3(1/cos(v.x), 1.0f, 0.0f));
    normal = normalize(gl_NormalMatrix * n);

    // normalize the halfVector to pass it to the fragment shader
    halfVector = normalize(gl_LightSource[0].halfVector.xyz);

    // compute diffuse and ambient colors
    diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
    ambient = gl_FrontMaterial.ambient * (gl_LightSource[0].ambient + gl_Light

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.13: flaglighting.avs

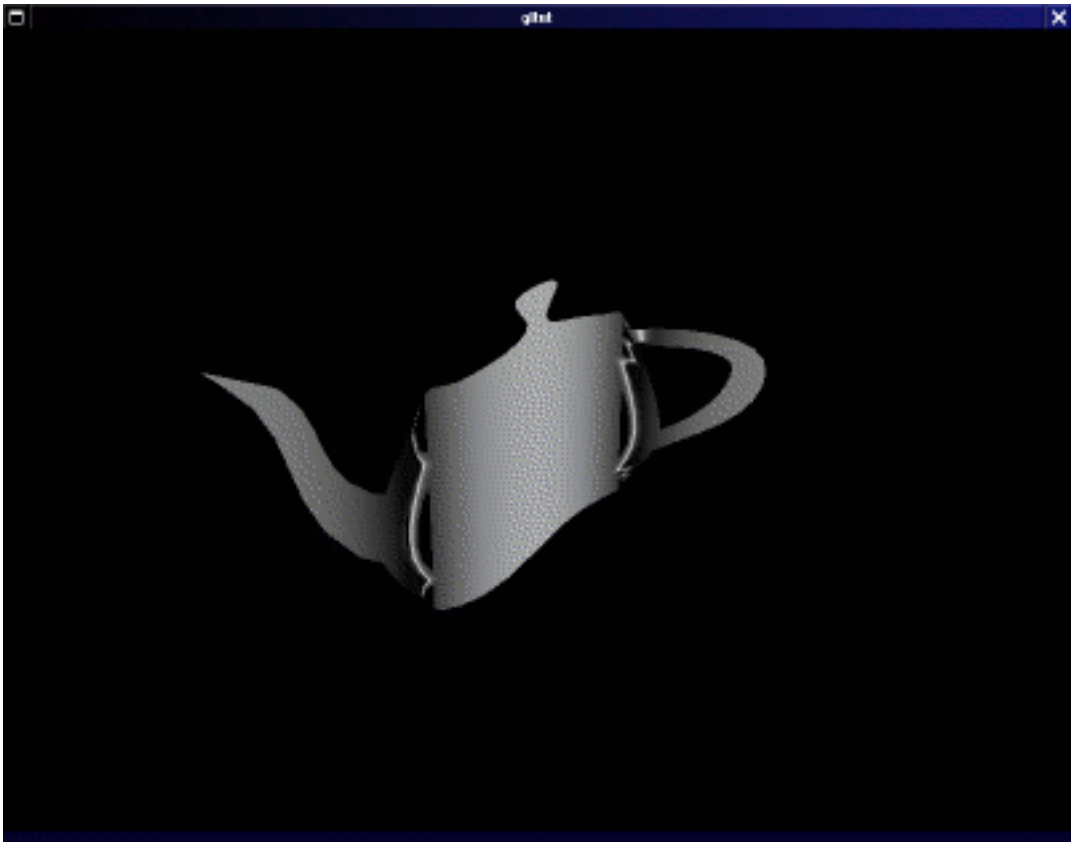

Abbildung 5.6: Die Flagge mit per Pixel Lighting

An dieser Stelle sollte nichts unbekanntes mehr hinzugekommen sein, wir berechnen weiterhin die Lichtrichtung, druecken dann den teapot flach und verformen ihn nach einer Sinusfunktion. Danach berechnen wir wieder unsere (falschen) Normalen, geben den Halbvektor fuer die Berechnung des spekulaeren Lichtanteiles nach Blinn sowie die diffusen und ambienten Farbanteile an den Fragment Shader weiter und transformieren die Vertexposition in die Projektionsebene - alles wie gehabt.

Am Fragment Shader selbst aendert sich logischerweise im Vergleich zum vorherigen Kapitel gar nichts. Das Ergebnis (./gltut shader/flag/flaglighting.avs shader/lighting/perpixel.afs) sieht zwar aufgrund unserer falschen Normalen nicht wirklich realistisch aus, aber dennoch koennen wir die Verformung der Ebene in Abb 5.6 nun deutlich erkennen.

5.8 wehende Flagge

Zum Abschluss wollen wir noch die in Kapitel 4.7 gesetzte, sich ueber die Zeit veraendernde, Variable nutzen um unsere Flagge auch zum wehen zu bringen:

5.8.1 GLSL waving flag Shader

Unsere Flagge soll innerhalb einer Sekunde einmal komplett durchschwingen, sprich wir wollen innerhalb dieser Zeitspanne die Sinusfunktion um eine komplette Periode verschieben. Hierzu muss lediglich der Vertex Shader aus 5.7.1 ein wenig angepasst werden:

```
const float PI = float(4.0f * atan(1.0f));

// uniform variable set by the application
uniform float timeElapsed;

varying vec4 diffuse, ambient;
varying vec3 normal, lightDir, halfVector;

void main()
{
    // normalize light direction
        lightDir = normalize(vec3(gl_LightSource[0].position));

    vec4 v = vec4(gl_Vertex);
    float offset = timeElapsed * 2.0f * PI;
    v.y = sin(v.x + offset);

    vec3 n = normalize(vec3(1/cos(v.x + offset), 1.0f, 0.0f));
    normal = normalize(gl_NormalMatrix * n);

    // normalize the halfVector to pass it to the fragment shader
    halfVector = normalize(gl_LightSource[0].halfVector.xyz);

    // compute diffuse and ambient colors
    diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
    ambient = gl_FrontMaterial.ambient * (gl_LightSource[0].ambient + gl_Light

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Listing 5.14: flagwave.avs

Nachdem wir uns π als Konstante definiert haben deklarieren wir die spaeter von der Applikation gesetzte uniform-Variable *timeElapsed*. Nun ist die einzige noch durchzufuehrende Aenderung die Berechnung der Verschiebung unserer Sinuskurve (*offset*) welche dann sowohl zum Setzen der neuen Position als auch zur Berechnung unserer - immernoch falscher - Normalen eingesetzt wird.

Diesen Vertex Shader kann man nun mit jedem unserer Fragment Shader zusammen benutzen, auch wenn bei den beleuchtenden Shadern gerade in der Bewegung doch sehr auffaelt dass die Normalen nicht korrekt gesetzt werden.

6 Vergleich OpenGL und Direct3D

Direct3D von Microsoft und OpenGL beanspruchen beide *die* API für dreidimensionale Echtzeitanwenungen zu sein. In diesem Kapitel sollen sie verglichen und einige Unterschiede zwischen ihnen herausgearbeitet werden. Während der Autor dieses Dokumentes lange Zeit mit OpenGL gearbeitet hat beschränken sich seine Erfahrungen mit Direct3D auf einige wenige Experimente, deswegen erhebt dieses Kapitel keinerlei Anspruch auf Vollständigkeit, Richtigkeit oder Freiheit von persönlicher Meinung.

Zuallererst sei gesagt dass OpenGL und Direct3D sich sehr ähnlich sind, und sich vor allem in ihrer Mächtigkeit nicht unterscheiden. Dies wird auch im Aufbau der Grafik-Pipelines deutlich. Abbildung 1.1 zeigt die Pipeline von OpenGL, Abbildung 6.1 zeigt die von Direct3D. Wie man sehr leicht erkennen kann unterscheiden sich die Konzepte kaum.

Einer der auffälligsten Unterschiede ist der dass Direct3D auf COM, dem Common Object Model - also C++ - basiert, während auf OpenGL im Grunde ein einfacher Zustandsautomat ist auf dessen Funktionalität ausschließlich über plain C Methoden zugegriffen wird. Im ersten Moment scheint Direct3D also die modernere API zu sein, allerdings wird OpenGL von vielen Entwicklern als sauberer empfunden, weil - im Gegensatz zu Direct3D - auf komplizierte Makros verzichtet wird und die Dokumentation immer noch kompletter erscheint. Allerdings hat Direct3D hier in den letzten Versionen deutlich aufgeholt. Das Interface von OpenGL ist prozedural, man führt Operationen aus die Vertices und Primitive an die Hardware schicken:

```
glBegin(GL_TRIANGLES);
    glVertex (0,0,0);
    glVertex (1,1,0);
    glVertex (2,0,0);
glEnd();
```

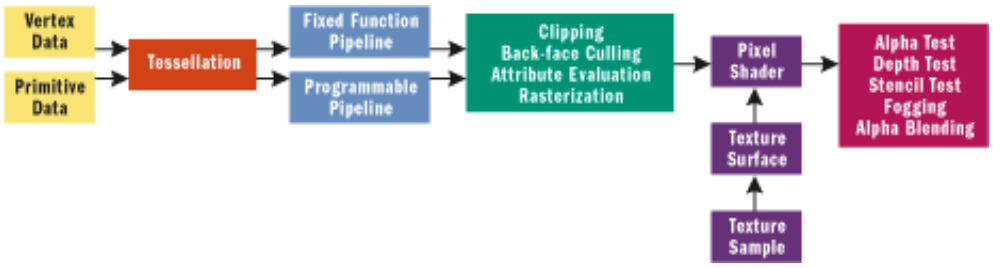

Abbildung 6.1: Die Direct3D Visualisierungspipeline (aus [14])

Listing 6.1: Ein Dreieck in OpenGL

Direct3D hingegen operiert auf Puffern und sendet diese komplett an die Hardware. Dies scheint ein großer Geschwindigkeitsvorteil zu sein, da es viel Overhead beim Funktionsaufruf spart, was bei einigen Zehntausend oder gar Hunderttausend Vertices in einer Szene einer heutigen Applikation durchaus einen Vorteil bringt. In Wirklichkeit bedeutet dies aber einen großen programmiertechnischen Aufwand für den Entwickler:

```
v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0;
v++;
v->x = 1; v->y = 1; v->z = 0;
v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

Listing 6.2: Ein Dreieck in Direct3D pseudo Code

Hier wird auch ersichtlich dass die Objektorientierung von Direct3D kaum Übersichtlichkeit bringt. Auch bietet OpenGL zur Reduktion des Overheads durch zu viele Funktionsaufrufe, den so genannten immediate mode. Anstatt einen Vertex einzeln durch viele Funktionsaufrufe zusammenzustellen

```
// send vertex normal
glNormal3f (0.5, 0.1, 0.3);
// send color
glColor4f (0.3, 0.3, 0.8, 1.0);
// send position
glVertex3i (2, 6, 8);
```

Listing 6.3: ein Vertex ohne immediate mode

und dies beliebig oft zu wiederholen kann man die Daten einzeln als Streams erstellen und dann gesammelt an die Hardware senden:

```
// set the pointer to the area of memory that contains the data
glInterleavedArrays (GL_T2F_C4F_N3F_V3F, 0, myPointer);

// draw vertices (as triangles) according to a stream of indicies
glDrawElements (GL_TRIANGLES, numberOfIndicies, GL_UNSIGNED_SHORT, indexPointe
```

Listing 6.4: ein Vertex mit immediate mode

Eine derartige Vereinfachung bietet Direct3D wiederum nicht. Dafür ist die DirectX API sehr viel umfangreicher und bietet unter anderem direkte Unterstützung zum Laden von Texturen und kompletten Modellen, was in OpenGL von weiteren APIs oder vom Entwickler selber geleistet werden muss.

Eine weitere häufig gestellte Frage bei der Auswahl der 3D API ist die nach der Performance. Diese lässt sich allerdings nicht ohne weiteres beantworten: beide Seiten behaupten dass ihre API die schnellere ist und liefern dafür auch Beispiele. Die Antwort liegt wohl eher darin dass die Performanceunterschiede heutzutage kaum mehr eine Rolle spielen, und dass ein Entwickler durch das Beherrschen einer der beiden APIs mehr Geschwindigkeit herausholen kann als durch marginale Unterschiede in speziellen Anwendungsfällen.

Der Grund warum Direct3D überhaupt die Chance hatte sich zu einer bedenkenswerten Alternative zu entwickeln liegt in der Historie begründet. OpenGL wurde zu den Anfangszeiten von beschleunigender Hardware nur sehr rudimentär von Windows unterstützt während Direct3D diese zwar oft unschön aber viel schneller umgesetzt hat. Auch wurden neue Features traditionell immer erst später in OpenGL implementiert, vor allem programmierbare Vertex- und Pixel-Shader.

Die Frage nach der besseren API ist sehr schwer zu beantworten. Soll die Applikation außerhalb von Windows (und XBox) eingesetzt werden führt kein Weg an OpenGL vorbei, da es sehr portabel ist und auf vielen Geräten - mit OpenGL ES ([2]) auch immer mehr im embedded-Bereich - lauffähig ist. Ist Portabilität allerdings kein Thema dann fällt die Entscheidung sehr schwer. Die Empfehlung des Autors in diesem Fall ist eine Münze zu werfen. Der Autor selbst hat sich für OpenGL entschieden - nicht nur weil er kein Windows einsetzt, sondern auch weil er die API für schöner hält, und weil er generell den Einsatz offener Standards bevorzugt.

Literaturverzeichnis

[1] http://OpenGL.org - OpenGL Homepage

[2] OpenGL ES - The Standard for Embedded Accelerated 3D Graphics (http://www.khronos.org/opengles/)

[3] OpenGL Programming Guide, Addison-Wesley. (http://fly.cc.fer.hr/ unreal/theredbook/)

[4] James F. Blinn, Models of Light Reflection for Computer Synthesized Pictures, Computer Graphics, Vol. 11, No. 2, July, 1977, 192-198. (http://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/blinn_model_fo

[5] http://mesa.org - MESA, eine freie OpenGL Implementation

[6] http://www.sgi.com/ - Silicon Graphics

[7] http://de.wikipedia.org/ - WikiPedia

[8] http://glew.sourceforge.net/ - GLew Homepage

[9] http://oss.sgi.com/projects/ogl-sample/registry/ - OpenGLÂ® Extension Registry

[10] http://nvidia.com/ - NVIDIA Homepage

[11] http://developer.nvidia.com/object/cg_toolkit.html - NVIDIA Cg Toolkit

[12] http://www.opengl.org/documentation/oglsl.html - GLSL Homepage

[13] http://www.lighthouse3d.com/opengl/glsl/ - Lighthouse 3D GLSL Tutorial

[14] DirectX 9.0 Introducing the New Managed Direct3D Graphics API in the .NET Framework (http://msdn.microsoft.com/msdnmag/issues/03/07/DirectX90/default.aspx)