# Kapitel 21

# OpenGL 3.1

## 21.1 Einordnung und Motivation

In der Rastergrafik, deren Grundlagen in den Kapiteln 13 bis 19 behandelt wurden, wird Materie als Geometrie modelliert und zusätzlich werden deren Oberflächeneigenschaften beschrieben. Dies geschieht, etwa im Falle der Auswertung des lokalen Beleuchtungsmodells, für jeden Punkt unabhängig vom Rest der Szene. Weil in das direkte Beleuchtungsmodell ausschließlich die Lichtquellen und Eigenschaften der Materie in diesem Punkt, nicht jedoch eventuell in der Verbindungsstrecke liegende Materie eingehen, existieren beispielsweise keine Schatten. Stattdessen muss der Programmierer eigene Routinen definieren, mit den Schatten explizit geworfen und nachträglich in die Szene eingefügt werden können. Weiterhin wurde, um nicht im direkten Einflussbereich einer Lichtquelle befindliche Materie überhaupt sichtbar erscheinen zu lassen, der Term des ambienten Lichts eingeführt. In Wirklichkeit erhalten diese Bereiche einer Szene durch indirektes Licht ihre Beleuchtung, welches ein oder mehrmals von anderer Materie in der Szene reflektiert wurde. Leuchtet ein Modell die Szene vollständig auf diese Weise aus und verzichtet auf einen ambienten Term, spricht man von Globaler Beleuchtung. Dementsprechend hat die beschriebene Vorgehensweise der Rastergrafik nahezu nichts mit den physikalischen Gesetzen, etwa der Lichtausbreitung, gemein.
Besonders deutlich zeigen sich die Unzulänglichkeiten des Modells anhand von Materialien, deren Eigenschaften nicht allein durch ihre Oberfläche beschrieben werden können. Dazu zählen Flüssigkeiten und Gase aber auch ein Beleuchtungsmodell der menschlichen Haut muss z.B. tiefer liegende Schichten berücksichtigen, um kein unnatürliches, „plastikartiges“, Aussehen zu erzeugen. Zwar existiert für die angesprochenen Probleme, gefördert u.a. durch zahlreiche Computerspiele, eine äußerst umfangreiche Sammlung von Algorithmen, welche oft als „Effekte“ bezeichnet werden. Jedoch simulieren diese „Hacks“ das Verhalten derartiger Materialien üblicherweise nicht physikalisch korrekt und liefern dementsprechend nur unter bestimmten Rahmenbedingungen glaubwürdige Ergebnisse. Beispielsweise kann ein 3D Oberflächenmodell einer Wolke, aus einer gewissen Entfernung betrachtet, einigermaßen glaubwürdig wirken. Nähert sich jedoch der Betrachter, werden die unrealistisch scharfen Kanten erkennbar. Ein denkbarer Trick, ohne physikalische Entsprechung, ist nun, diese Kanten nachträglich zu „verwischen“. Weiterhin genügt es gerade im Falle von Wolken nicht, lediglich ein lokales Beleuchtungsmodell für die Oberfläche auszuwerten. Stattdessen hängt die Beleuchtung in einem Punkt der Wolke u.a. von der Strecke ab, die Lichtstrahlen durch diese und andere Wolken auf dem Weg zu der Stelle durchdringen mussten, sowie Eigenschaften der Materie (Dichte, Konsistenz, ...) auf dem zurückgelegten Pfad. Zusätzlich werden die Teilchen auf diesem

Weg ebenfalls angeregt und strahlen ihrerseits Licht ab. Aufgrund der beschriebenen Funktionsweise der Rastergrafik kann eine allgemeingültige Nachbildung solcher optischer Erscheinungen beliebiger Materialien als sehr unwahrscheinlich eingestuft werden.

Im Gegensatz dazu steht Ray Tracing, in der Computergrafik ein Oberbegriff einer Klasse von Algorithmen, die Strahlen durch eine virtuelle Szene schicken, um ein Bild zu erzeugen. Diese Vorgehensweise ist weitaus näher an der realen Physik, weshalb viele der Algorithmen allgemeiner einsetzbar, vergleichsweise intuitiv verständlich und auch in der Computergrafik als Wissenschaft wesentlich weniger umstritten sind. Ein Beispiel eines einfachen Ray Tracers mit einem lokalen Beleuchtungsmodell folgt in Kapitel 23 dieses Skripts. Im Allgemeinen werden die Strahlen je nach Variante vom Betrachterstandpunkt und/oder von den einzelnen Lichtquellen ausgesandt. Die aufwändigeren Varianten können durchaus die Strahlen solange reflektieren und verfolgen, bis vom Betrachter ausgehende Srtahlen eine (nicht punktförmige) Lichtquelle oder von den Lichtquellen ausgesandte die Bildebene erreichen. Folglich ist Globale Beleuchtung umsetzbar und auch die anderen obengenannten Materialeigenschaften lassen sich, mit Varianten aus dem verwandten Bereich Volume Rendering, simulieren. Umgekehrt lässt sich nichts durch Rastergrafik darstellen, das nicht auch durch einen Raytracer erreichbar ist.
Aus diesen Gründen hat Rastergrafik im Bereich High Quality Graphics mittlerweile nahezu keinerlei Relevanz mehr. Ganz anders sieht es dagegen im Bereich Interactive Graphics aus. Denn dort ist nicht die bestmögliche Grafik, sondern die beste, noch in Echtzeit erzeugbare Darstellung, ausschlaggebend. Hier liegt die Rastergrafik derzeit weit in Führung, da der ihr zugrunde liegende Brute Force Ansatz hervorragend zur Ausführung auf moderner Grafikhardware geeignet ist. Nicht nur können Primitives, Vertices und Fragments völlig unabhängig voneinander verarbeitet werden, sondern zusätzlich werden identische Operationen auf großen Teilmengen davon ausgeführt, wovon die SIMD-artige Architektur moderner Grafikprozessoren (GPU) sehr stark profitiert. Zusätzlich können einige Teile der Berechnungen aufgrund ihrer Einfachheit sogar in fester Hardware umgesetzt werden, wodurch sich die Leistung weiter steigert. Dieser durch die Grafikhardware ermöglichte Performancevorteil von mehreren Größenordnungen im Vergleich mit einer Umsetzung auf CPUs sichert zumindest gegenwärtig die Daseinsberechtigung der Rastergrafik in Echtzeitanwendungen. Ansätze zur Beschleunigung der wesentlich flexibleren Ray Tracing Algorithmen sind bislang nicht von vergleichbarem Erfolg gekrönt.
Die bekanntesten APIs für Rastergrafik mit Hardwareunterstützung sind Microsofts DirectX unter Windows und die freie Bibliothek OpenGL. Ersteres zeichnet sich durch seine Leistungsfähigkeit, die breite Verfügbarkeit ausgereifter Implementierungen unter Windows sowie Hilfswerkzeuge für die Entwicklung aus. Das primäre Einsatzgebiet stellen traditionell Computerspiele dar. In letzter Zeit wird es jedoch auch vermehrt zur Forschung im Bereich der Computergrafik eingesetzt. OpenGL ist dagegen nicht nur unter Windows, sondern für die verschiedensten Betriebssysteme und Plattformen verfügbar. Aufgrund des daraus resultierenden breiten Anwendungsspektrums, etwa für Grafik im Web oder für Smartphones wie dem iPhone, liegt der Fokus dieser Veranstaltung auf OpenGL.
Gegenstand dieses Kapitels ist eine Einführung in OpenGL 3.1. Sofern nicht anders gekennzeichnet, beziehen sich alle Aussagen auf diese Version, auch wenn vereinfachend die Bezeichnungen OpenGL oder GL Verwendung finden. Vermittelt werden sollen dabei primär die zum Umsetzen einfacher Computergrafik Algorithmen dieser Veranstaltung mit Grafikhardwareunterstützung nötigen Kenntnisse.

## 21.2 Einleitung

OpenGL stellt eine Menge von Kommandos bereit, welche zum einen die Spezifikation von geometrischen Objekten in zwei oder drei Dimensionen ermöglichen und zum anderen solche die festlegen wie diese Objekte in einen Framebuffer gerendert werden. Die geometrischen Objekte setzen sich aus elementaren grafischen Grundformen (Primitives) zusammen. GL-Primitives sind ausschließlich: Punkt, Liniensegment und Dreieck. Nicht enthalten sind Routinen zum Beschreiben oder Verwalten komplexerer Objekte wie Kugel, Spline, Mesh, etc.. Deren Modellierung und Repräsentation durch die GL-Primitives bleibt der Applikation bzw. einer auf der GL aufsetzenden Hilfsbibliothek überlassen.

Die Zielsetzung von OpenGL ist, den Zugriff auf die Fähigkeiten der Grafikhardware auf der tiefsten Ebene zu gewährleisten, die noch hardwareunabhängig ist. Über ein einziges Interface soll der volle Funktionsumfang der GL auf verschiedenen Plattformen verfügbar sein, wobei ggf. fehlende Funktionalitäten in Software emuliert werden müssen. Zusätzlich vorhandene Funktionen können mithilfe sogenannter Extensions bereitgestellt werden, welche dann in einer Implementation berücksichtigt werden können aber nicht müssen. Auf diese Weise können zum einen neue Features sehr schnell integriert und zum anderen herstellerspezifische Eigenarten berücksichtigt werden. Aufgrund der Ausrichtung als procedurale API ist der Programmierer gezwungen, alle zum Rendern benötigten Schritte zu konfigurieren bzw. selbst im Detail festzulegen. Dies stellt einen fundamentalen Unterschied im Vergleich zu deskriptiven APIs (etwa Szenegraphen wie VRML, Kapitel 20) dar, bei denen der Programmierer das Aussehen einer Szene angibt und der Bibliothek die Details zu rendern überlässt. Dementsprechend benötigt der Programmierer sehr genaue Kenntnisse über die Graphics Pipeline, kann dafür jedoch eigene Algorithmen implementieren.

OpenGL ist, von der integrierten Shading Language GLSL abgesehen, eine API und keine Programmiersprache. Eine OpenGL Applikation ist demnach in einer anderen Sprache geschrieben und verwendet zusätzlich Befehle aus der OpenGL Bibliothek. Ein zentrales Merkmal ist die Auslegung zur Unterstützung verschiedener Plattformen (Smartphones, Spielkonsolen, PCs, Macs, etc.) und Betriebssysteme (Windows, MacOs, Linux, etc.). Weiterhin ist die API aus vielfältigen Sprachen wie etwa Fortran, C, C++, Java, Ruby, Php und sogar JavaScript heraus ansprechbar. In dieser Vorlesung wird OpenGL in Java Applets integriert, wobei die Bibliothek JOGL den Zugriff auf die nativen Funktionen ermöglicht. Auf diese Weise können die Beispielprogramme der Veranstaltung direkt in einem Webbrowser ausgeführt werden.

In der Spezifikation sind für eine Menge von ca. 250 Funktionen deren Verhalten, Wechselwirkungen mit anderen GL Funktionen sowie Auswirkungen auf den GL State nicht aber deren genaue Funktionsweise definiert. Eine Implementation dieser Spezifikation kann sowohl eine reine Software Bibliothek oder ein Treiber zusammen mit einer Grafikkarte sein. In dem Treiber ist dann festgelegt, welche Anteile einzelner Kommandos von der CPU oder der Grafikhardware übernommen werden. Wie groß der Anteil der Hardwarebeschleunigung ist, bleibt i.d.R. vor dem Programmierer verborgen. Da OpenGL auf sehr unterschiedlichen Zielplattformen implementiert werden können soll, legt die Spezifikation lediglich das ideale Verhalten fest. Teilweise ist eine bestimmte Abweichung von diesem Ideal erlaubt, sodass verschiedene OpenGL Implementationen bei der Ausführung identischer GL Operationen nicht zwingend für jeden Pixel übereinstimmende Ergebnisse liefern. Insgesamt legt

die Spezifikation die Funktionalität und das Ergebnis eines Rendervorgangs fest, allerdings ist dessen Qualität etwas und die zu erwartende Performance sehr stark implementationsabhängig.

## 21.3 Entwicklungsgeschichte

Die Entwicklung von OpenGL begann 1992 mit der Veröffentlichung der Spezifikation (Version 1.0) durch Silicon Graphics Inc. (SGI). Im selben Jahr wurde das OpenGL Architectural Review Board gegründet, dessen Mitglieder in den folgenden Jahren den Standard pflegen und erweitern sollten. Die Gründungsmitglieder der OpenGL ARB waren: SGI, Digital Equipment Corporation, IBM, Intel und Microsoft. Seit 2006 liegt die Kontrolle über den OpenGL Standard bei der Khronos Group, einem Konsortium zur Entwicklung offener Medienstandards auf verschiedenen Plattformen. Es besteht aus über hundert Mitgliedern, einige der Board Member sind: AMD/ATI, Apple, ARM, Imagination Technologies, Intel, Motorola, Nokia, nVidia, Samsung, Sony und Sun. Im Folgenden sollen einige wichtige Stationen von OpenGL hinsichtlich der Entwicklung eines Standards für hardwarebeschleunigte Grafik skizziert werden.

- **1.x**, ab 1992
  In frühen Versionen von OpenGL (1.X) bestand die Graphics Pipeline aus einer Folge fester, aber bis zu einem gewissen Grad konfigurierbarer Schritte (fixed Pipeline). So konnte beispielsweise die Beleuchtung mit Parametern für einzelne Bestandteile wie dem spekularen oder diffusen Anteil eingestellt werden. Nicht ohne weiteres möglich war dagegen die Implementierung eines eigenen Beleuchtungsmodells oder auch nur die Auswertung auf Fragment- statt Vertexebene. Das in diesem Skript als Phong Shading bezeichnete Beleuchtungsmodell war somit beispielsweise nicht verfügbar.

- **2.x**, ab 2004
  Um die stark eingeschränkte Flexibilität der festen Pipeline zu überwinden, wurde die programmierbare Pipeline eingeführt, bei der die starre Vertex- und Fragmentverarbeitung durch Vertex- und Fragment Shader ersetzt wurde. Dabei handelt es sich um praktisch beliebige auf der Grafikkarte ausführbare Programme, welche in die restliche weiterhin feste Pipeline integriert und anstelle der entsprechenden zuvor festen Schritte ausgeführt werden. Damit konnten zum einen sehr viel innovativere und oft auch effizientere Algorithmen implementiert werden und zum anderen war der Grundstein zum Nutzung der Grafikhardware für allgemeine Berechnungen (GPGPU) gelegt.

- **3.x**, ab 2008
  Über lange Zeit zeichnete sich OpenGL durch vollständige Abwärtskompatibilität aus. Die GL bestand in der Version 3.0 aus etwa 670 Kommandos, von denen viele redundant waren. Beispielsweise existierte die feste Pipeline weiterhin neben der Programmierbaren, mit der sich Erstere problemlos emulieren lässt. Ferner waren einzelne Funktionen sehr viel weniger effizient implementierbar als andere, die ein identisches Ergebnis lieferten. Infolgedessen wurden in der OpenGL Version 3.0 viele dieser redundanten Funktionen als deprecated ausgewiesen, um sie in zukünftigen Versionen entfernen zu können. Die im Mai 2009 veröffentlichte Version 3.1 entfernte die meisten der als deprecated markierten Funktionen aus dem OpenGL Kern und überführte sie in die neue ARB_compatibility Extension. Damit sind diese Funktionen nicht mehr zwingend Teil einer OpenGL Implementation und können, selbst wenn vorhanden, nicht

mit neuen Funktionen zusammen genutzt werden. Der Erhalt des mit OpenGL 1.x / 2.x kompatiblen Kontexts ist vor allem durch die CAD-Branche begründet, welche größtenteils weiterhin die bestehende Codebasis verwenden möchte. Ende 2009 folgte die OpenGL Version 3.2, welche unter anderem den bereits aus DirectX-10 bekannten Geometry Shader zur programmierbaren Verarbeitung von Primitives als Teil des Kerns einführte. Diese im Vergleich zu 3.0 deutlich schlankere API besteht aus ca. 250 Befehlen.

- **4.0**, ab 2010
  Im März 2010 wurden die Spezifikationen zu OpenGL 4.0 sowie zu GLSL Version 4.0 veröffentlicht. In OpenGL 4.0 hält die aus DirectX 11 bekannte Tessellation zur dynamischen Verfeinerung der Geometrie auf der GPU Einzug. Dazu werden zwei neue Shader, Tessellation Control und Tessellation Evaluation, sowie eine feste Pipeline Stage, der Primitive Generator, eingeführt. Weiterhin wurde die Möglichkeit zur Zusammenarbeit mit OpenCL, u.a. durch Einführung von 64 Bit Datentypen in GLSL, deutlich ausgebaut. Außerdem können mithilfe des Per-Sample-Fragment-Shaders flexiblere Anti Aliasing Techniken implementiert werden.

Bei der Entwicklung der API hat die Implementierbarkeit auf der zum jeweiligen Zeitpunkt aktuellen Grafikhardware immer eine zentrale Rolle gespielt. Anfangs wurden lediglich der Rasterizer und einige der anschließenden Operationen hardwarebeschleunigt (etwa Voodoo Graphics, 1996), später konnten zumindest teurere Grafikkarten auch die Transformation und Beleuchtung (T&L) der festen Pipeline übernehmen (etwa GeForce 256, 1999). Die später eingeführten Shaderprogramme wurden zunächst auf dedizierten Vertex- und Fragmentprozessoren ausgeführt (etwa GeForce 6800, 2004). Dies führte jedoch oft zu ungleichmäßiger Auslastung der Vertex- und Fragment- Prozessoren. Aus diesem Grund wurde das Unified Shader Modell eingeführt, bei dem Vertex-, Geometry- und Fragment Programme über praktisch identische Fähigkeiten verfügen und daher von der gleichen Hardware ausgeführt werden können. Verwendet wird dazu heute in der Regel an die bekannte SIMD-Architektur angelehnte Hardware (etwa GeForce 8800, 2006). Jene ist weiterhin sehr gut für parallele Computing Languages wie das 2008 ebenfalls von der Khronos Group veröffentlichte OpenCL geeignet. Dadurch sind prinzipiell beliebige Algorithmen auf dem Grafikprozessor (GPU) ausführbar und zudem ist eine direkte Zusammenarbeit mit OpenGL ist ebenfalls möglich. Zusammenfassend lässt sich ein Trend ausgehend von fester Hardware über dedizierte programmierbare Hardware hin zu Software, welche in massiv parallelen Umgebungen ausführbar ist, feststellen.

## 21.4 Spracheigenschaften und Syntax

OpenGL kann als Client Server Modell beschrieben werden. Eine Applikation (der Client) setzt OpenGL Befehle ab und die GL Implementation (der Server) führt diese aus. Typischerweise befinden sich Client und Server auf demselben Rechner, es besteht aber auch die Möglichkeit beide auf zwei Computer aufzuteilen und über Netzwerk miteinander kommunizieren zu lassen. OpenGL ist eine Zustandsmaschine, d.h. einmal gesetzte State Variablen, etwa die Hintergrundfarbe, behalten bis zu ihrem Widerruf ihre Gültigkeit. Auf diese Weise wird die Kommunikation zwischen Client und Server minimiert. Die Gesamtheit aller State Variablen zusammen mit den aktivierten Shadern legt genau fest, wie Primitives in den Framebuffer gerendert werden.

Neben der OpenGL Kern Bibliothek, deren Befehle mit **gl** beginnen, existiert eine Reihe weiterer Hilfsbibliotheken, von denen zwei sehr verbreitete hier kurz beschrieben werden:

- **OpenGL Utility Library (GLU)**
  Befehle dieser Bibliothek beginnen mit **glu** und stellen eine abstraktere Schicht über den Kommandos der OpenGL Library dar, um komplexere Operationen auszuführen. Darunter fallen unter anderem das vereinfachte Manipulieren von Projektionsmatrizen und das Modellieren von NURBS. Das OpenGL Utility Toolkit, dessen aktuelle Version 1.3 aus dem Jahre 1998 stammt, ist üblicherweise ebenfalls im Standard OpenGL Paket enthalten. Vorsicht: Auch wenn bislang keine der glu Befehle entfernt oder als deprecated markiert wurden, so basieren viele trotzdem auf Funktionen, welche ab der OpenGL Version 3.1 nicht mehr Teil des Kerns sind. Das Verhalten großer Teile dieser Bibliothek ist somit unklar.

- **OpenGL Utility Toolkit (GLUT)**
  OpenGL verfügt über keine Möglichkeit, die gerenderte Ausgabe an das jeweilige Fenstersystem anzubinden und kann keine Benutzereingaben verarbeiten. Diese Aufgaben kann für viele Plattformen das OpenGL Utility Toolkit übernehmen. Aufgrund dessen Einfachheit und eingeschränkter Funktionalität wird es primär in kleinen Demos zum Lernen von OpenGL eingesetzt. Das OpenGL Utility Toolkit gehört allerdings nicht zum Standard OpenGL Paket und wird seit einiger Zeit nicht mehr weiterentwickelt. Inzwischen gibt es unabhängige Hilfsbibliotheken mit gleicher Zielsetzung wie freeglut oder OpenGLUT. In dieser Veranstaltung wird GLUT nicht benötigt, da Java bzw. JOGL die Ein- und Ausgabe übernehmen.

Um die Portierung des OpenGL Codes zwischen verschiedenen Plattformen zu erleichtern, definiert OpenGL eigene Datentypen. Diese entsprechen gewöhnlichen C/C++ Datentypen, welche stattdessen verwendet werden können. Die folgende Tabelle führt die einzelnen Datentypen auf. Darin legt der ersten Spalte genannte Suffix als letzter Buchstabe eines Kommandos den erwarteten Datentyp an.

| Suffix | Datentyp | C-Korrespondenz | OpenGl Name |
|---|---|---|---|
| b | 8-bit integer | signed char | GLbyte |
| s | 16-bit integer | short | GLshort |
| i | 32-bit integer | int | GLint |
| f | 32-bit floating point | float | GLfloat, GLclampf |
| d | 64-bit floating point | double | GLdouble, GLclampd |
| ub | 8-bit unsigned integer | unsigned char | GLubyte, GLboolean |
| us | 16-bit unsigned integer | unsigned short | GLushort |
| ui | 32-bit unsigned integer | unsigned int | GLuint, GLenum, GLbitfield |

Die Mehrheit der OpenGL Funktionen folgt einer Namenskonvention, deren erster Teil die Bibliothek angibt, aus der dieser Befehl stammt, gefolgt vom Namen der GL Wurzel Funktion. OpenGL kennt kein Überladen der Funktionen. Daher werden bei Kommandos mit gleicher Funktionalität aber anderen Parametern Befehlssuffixe hinzugefügt, welche diese Argumente festlegen. Die Form der OpenGL Befehle ist:

```
<Library prefix><Root command>[arg count][arg type][vektoriel]
```

Dabei kennzeichnen eckige Klammern optionale Suffixe. Ein Beispiel:

```
glUniform4fv(...)
```

Damit wird der Wurzelbefehl **Uniform** aus der Bibliothek **gl** aufgerufen und ihm 4 Float Werte in vektorieller Form (etwa als C-Pointer) übergeben. Eine weitere Spezialisierung vieler GL Funktionen ist durch die übergebenen GL Konstanten möglich. Beispielsweise hat der Befehl **glEnable(**`int cap`) in Abhängigkeit des übergebenen Parameters `cap` sehr unterschiedliche Auswirkungen auf den GL State. Ein Auszug:

| | |
|---|---|
| GL_DEPTH_TEST | Aktiviert eine Vorschrift, gemäß der nur das vorderste oder das hinterste Fragment Einfluss auf den Framebuffer hat |
| GL_CULL_FACE | Aktiviert die drehsinnabhängige Entfernung der Vorder- und/oder Rückseite der Dreiecke (Culling) |
| GL_PRIMITIVE_RESTART | Aktiviert eine spezielle Indizierungstechnik für Primitives |
| GL_VERTEX_PROGRAM_POINT_SIZE | Ermöglicht dem Vertex Shader das Verändern der Größe von Point Primitives |

Weitere Beispiele für OpenGL Funktionen:

```
// Setzen der Hintergrundfarbe
void glClearColor(float red, float green, float blue, float alpha);

// Setzen von Position und Größe eines rechteckigen Bereichs im Fenster,
// in den die Ausgabe gerendert wird
void glViewport(int x, int y, int width, int height);
```

## 21.5 JOGL und Codebeispiele

Wie bereits erwähnt, werden in dieser Veranstaltung native OpenGL Befehle aus Java Applikationen über die Java OpenGL Anbindung JOGL ausgeführt. JOGL ist verfügbar unter: `http://kenai.com/projects/jogl/pages/Home`. Zusammenfassend werden an dieser Stelle die für die Codebeispiele relevanten Unterschiede und Gemeinsamkeiten zu einer C-Anbindung aufgezählt:

- Namen und Bedeutung der OpenGL Funktionen und Konstanten sind identisch
- Ein- und Ausgabe erfolgt mithilfe von Java- und JOGL-Befehlen
- Da in Java keine Pointer existieren, werden in JOGL stattdessen vor allem von `java.nio.Buffer` abgeleitete Klassen für die Parameterübergabe verwendet
- Anstelle der GL Datentypen werden die entsprechenden aus Java eingesetzt
- Das JOGL Interface `GL3` stellt eine zum OpenGL 3.1+ Kontext kompatible Schnittstelle bereit und enthält alle Konstanten
- Der Zugriff auf OpenGL Befehle wird über die Methoden einer Instanz einer `GL3` implementierenden Klasse ermöglicht. Dieses Java Objekt hat in den folgenden Codebeispielen den Namen `gl`

Beispiel: Aus dem OpenGL Code einer C-Anbindung

```
glEnable(GL_DEPTH_TEST);
```

wird hier in JOGL:

```
gl.glEnable(GL3.GL_DEPTH_TEST);
```

Dementsprechend einfach ist die Portierung des OpenGL Codes aus einer C-Applikation und umgekehrt. Die in den folgenden Kapiteln enthaltenen Codebeispiele sind Auszüge eines größeren, ausführbaren OpenGL 3.1 Programms, welches auf der Webseite zu dieser Veranstaltung erreichbar ist. Das Programm kann eine mit einer Farbtextur versehene Geometrie rendern, wie in Abbildung 21.1 zu sehen. Da es sich um ein durchgehendes Beispiel handelt, werden gelegentlich Variablen vorheriger

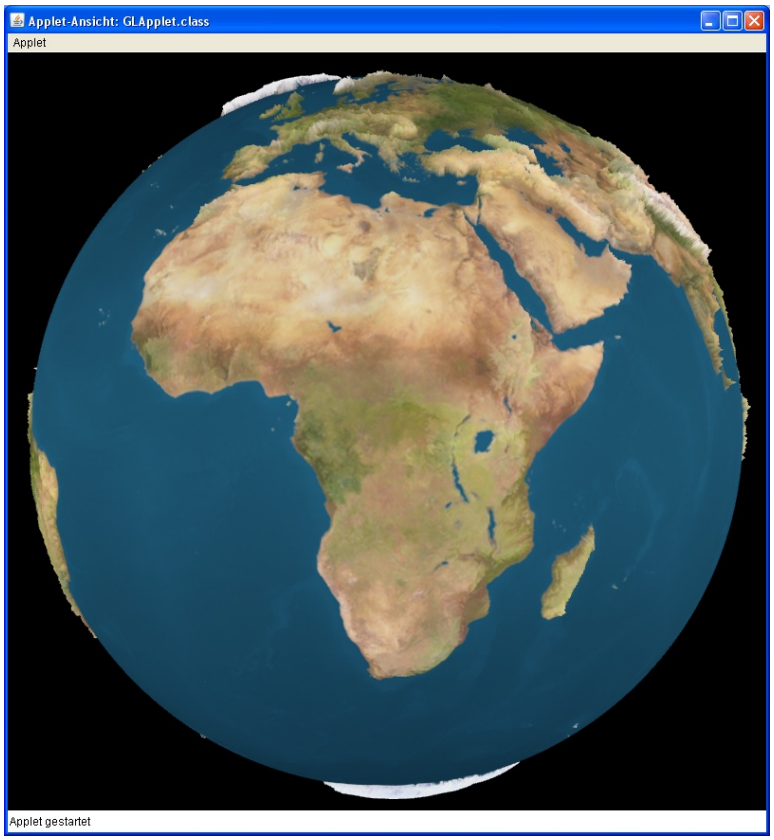



Abbildung 21.1: Darstellung einer Geometrie mit Farbtextur

Listings wiederverwendet. Es handelt sich bei den Beispielen um OpenGL (via JOGL) und GLSL Code. Dabei werden die jeweiligen Schlüsselbegriffe hervorgehoben und zur besseren Unterscheidung wird der OpenGL Code wie

```
gl.glBufferData(target, size, data, usage);
```

durch die Hintergrundfarbe deutlich von GLSL Code wie

```
uniform sampler2D colorTexture;
```

abgegrenzt.

## 21.6 Arten von Informationen

### 21.6.1 Vertices

Bei einem Vertex handelt es sich um einen mathematischen Punkt im Raum, der oft Eckpunkt einer geometrischen Figur ist. Neben der geometrischen Information, hinterlegt in der Position, enthält er häufig weitere Eigenschaften der Oberfläche in diesem Punkt. Verbreitete Beispiele sind Farbinformationen, Normalen und Texturkoordinaten aber auch andere Eigenschaften wie Materialdichte, Beschleunigung und Geschwindigkeit können zu den Vertex Attributen zählen. Vertexdaten können in OpenGL in Form von serverseitigen Datenstrukturen, den Buffer Objects, hinterlegt werden. Dabei handelt es sich um eine eindimensionale Folge der Attribute aller zu rendernder Vertices.
Sei `bufferNames` ein Java Objekt des Typs `IntBuffer`, so lautet der Code zum Erzeugen zweier (leerer) Buffer Objects mit **glGenBuffers**:

```
// Anzahl der zu erzeugenden Buffer Objects
int n = 2;

// Java Buffer, in dem die Indices der Buffer Objects hinterlegt werden
Java.nio.IntBuffer buffers = bufferNames;

gl.glGenBuffers(n, buffers);
```

Anschließend kann das zu einer Id gehörende Buffer Object zur Verwendung für Per Vertex Daten mit der Funktion **glBindBuffer** initialisiert werden:

```
int coordBuffer = bufferNames.get(0);
int buffer = coordBuffer;

// Gibt an, ob Vertex- oder Indexdaten enthalten sind. Hier: Vertexdaten
int target = GL3.GL_ARRAY_BUFFER;

gl.glBindBuffer(target, buffer);
```

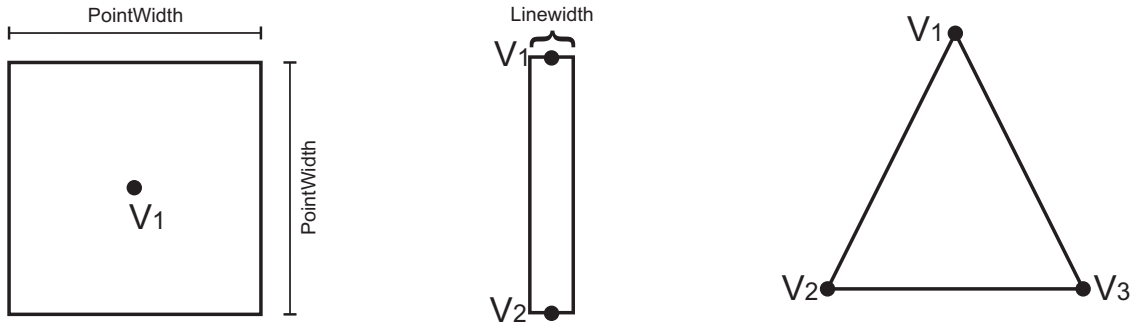
Dabei gibt die Konstante `GL_ARRAY_BUFFER` an, dass das Buffer Object Vertex Daten enthalten soll. Neben der Initialisierung, welche nur beim ersten Aufruf mit einer Vertex Buffer Id erfolgt, aktiviert der Befehl **glBindBuffer** das Buffer Object. Nachfolgende Befehle beziehen sich dann auf dieses Objekt. Anschließend kann der benötigte Speicher angefordert und optional mit Werten initialisiert werden. Vertex Daten, wie das Java `FloatBuffer` Objekt `vertexCoords`, können durch den Befehl **glBufferData** hinzugefügt werden:

```
// Benötigter Speicher in Byte
int size = vertexCoords.capacity * 4;

// Erwartete Art der Benutzung.
// Passende Wahl führt eventuell zu besserer Performance
int usage =  GL3.GL_STATIC_DRAW;

// Die Daten. Hier: FloatBuffer mit den Positionen aller Vertices.
Buffer data = vertexCoords;

target = GL3.GL_ARRAY_BUFFER;
gl.glBufferData(target, size, data, usage);
```

Hinweis: Mit allen anderen Vertex Daten, wie etwa Texturkoordinaten, wird analog verfahren.

### 21.6.2 Primitives

Die beschriebenen Vertices haben keinerlei Ausdehnung und können demnach nicht angezeigt werden. In OpenGL existieren drei Arten von elementaren geometrischen Grundformen, genannt Primitives, welche Gruppen aus ein bis drei Vertices eine räumliche Ausdehnung zuweisen können. Die erste sind Punkte, die über einen Vertex mit einer zusätzlichen Breite definiert werden. Die Breite wird in Pixeln angegeben und veranlasst den Rasterizer, für eine Breite · Breite große Fläche mit dem Vertex im Mittelpunkt Fragments zu erzeugen (siehe Abb. 21.2(a)).




(a) Ein Punkt, definiert über Mittelpunkt und Seitenlänge

(b) Eine Linie, definiert über Endpunkte und Breite

(c) Ein Dreieck, definiert über drei Eckpunkte

Abbildung 21.2: Primitives in OpenGL

Weiterhin gibt es mit Linien eindimensionale Primitives, die über zwei durch eine Gerade verbundene Vertices und einer zusätzlichen Linienbreite in Pixeln definiert werden (siehe Abb. 21.2(b)). Am häufigsten Verwendung finden jedoch zweidimensionale Primitives, in OpenGL ausschließlich Dreiecke. Sie werden durch drei Vertices repräsentiert und der Rasterizer erzeugt typischerweise Fragments für die gesamte Dreiecksfläche (siehe Abb. 21.2(c)).
Mit Ausnahme der Punkte benötigen alle Primitives zusätzlich zu den geometrischen auch topologische Informationen. Diese wird zum einen durch eine Sequenz von Indices definiert, wodurch die Reihenfolge, in der die Vertices zu Primitives zusammenzusetzen sind, festgelegt ist. Zum anderen können zum Indizieren von Linien und Dreiecken verschiedene Schemata eingesetzt werden. Einige sind für sechs Vertices und das Index Array {0, 1, 2, 3, 4, 5} in Abbildung 21.3 zu sehen.

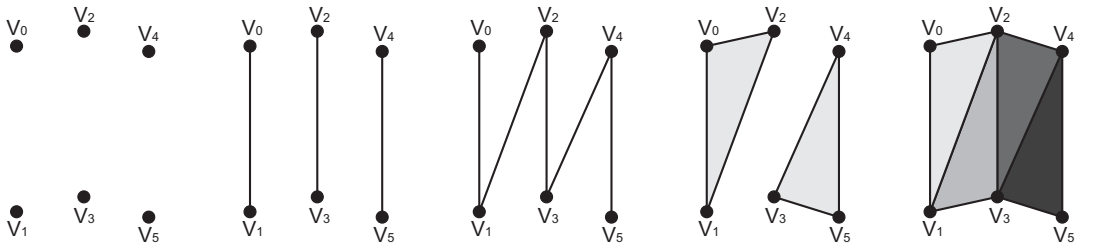



Abbildung 21.3: Primitive Indizierung

Im Einzelnen werden für diese Indexfolge die folgenden Primitives (in Klammern) erzeugt:

- `GL_POINTS`
  Ein Punkt besteht aus genau einem Vertex: (0), (1), (2), (3), (4), (5)
- `GL_LINES`
  Je zwei Vertices definieren eine Linie: (0, 1), (2, 3), (4, 5)
- `GL_LINE_STRIP`
  Eine Linienfolge wird definiert durch mindestens zwei Vertices, die der Reihe nach mit Linien verbunden werden: (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)
- `GL_TRIANGLES`
  Je drei Vertices definieren ein Dreieck: (0, 1, 2), (3, 4, 5)
- `GL_TRIANGLE_STRIP`
  Eine Dreieckfolge wird definiert durch mindestens drei Vertices, die der Reihe nach zu Dreiecken verbunden werden: (0, 1, 2), (2, 1, 3), (3, 1, 4), (4, 1, 5)

Weil z. B. im Inneren geschlossener Flächen jeder Vertex Teil etlicher Primitives ist, können diese einfach durch Mehrfachindizierung verschiedenen Primitives zugewiesen werden. Veranschaulicht wird das Vorgehen anhand des in Abbildung 21.4 dargestellten Dreiecknetzes. Dort wird mithilfe zweier Index Arrays eine zusammenhängende Fläche definiert, ohne einen Vertex doppelt setzen zu müssen. Insbesondere ist der Vertex V4 Teil von sechs Dreiecken.

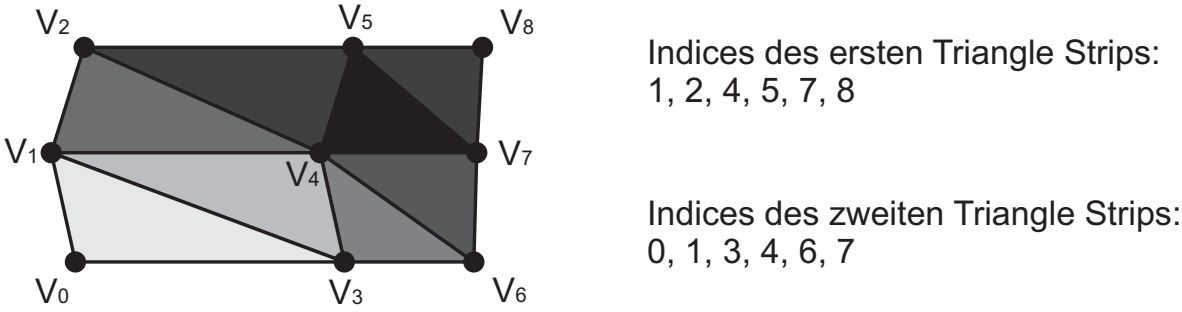



Abbildung 21.4: Ein Mesh bestehend aus zwei Triangle Strips

Die Indices werden ebenfalls in einem Buffer Object hinterlegt, dessen Erzeugung ähnlich wie bei den Vertex Daten abläuft. Sei `indices` ein Java `IntBuffer` mit der Folge der Indices. Dann lautet der Code zum Erzeugen des Index Buffer Objects:

```
int indexBuffer = bufferNames.get(1);
buffer = indexBuffer;

// Gibt an, ob Vertex- oder Indexdaten enthalten sind. Hier: Indexdaten
target = GL3.GL_ELEMENT_ARRAY_BUFFER;

// Benötigter Speicher in Byte
size = indices.capacity * 4;

data = indices;
gl.glBindBuffer(target, buffer);
gl.glBufferData(target, size, data, usage);
```

Dabei enthält das Java IntBuffer Objekt `indices` die Indexfolge und die Konstante GL_ELEMENT_ARRAY_BUFFER gibt an, dass das Buffer Object Indices enthält. Ist genau ein Index Array und ein oder mehrere Vertex Attribute Arrays aktiviert und letztere an Variablen des aktiven Shaders angebunden (siehe Abschnitt 21.10), können die Daten mithilfe des Befehls **glDrawElements** durch die Graphics Processing Pipeline verarbeitet werden:

```
// Anzahl der zu rendernden Elemente, hier: alle
int count = indices.capacity();

// Verweis auf das erste Element
int pointer = 0;

// Art des Primitives
int mode = GL3.GL_TRIANGLE_STRIP;

int type = GL3.GL_UNSIGNED_INT; // Datentyp

// Auslösen des Renderns
gl.glDrawElements(mode, count, type, pointer);
```

### 21.6.3 Globale Daten

In diese Kategorie gehören für die gesamte Geometrie eines Renderaufrufs identische Daten. Dazu gehören oft Projektionsmatrizen und Lichtquellen. Weiterhin zählen Texturen zu dieser Kategorie. Anders als die allgemeine Definition einer Textur, nämlich einer Vorschrift zum Versehen einer Oberfläche mit zusätzlichen Details, handelt es sich in OpenGL bei einer Textur schlicht um eine ein- bis dreidimensionale diskrete Datenstruktur zusammen mit einer Reihe von Funktionen zum i.d.R. kontinuierlichen Zugriff darauf. Die einzelnen Elemente einer Textur wiederum sind ein- bis vierdimensional und werden als Texel bezeichnet. Diese Datenstruktur kann beliebige numerische Informationen enthalten und ist aus allen Shadern heraus lesbar. So kann eine OpenGL Textur, insbesondere im Vertex Shader, auch für andere Aufgaben als zum Einfärben der Oberfläche verwendet werden. Nachfolgend wird ein Weg zum Erstellen einer gewöhnlichen 2D-Textur aus Rasterdaten und ohne Mipmapping beschrieben. Dafür muss zunächst mit dem Befehl **glGenTextures** ein leeres Texture Object erstellt werden:

```
// Anzahl der zu erzeugenden Texture Objects
int n = 1;

// Java Buffer, in dem die Indices der Texture Objects hinterlegt werden
IntBuffer textures = texNames;

gl.glGenTextures(n, textures);
```

Außerdem muss mit **glActiveTexture** eine der Textureinheiten, deren Anzahl implementationsabhängig ist, aktiviert werden:

```
// ID einer, hier der ersten, Textureinheit
int texUnit = GL3.GL_TEXTURE0;

gl.glActiveTexture(texUnit);
```

Nachfolgende Texturbefehle beziehen sich dann auf diese Textureinheit. Anmerkung: Gleichzeitig zu verwendende Texturen müssen verschiedenen Textureinheiten zugewiesen werden.

Anschließend kann das eingangs erzeugte Texture Object als 2D Textur initialisiert, aktiviert und mit **glBindTexture** an die zuvor aktivierte Textureinheit angebunden werden:

```
// Es soll eine 2D-Textur erzeugt werden
int target = GL3.GL_TEXTURE_2D;

// ID unserer Textur
textureName = texNames.get(0);

gl.glBindTexture(target, textureName);
```

Dem nun aktiven zweidimensionalen Texture Object können mithilfe der korrespondierenden Funktion **glTexImage2D** Daten, hier enthalten im Java `ByteBuffer image`, zusammen mit einer Beschreibung derselben übergeben werden:

```
target = GL3.GL_TEXTURE_2D;

// Mipmapping Level; Hier: deaktiviert, sonst 1...Max Level
int level = 0;

// Internes format der Texel
int internalFormat = GL3.GL_RGB;

// Breite der Textur in Texeln
int width = 503,

// Höhe der Textur
int height = 123,

// Zusätzlicher Rand; muss ab OpenGL 3.1 0 sein
int border = 0,

// Format der übergebenen Daten
int format = GL3.GL_RGB;

// Datentyp der übergebenen Daten
int type = GL3.GL_UNSIGNED_BYTE;

// Die Daten
Buffer pixels = image;

gl.glTexImage2D(target, level, internalFormat, width, height, border, format, type, pixels);
```

Soll Mipmapping eingesetzt werden, ist dieser Befehl für jede Mipmap zu wiederholen. Weitere Optionen, wie Interpolationsart, Interpretationsvorschrift für die Texturkoordinaten, etc. können mithilfe der Funktion **glTexParameter*** eingestellt werden.

### 21.6.4 Fragments

Die bisherigen Daten werden an die GL übergeben und können durch diese verarbeitet werden. Im Gegensatz dazu entstehen Fragments erst im Durchlauf der Graphics Pipeline aus der Geometrie. Der Rasterizer erzeugt aus den in den Bildraum projizierten Primitives für jeden überlappten Pixel eine Datenstruktur, genannt Fragment. Diese erhält neben der diskreten Pixelkoordinate eine Tiefeninformation und die für den Ort des Pixels interpolierten Daten der Vertices des Primitives. Im weiteren Verlauf der Graphics Pipeline (siehe Abschnitt 21.8) kann dem Fragment eine Farbe zugewiesen werden, welche eventuell Einfluss auf die Einfärbung des korrespondierenden Pixels hat. Im beschriebenen Spezialfall kann man sich das Fragment anschaulich als eine zum aktuellen Primitive gehörige Vorstufe zu diesem Pixel vorstellen.

## 21.7 Grober Ablauf eines Anwendungsbeispiels

In diesem Abschnitt soll anhand eines einfachen Beispiels ein Überblick über den gesamten Prozess von der Modellierung einer Szene bis zu einem gerenderten Bild beschrieben werden.

- **Modellierungssoftware (optional)**
  Komplexere Szenen, etwa in aktuellen Computerspielen, werden üblicherweise von Künstlern mithilfe spezieller Modellierungs- und Zeichenwerkzeuge erzeugt. Diese Programme verbergen die zugrundeliegende Computergrafik und performancekritische Details vor dem üblicherweise nicht in dieser Richtung ausgebildeten Künstler. Die auf diese Weise erzeugten 3D-Modelle, Texturen etc. sind i.d.R. bei weitem zu detailliert, als das sie direkt in der Praxis eingesetzt werden könnten. Deswegen werden die Modelle anschließend auf Detailstufen heruntergerechnet, welche den Anforderungen der einzelnen Zielplattformen entsprechen. Weiterhin können so auch für eine einzelne Plattform verschiedene Detailstufen (LOD) erzeugt werden, um beispielsweise für weiter vom Betrachter entfernte Figuren weniger Rechenleistung aufwenden zu müssen. Da dieser Teil nicht für das Laufzeitverhalten der Zielapplikation relevant ist, können hier sehr aufwändige Verfahren eingesetzt werden. Zuletzt werden die erzeugten Modelle in einem für die Zielapplikation lesbaren Format exportiert.

- **Applikation**
  Zunächst müssen die zuvor modellierten statischen Daten wie etwa Landschaften, 3D-Figuren-Modelle sowie zugehörige Texturen geladen und initial in der Szene angeordnet werden. Zur Laufzeit werden bewegliche Objekte ihre Position und i.d.R. auch geometrische Eigenschaften ändern (etwa Beine etc.). Die Aufstellung der zugehörigen Matrizen für diese dynamische Modellierung ist Aufgabe der Applikation. Weiterhin muss das Frustum samt zugehöriger Matrizen für die Betrachtungstransformation aufgestellt werden. Dieser Teil der Applikation, auch als 3D-Engine bezeichnet, testet weiterhin größere Objekte auf Sichtbarkeit, um diese gar nicht erst an den Renderer zu übergeben, legt die aktuelle LOD Stufe der Objekte fest und sortiert zumindest transparente Flächen vor. Außerdem sollte der Render State, etwa die gerade aktiven Texturen oder Shader, möglichst selten geändert werden müssen. Andernfalls müssten ständig andere Daten in den Grafikspeicher geladen und die Processing Pipeline reorganisiert werden. Eine in der Praxis schwierige Aufgabe für die 3D-Engine ist demnach, die Szene in einer Weise zu verwalten, sodass eine sinnvolle Balance aus räumlicher und Render State Kohärenz erreicht wird. Der in diesem Abschnitt beschriebene Teil kann direkt in der jeweiligen Programmiersprache (Java, C, C++, etc.) implementiert sein, alternativ existiert gerade für Standardaufgaben eine große Anzahl von Hilfsbibliotheken (u.a. Szenegraphen).

- **OpenGL**
  Am Ende des vorherigen Schrittes steht fest, welche Teile der Szene in welcher Reihenfolge und mit welchen Einstellungen an den Renderer zu übergeben sind. Diese Teile werden unabhängig voneinander in der zuvor festgelegten Reihenfolge gerendert. An dieser Stelle beginnt erst der Aufgabenbereich von OpenGL, der im Wesentlichen für jeden gleichzeitig zu rendernden Teil der Szene aus zwei Punkten besteht. Zum einen muss der feste Teil der Graphics Processing Pipeline konfiguriert sowie Shader aktiviert werden und zum anderen müssen die für diesen Teil benötigten Daten (Szenengeometrie, Texturen, etc. ), aber auch Lichtquellen und Transformationsmatrizen von der Applikation an die GL Implementation übergeben werden (vergl. Abschnitt 21.10).

- **Graphics Processing Pipeline**
  Anschließend durchläuft die Geometrie die Graphics Processing Pipeline (Siehe Abschnitt 21.8). Diese übernimmt üblicherweise u.a. die Projektion der Geometrie, wertet die Beleuchtung aus, färbt die gerasterten Oberflächen ein und schreibt das Ergebnis in den Framebuffer. Dieser Prozess läuft idealerweise vollständig auf der GPU ab und wird durch ein OpenGL Kommando lediglich angestoßen. Dementsprechend findet sich der zugehörige Code, mit Ausnahme der in einer Shading Language, wie GLSL, verfassten Shader, nicht in der Applikation wieder.

Abschließend bleibt anzumerken, dass diese Vorgehensweise zwar nicht untypisch, jedoch keinesfalls zwingend ist. Gerade Multipass Renderer können deutlich davon abweichen und auch sonst kann es etwa sinnvoll sein, einzelne Aufgaben aus der Applikation in die Shader zu verlagern und umgekehrt. Weiterhin können Renderer implementiert werden, welche mit klassischer Rastergrafik praktisch gar nichts gemein haben.

## 21.8 Graphics Processing Pipeline

Aus Programmierersicht ist OpenGL eine feste Sequenz von Operationen, welche die Ausgangsdaten in ein Bild überführen kann. Diese wird als Graphics Processing Pipeline oder Graphics Pipeline bezeichnet und ist (in vereinfachter Form) in Abb. 21.5 zu erkennen.

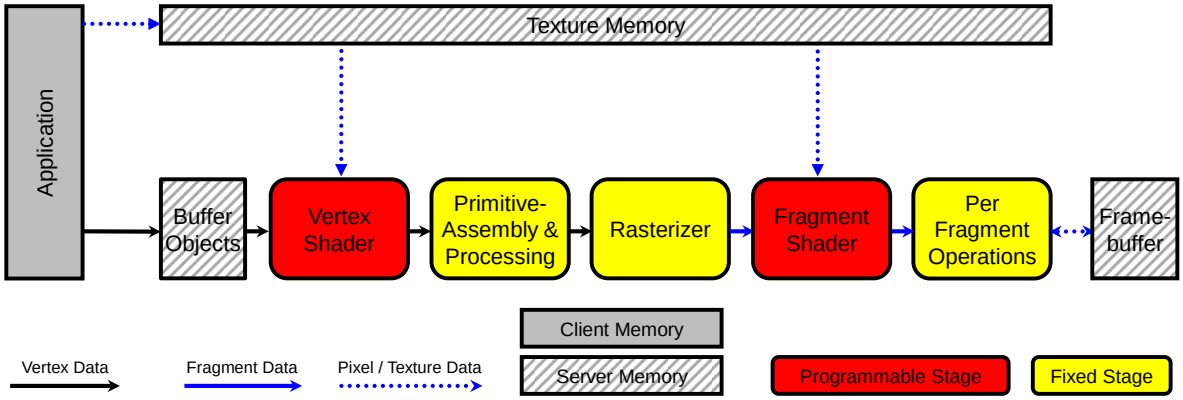



Abbildung 21.5: Vereinfachte Darstellung der Graphics Processing Pipeline in OpenGL

Tatsächlich muss eine GL Implementation lediglich mit dieser Pipeline identische Ergebnisse liefern, sich jedoch nicht im Detail an die Folge der einzelnen Schritte halten. Die wichtigsten Abschnitte sind:

- **Vertex Shader**
  Der Vertex Shader ist ein nahezu beliebiges Programm, welches Per Vertex Daten verarbeitet. Typische Aufgaben, die von ihm übernommen werden können, sind die Transformation von Vertices und Normalen oder auch die Beleuchtung einzelner Vertices (in diesem Skript als Gouraud Shading bezeichnet). Eine Instanz des Vertex Shaders verarbeitet immer nur die zu einem Vertex gehörenden Daten und hat keinerlei Lese- oder Schreibzugriff auf Daten anderer Vertices. Folglich kann eine beliebige Anzahl von Vertices ohne Synchronisationsaufwand parallel verarbeitet werden.

| | Eingang | Ausgang |
|---|---|---|
| Art | Vertices | Vertices |
| Verhältnis pro Instanz | 1 | 1 |

- **Primitive Assembly**
  Nachdem bislang die zu den einzelnen Vertices gehörigen Informationen völlig unabhängig voneinander verarbeitet wurden, werden in diesem Abschnitt mithilfe der topologischen Informationen die jeweiligen Primitives wiederhergestellt und die zugehörigen Daten gesammelt. Dementsprechend verlassen diese Stufe entweder Punkte, Linien oder Dreiecke samt der mit ihnen assoziierten Daten.

| | Eingang | Ausgang |
|---|---|---|
| Art | Vertices | Primitives |
| Verhältnis pro Instanz | 1, 2, 3 | 1 |

- **Primitive Processing**
  In diesem Abschnitt finden mit Clipping und Culling diejenigen Operationen statt, für die Kenntnisse über das gesamte Primitive nötig sind. Anschließend werden die Koordinaten in einem als Perspective Divide bezeichneten Schritt durch die homogene Koordinate geteilt.

| | Eingang | Ausgang |
|---|---|---|
| Art | Primitives | Primitives |
| Verhältnis pro Instanz | 1 | $\geq 0$ |

- **Rasterizer**
  Aufgabe des Rasterizers ist die Überführung von Primitives in Fragments. Dazu wird für jedes von einem Primitive überlappte Pixel ein Fragment erzeugt, welches die 2d-Komponente des korrespondierenden Pixels sowie eine zusätzliche Tiefeninformation erhält. Weiterhin werden alle Daten der Eckpunkte des Primitives für diese Koordinate interpoliert und ebenfalls im Fragment gespeichert.

| | Eingang | Ausgang |
|---|---|---|
| Art | Primitives | Fragments |
| Verhältnis pro Instanz | 1 | 0 ... Fensterauflösung (i.d.R.) |

- **Fragment Shader**
  Der Fragment Shader ist ein nahezu beliebiges Programm, welches die durch den Rasterizer erzeugten Fragments verarbeitet. Dabei werden auf Basis der Per Fragment Daten und globaler Daten wie etwa Texturen Berechnungen durchgeführt, um beispielsweise die Farbe des Fragments festzulegen. Oft finden hier die Texturierung der Oberflächen und die Auswertung eines Beleuchtungsmodells statt (in diesem Skript: Phong Shading). Ebenso wie im Vertex Shader gibt es keinerlei Möglichkeit Ergebnisse zwischen den Fragments auszutauschen, um die Parallelität der Berechnungen nicht einzuschränken.

| | Eingang | Ausgang |
|---|---|---|
| Art | Fragments | Fragments |
| Verhältnis pro Instanz | 1 | 0, 1 |

- **Per Fragment Operations**
  Die Per Fragment Operations regeln im Wesentlichen ob, und auf welche Weise die bereits verarbeiteten Fragments Einfluss auf den Framebuffer haben. Gerade in komplexen Szenen können Pixel von mehreren Primitives überlappt werden, sodass viele Fragments mit einem Pixel korrespondieren. Bei undurchsichtigen Oberflächen kann mithilfe des Tiefentests festgelegt werden, dass sich immer das zuvorderst liegende Fragment durchsetzt und einen ggf. bereits im Framebuffer hinterlegten Wert überschreibt. Alternativ besteht die Möglichkeit, den Wert des jeweils aktuellen Fragments und den bereits im Framebuffer eingetragenen als Eingaben für eine Blending Funktion zu verwenden und das Resultat wieder in den Framebuffer zu schreiben. Auf diese Weise lassen sich beispielsweise Transparenzeffekte realisieren.

| | Eingang | Ausgang |
|---|---|---|
| Art | Fragments | Pixel |
| Verhältnis pro Instanz | 1 | 0, 1 |
| Verhältnis pro Pixel | 0 ... N | 1 |

## 21.9 OpenGL Shading Language

Der Begriff Shader wurde erstmals im Jahre 1984 von Cook in dessen Paper „Shade Trees“ eingeführt und meint ein Programm zur Beschreibung von Oberflächeneigenschaften. Im High-Quality-Rendering-Bereich existieren schon lange entsprechende Sprachen, wie die verbreitete und an C angelehnte Renderman Shading Language, an welcher sich auch aktuelle Shading Languages orientieren. Im Bereich der hardwarebeschleunigten 3D APIs, wie OpenGL oder DirectX, versteht man unter einem Shader dagegen ein in einer Shading Language (GLSL, HLSL, Cg, ...) geschriebenes Programm, welches auf einer GPU ausführbar ist. Sie waren nach Einführung von nVidias GeForce 3 im Jahre 2001 erstmalig einsetzbar. Es mussten allerdings zu ihrer Verwendung in OpenGL die entsprechende Extensions bemüht werden. Shader sind, wie der Vertex- und Fragment Shader (siehe 21.9), oft Teil der Graphics Processing Pipeline, werden jedoch nicht zwingend zum Berechnen der Oberflächeneigenschaften oder gar „Schattieren“ eingesetzt. Praktisch nichts mehr mit Cooks Definition gemein hat der Compute Shader, welcher in DirectX für allgemeine Berechnungen auf der GPU zuständig ist. Aufgrund des zu großen Umfangs der Thematik beschränkt sich dieses Skript auf eine exemplarische Behandlung von GLSL mit dem Fokus auf Vertex- und Fragment Shader.

Die Shading Language GLSL (auch: glSlang) ist seit der GL Version 2.0 (2004) Teil des OpenGL Kerns und ihre aktuelle Version ist 1.5 (2009). GLSL ist ebenso wie OpenGL Plattform und Betriebssystemunabhängig. Der GLSL Compiler ist Teil des Display Drivers und übersetzt die Shader erst zur Laufzeit, sodass optimierter Code für die jeweilige Hardware Architektur erzeugt werden kann. GLSL ist eine Hochsprache mit einer an C angelehnten Syntax. Es existieren allerdings eine Reihe Unterschiede, von denen einige auszugsweise im Folgenden erläutert werden. In diesem Skript kann lediglich ein kurzer und sehr unvollständiger Einblick in GLSL gegeben werden.

In GLSL existieren verschiedene Qualifier, welche Variablen vorangestellt sind und vor allem die Schnittstelle der Shader mit der restlichen Graphics Pipeline festlegen. Dies sind:

**in** Kennzeichnet im Shader lesbare Variablen, deren Wert für jeden Vertex bzw. jedes Fragment verschieden ist. Im Vertex Shader erhalten In-Variablen ihren Wert aus der Applikation und im Fragment Shader handelt es sich dabei um die interpolierten Vertexdaten nach dem Rastern der Primitives.

**uniform** Kennzeichnet im Shader lesbare globale Variablen, deren Wert für alle Vertices und Fragments während eines Durchlaufs der Graphics Pipeline konstant ist. Uniform-Variablen können ausschließlich durch die Applikation geschrieben werden und sind dann aus allen Shadern heraus in gleicher Weise lesbar.

**out** Kennzeichnet im Shader schreibbare Variablen, welchen die Ergebnisse der Berechnungen des jeweiligen Shaders zugewiesen werden. Diese Daten werden anschließend durch die restliche Graphics Pipeline verarbeitet.

**const** Kennzeichnet gewöhnliche Konstanten.

In GLSL existieren folgende skalare Datentypen: `float`, `int`, `uint` und `bool`. Beispiele:

```
float f = 2.5;
bool b = true;
```

Zusätzlich existieren vektorielle Datentypen, welche 2 bis 4 Komponenten, bestehend aus obengenannten Skalaren, besitzen:

**vec2, vec3, vec4** Vektordatentypen, bestehend aus 2, 3 und 4 Floats.

**ivec2, ivec3, ivec4** Vektordatentypen, bestehend aus 2, 3 und 4 Integers.

**uvec2, uvec3, uvec4** Vektordatentypen, bestehend aus 2, 3 und 4 Unsigned Integers.

**bvec2, bvec3, bvec4** Vektordatentypen, bestehend aus 2, 3 und 4 Booleans.

Auf die einzelnen Komponenten kann wahlweise über die Namensschemata x,y,z,w oder r,g,b,a oder s,t,p,q zugegriffen werden:

**x, r, s** Zugriff auf die erste Komponente eines Vektors

**y, g, t** Zugriff auf die zweite Komponente eines Vektors

**z, b, p** Zugriff auf die dritte Komponente eines Vektors

**w, a, q** Zugriff auf die vierte Komponente eines Vektors

Die Operatoren +, -, * und / sind für Vektortypen komponentenweise definiert. Einige Beispiele:

```
vec2 vector1 = vec2(1.0, 2.0);
vec2 vector2 = vec2(0.0, 1.0);
vec2 compWiseMul = vector1 * vector2;
```

Dementsprechend liefert `compWiseMul.x` den Wert 0.0 und `compWiseMul.t` den Wert 2.0. Zusätzlich existieren Matrizen aus Fließkommazahlen bis zu einer Größe von 4x4.

Sampler enthalten die zum Zugriff auf eine Textur benötigte Information und werden einem Shader von der Applikation übergeben. Beispielsweise ermöglicht sampler2D mithilfe der Funktion **texture** den Zugriff auf eine zweidimensionale Textur:

```
uniform sampler2D colorTexture;
vec2 texCoords = vec2(0.0, 0.0);
vec3 color = texture(colorTexture, texCoords);
```

Dabei stellt *texCoords* einen zweidimensionalen Index zum Zugriff auf eine Stelle in der Textur `colorTexture` dar und die Funktion **texture** liefert den entsprechenden Wert. In diesem Fall handelt es sich dabei um einen 3-Vektor der als RGB Tripel interpretiert wird.
Die Flusskontrolle ist ebenfalls an C angelehnt. So ist der Eintrittspunkt in einen Shader die Funktion main. Es existieren Schleifen (for, while, do-while) sowie die Schlüsselwörter break und continue. Verzweigung ist mit if und if-else möglich. Darüber hinaus besteht im Fragment Shader die Möglichkeit mit dem Schlüsselwort discard einen Beitrag des aktuellen Fragments zum Framebuffer zu verhindern. Funktionsaufrufe sind im Wesentlichen an C++ angelehnt, allerdings besteht keine Möglichkeit diese rekursiv aufzurufen. Anders als OpenGL kennt GLSL Überladung, sodass im obigen Beispiel nicht für jeden Texturtyp eine eigene Funktion bereitgestellt werden muss. Weiterhin enthält GLSL eine recht umfangreiche Bibliothek von mathematischen Standartfunktionen für Skalare, Vektoren und Matrizen. Dazu zählen Exponential-, Trigonometrische- und Interpolationsfunktionen, Kreuz- und Skalarprodukt, sowie Funktionen zum transponieren von Matrizen, etc.

### 21.9.1 Vertex Shader

Der Vertex Shader ist ein Programm, welches die Daten einzelner Vertices unabhängig voneinander verarbeitet. Vertexdaten haben für jeden Vertex einen anderen Wert, typische Beispiele dafür sind etwa die Position, die Normale und Texturkoordinaten. Abbildung 21.6 zeigt die in einen Vertex Shader eingehenden und die durch ihn schreibbaren Daten. Erstere sind im Wesentlichen:

- **User-defined In Variablen**
  Durch den Programmierer zu definierende Variablen, welche Per Vertex Daten enthalten. Auch gängige Vertexeigenschaften wie Position oder Normale müssen selbst definiert werden, da sie nicht zwingend zu einem Vertex gehören. Diese Variablen erhalten ihren Wert durch die Applikation und können im Vertex Shader gelesen werden.

- **User-defined Uniform Variablen**
  Durch den Programmierer zu definierende Variablen, welche globale Daten, etwa Projektionsmatrizen und Lichtquellen, enthalten. Sie erhalten ihren Wert durch die Applikation und können im Vertex- und Fragment Shader gelesen werden.

- **Build-in Uniform Variablen**
  Durch GLSL definierte Variablen, welche globale Daten enthalten.

- **Texture Maps**
  Texturen sind eine spezielle Form der User-defined Uniform Variablen, die ebenfalls globale Daten enthalten. Für einen Vertex wird jedoch i.d.R. nicht alles, sondern beispielsweise über eine Texturkoordinate nur ein bestimmter Wert ausgelesen.

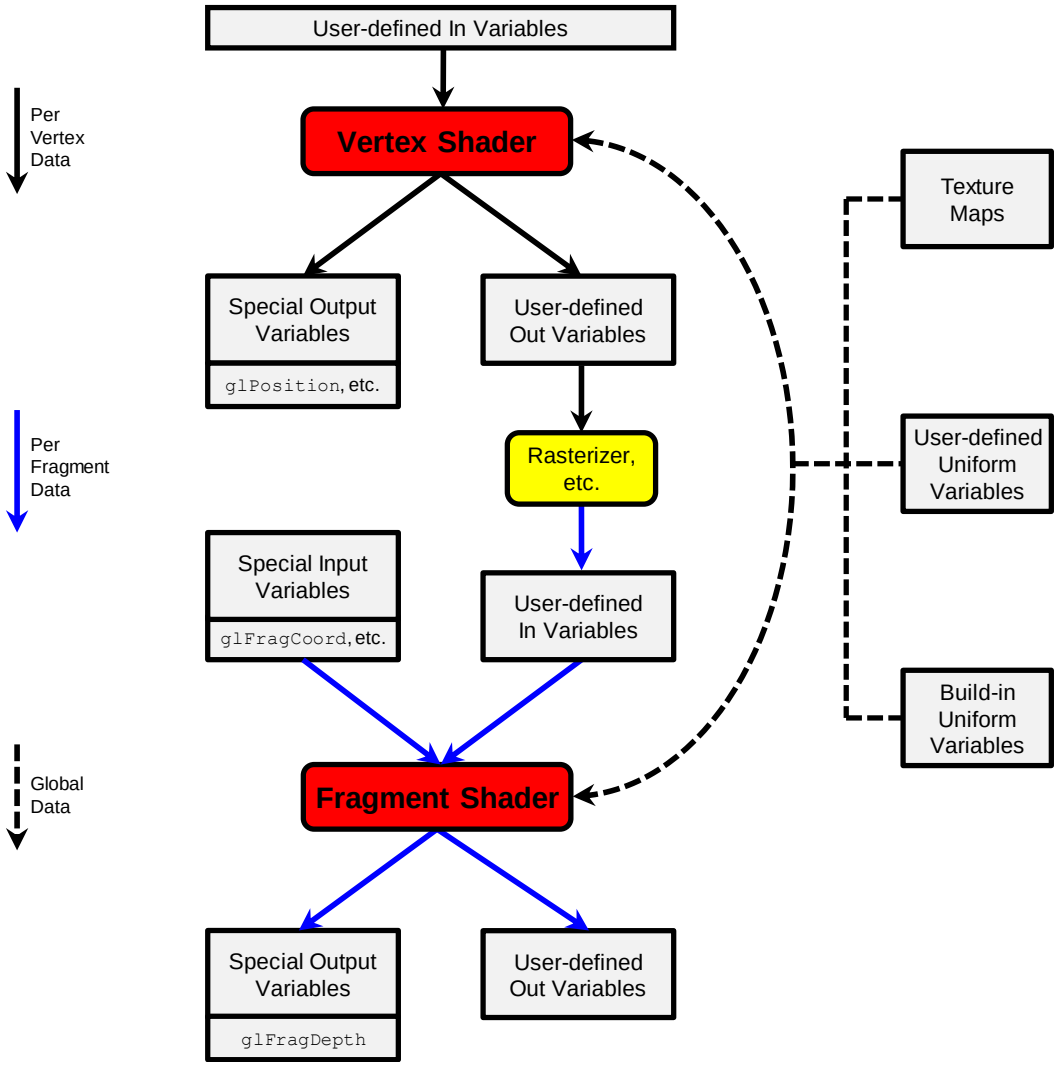



Abbildung 21.6: Datenstrom durch Vertex- und Fragment Shader

Basierend auf den obengenannten Daten kann der Vertex Shader praktisch beliebige Berechnungen durchführen, deren Ergebnisse mithilfe zweier Arten von schreibbaren Variablentypen ausgeben werden können:

- **User-defined Out Variablen**
  Durch den Programmierer definierte Variablen, mit deren Hilfe der Vertex Shader praktisch beliebige Informationen zur weiteren Verarbeitung durch den Fragment Shader ausgeben kann. Denkbare Beispiele sind: Transformierte Normalen oder Texturkoordinaten, Farbwerte, Krümmung, Geschwindigkeit, Gewicht, Temperatur, Druck, etc.

- **Special Output Variablen**
  Durch OpenGL definierte Variablen, deren Werte ggf. für Teile der auf den Vertex Shader folgenden Festen Pipeline benötigt werden. Ein Beispiel ist **gl_Position**, welche mit der in Clipping Koordinaten transformierten Vertexposition belegt werden muss, um eine sinnvolle Ausführung des Rasterizers zu ermöglichen.

Das folgende Listing zeigt einen sehr einfachen Vertex Shader zum Projizieren von Geometrie:

```
// GLSL Version 1.40
#version 140

// User-defined Uniform Variable des Typs 4x4-Matrix zum Transformieren
// der Vertexposition aus Object Coordinates in Clip Coordinates.
uniform mat4 mvpMatrix;

// User-defined In Variable des Typs 4-Vektor.
// Enthält Koordinaten des jeweiligen Vertex.
in vec4 myPosition;

// Texturkoordinaten
in vec2 myTexCoords;

// User-defined Out Variable für Texturkoordinaten
out vec2 texCoords;

void main() {

 // Texturkoordinaten werden (hier) einfach weitergereicht.
 texCoords = myTexCoords;

 // Transformieren der Koordinaten durch
 // Multiplikation mit der Transformationsmatrix
 vec4 vPositionCC =  mvpMatrix * myPosition;

 // Weise das Ergebnis der Special Output Variable gl_Position zu.
 gl_Position = vPositionCC;

}
```

### 21.9.2 Fragment Shader

Der Fragment Shader ist ein Programm, welches die Daten einzelner Fragments unabhängig voneinander verarbeitet. Abbildung 21.6 zeigt die in einen Fragment Shader eingehenden und die durch ihn schreibbaren Daten. Erstere sind im Wesentlichen:

- **User-defined In Variables**
  Hierbei handelt es sich um die durch den Rasterizer für das jeweilige Fragment interpolierten User-defined Out Variablen des Vertex Shaders. Ihr Name muss mit diesen übereinstimmen.

- **Special Input Variables**
  Durch GLSL definierte Variablen, welche Per Fragment Daten enthalten. Ein Beispiel ist **gl_FragCoord**, die Position des Fragments in Window Coordinates.

Zusätzlich existieren auch im Fragment Shader die Uniform Variablen und Texture Maps, welche sich identisch zu denen des Vertex Shaders verhalten. Basierend auf den obengenannten Daten kann der Fragment Shader praktisch beliebige Berechnungen durchführen, deren Ergebnisse mithilfe zweier Arten von schreibbaren Variablentypen ausgegeben werden können:

- **User-defined Out Variables**
  Der Programmierer kann für einen Fragment Shader eine Reihe von Ausgabevariablen definieren, welche in verschiedene Buffer geschrieben werden. Oft ist das Ergebnis der Berechnungen

des Fragment Shaders eine Farbe, welche eventuell Einfluss auf die Einfärbung des mit dem jeweiligen Fragment korrespondierenden Pixels hat.

- **Special Output Variables**
  Hierbei handelt es sich ausschließlich um die durch GLSL definierte Variable **gl_FragDepth**, welche die Tiefe des jeweiligen Fragments angibt. Wird sie nicht geschrieben, findet implizit die z-Komponente von **gl_FragCoord** Verwendung.

Das folgende Listing zeigt einen sehr einfachen Fragment Shader, der mit dem obigen Vertex Shader kombinierbar ist und eine Oberfläche gemäß einer Farbtextur einfärbt:

```
// GLSL Version 1.40
#version 140

// Handle zum Zugriff auf eine 2D-Textur, welche Farbinformationen
// enthält (hier: RGBA)
uniform sampler2D colors;

// Texturkoordinaten des jeweiligen Fragments
in vec2 texCoords;

// User-defined Out Variable zur Ausgabe der Farbe des Fragments
out vec4 myFragColor;

void main() {

 // Hole den RGBA Farbwert für die Texturkoordinaten des Fragments
 // aus der Textur und gib diesen als Ergebnis aus
 myFragColor = texture(colors, texCoords);

}
```

## 21.10 OpenGL Shading Language API

Unter der Bezeichnung OpenGL Shading Language API versteht man diejenige Teilmenge der OpenGL Funktionen, die das Erzeugen, Übersetzen, Linken und Aktivieren von Shadern übernehmen sowie diese mit Daten versorgen. Es folgt ein Überblick über einige zum Erzeugen und Benutzen von Shadern benötigte GL Funktionen.

### Erzeugen eines Shaderprogramms

Zunächst muss ein Shader Programm, bestehend aus mindestens einem Vertex Shader sowie optional einem Fragment Shader, erstellt werden. Im folgenden Codebeispiel werden je ein leeres Vertex- und ein Fragment Shader Object mit **glCreateShader** erzeugt:

```
int myVs = gl.glCreateShader(GL3.GL_VERTEX_SHADER);
int myFs = gl.glCreateShader(GL3.GL_FRAGMENT_SHADER);
```

Dabei legt die übergebene Konstante die Art des zu erzeugenden Shaders fest und die Rückgabe ist ein Index zum weiteren Zugriff auf das serverseitig vorliegende Shader Objekt.

Im Anschluss daran wird den Shader Objekten der Sourcecode in Form eines Stringarrays mit der Funktion **glShaderSource** übergeben:

```
// Anzahl der Strings im Array
int count = 1;

// Längen der Strings im Array. In Java nicht unbedingt benötigt.
IntBuffer length = null;

// Id des Shaders
int shader = myVS;

// Der zugehörige Quellcode
String[] string =  {" Der VS-Quellcode (S. 250)... "};

// Erzeugen des Vertex Shaders
gl.glShaderSource(myVs, count, vsSource, length);

// Analog für den Fragment Shader...
shader = myFS;
string = {" Der FS-Quellcode (S. 251)... "};
gl.glShaderSource(shader, count, string, length);
```

In Java kann der Quellcode als einzelner String übergeben werden, dementsprechend genügt ein Array der Länge eins und weitere Angaben über die Längen der Arrays entfallen. Anschließend müssen die Shader mit **glCompileShader** übersetzt werden:

```
gl.glCompileShader(myVs);
gl.glCompileShader(myFs);
```

Weiterhin muss mit **glCreateProgram** ein (leeres) Program Object erzeugt werden:

```
int myShaderProgram = gl.glCreateProgram();
```

Diesem Program Object können dann mittels **glAttachShader** die zuvor kompilierten Shader Objects hinzugefügt werden:

```
int program = myShaderProgram;

// Füge den Vertex Shader ...
shader = myVS;
gl.glAttachShader(program, shader);

// und den Fragment Shader zum Shader Program hinzu
shader = myFS;
gl.glAttachShader(program, shader);
```

Zuletzt muss das Program Object noch mit **glLinkProgramm** gelinkt werden:

```
gl.glLinkProgram(myShaderProgram);
```

Falls beim Kompilieren der Shader Objects oder beim Linken des Shader Programs kein Fehler aufgetreten ist, kann das nun ausführbare Programm mit **glUseProgram** als Teil des aktuellen GL State gesetzt werden:

```
gl.glUseProgram(myShaderProgram);
```

### Steuern des Datenflusses

Für das zuvor erzeugte Shader Object muss weiterhin der Datenfluss in dieses hinein und aus ihm heraus geregelt werden. Einige Varianten werden nachfolgend beschrieben.

### In Variablen

Enthält ein Vertex Shader In-Variablen, welche die Per Vertex Attribute repräsentieren, muss die Applikation diesen Daten zuweisen. Hierzu müssen zunächst, wie in Abschnitt 21.6.1 beschrieben, die zugehörigen Buffer Objects erzeugt und das jeweils anzubindende aktiviert werden. Im Fall obiger Vertex Koordinaten somit:

```
gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
```

Ferner muss die Adresse der korrespondierenden Variable des Shaders abgefragt werden. Sei, wie im Vertex Shader aus Abschnitt 21.9.1, eine User-defined In-Variable `myPosition` deklariert als:

```
in vec4 myPosition;
```

Dann kann zunächst ihr Index mit **glGetAttribLocation** erfragt werden:

```
// Id des Program Objects
int program = myShaderProgram;

// Name der Variable des Vertex Shaders des Program Objects
String name = "myPosition";

int location = gl.glGetAttribLocation(program, name);
```

Anschließend kann das mit dieser Position zu assoziierende aktive Array mit Vertex Attributen aktiviert werden über:

```
int index = location;
gl.glEnableVertexAttribArray(index);
```

Zuletzt muss mit dem Befehl **glVertexAttribPointer** spezifiziert werden, wie die Daten im Buffer hinterlegt sind:

```
// Anzahl der Komponenten eines Vertex Attributs
int size = 3;

int type = GL3.GL_FLOAT; // Datentyp

// Normalisieren der Daten erforderlich?
boolean normalized = false;

//Abstand in Bytes zwischen konsekutiven Vertex Attributen
int stride = 0;

// Verweis auf das erste Element
int pointer = 0;

gl.glVertexAttribPointer(location, size, type, normalized, stride, pointer);
```

### Uniform variablen

Enthält ein Shader Program Uniform Variablen, kann deren Position mit **glGetUniformLocation** erfragt und anschließend ihr Wert mit **glUniform*** gesetzt werden. Enthalte ein oder mehrere Shader eines Program Objects `myShaderProgram` eine uniform Variable `mvpMatrix`, welche folgendermaßen deklariert ist:

```
uniform mat4 mvpMatrix ;
```

so kann deren Adresse bestimmt werden mit:

```
int location = glGetUniformLocation(myShaderProgram, "mvpMatrix");
```

Anschließend können der Variable Daten, enthalten im Java `FloatBuffer mvpMatrix`, zugewiesen werden. Dies geschieht im vorliegenden Spezialfall einer 4x4 Matrix mittels **glUniformMatrix4fv**:

```
// Nur eine Matrix und kein Array
int count = 1;

// GL_TRUE: Werte in row major order angegeben
// GL_FALSE: Werte in column major order angegeben
boolean transpose = GL_FALSE;

// Die von der Applikation aufgestellte Transformationsmatrix
FloatBuffer value = mvpMatrix;

glUniformMatrix4fv(location, count, transpose, value)
```

Texture Maps sind spezielle Uniform Variablen, die zunächst Erzeugt und Aktiviert werden müssen (siehe Abschnitt 21.6.3). Der weitere Verlauf ist dem zuvor Beschriebenen recht ähnlich und beginnt ebenfalls mit dem Abfragen der Adresse der im Fragment Shader aus Abschnitt 21.9.2 über

```
uniform sampler2D colors;
```

deklarierten Textur, hier mit allerdings mit **glGetUniformLocation**:

```
location = glGetUniformLocation(myShaderProgram, "colors");
```

Dieser wird nun der Textur zugeordneten Textureinheit übergeben. In diesem Fall demnach:

```
// Der zu übergebende Wert. Hier die Id der Textureinheit
int x = texUnit;

gl.glUniform1i(location, x);
```

### Out Variablen

Weiterhin können Out Variablen des Fragment Shaders an die einzelnen Render Buffer angebunden werden. Da lediglich der Buffer mit dem Index 0 angezeigt wird, müssen Farbwerte, welche nach dem aktuellen Rendervorgang sichtbar sein sollen, daran angebunden werden. Sei im Fragment Shader eine Out Variable `myFragColor` des Program Objekts `myShaderProgram` deklariert als:

```
out vec4 myFragColor;
```

dann kann sie vor dem Linken des Program Objects durch den Befehl **glBindFragDataLocation** an den anzeigbaren Buffer gebunden werden:

```
gl.glBindFragDataLocation(myShaderProgram, 0, "myFragColor");
```

## 21.11 Weitere Beispiele

In den vorherigen Abschnitten wurden die Konzepte einer GL Applikation beschrieben und an zentralen Stellen anhand von Codebeispielen erläutert. Nicht für OpenGL im Speziellen relevanter, reiner Java Applikationscode, etwa für Ein- und Ausgaben, die Routinen zum Aufstellen des Frustums sowie der zugehörigen Projektionsmatrizen und das Modellieren der Szene können dem auf der Webseite der Veranstaltung verfügbaren Quellcode entnommen werden. Weiterhin werden externe Daten benötigt, wie die Farbtextur, welche ebenfalls unter der angegebenen URL verfügbar ist. Das auf der nächsten Doppelseite folgende minimale Codebeispiel ist dagegen vollständig und benötigt keine externen Daten:

```
import javax.swing.JApplet; import java.nio.*; import com.sun.opengl.util.*;
import javax.media.opengl.*; import javax.media.opengl.awt.GLCanvas;

public class GLApplet extends JApplet implements GLEventListener {

    private GL3 gl; // Objekt zum Absetzen der GL-Befehle
    private int programObject, colorBuffer, coordBuffer;
    private GLCanvas canvas;

    public void init() {
        canvas = new GLCanvas(new GLCapabilities(GLProfile.get(GLProfile.GL3)));
        canvas.addGLEventListener(this);
        add(canvas);
    }

    // Erzeugt ein Shaderobjekt, siehe Abschnitt 21.10
    private int LoadShader(int type, String[] shaderSrc) {
        int shader = gl.glCreateShader(type);
        gl.glShaderSource(shader, 1, shaderSrc, null);
        gl.glCompileShader(shader);
        return shader;
    }

    // JOGL Methode, wird durch FPSAnimator aufgerufen
    public void display(GLAutoDrawable drawable) {
        gl.glClear(GL3.GL_COLOR_BUFFER_BIT); // Framebuffer loeschen

        // Per-Vertex Daten Re-Aktivieren und anbinden. (Abschnitt 21.10)
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
        int location = gl.glGetAttribLocation(programObject, "vPosition");
        gl.glVertexAttribPointer(location, 3, GL3.GL_FLOAT, false, 0, 0);
        gl.glEnableVertexAttribArray(location);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, colorBuffer);
        location = gl.glGetAttribLocation(programObject , "vColor");
        gl.glVertexAttribPointer(location, 3, GL3.GL_FLOAT, false, 0, 0);
        gl.glEnableVertexAttribArray(location);

        // Zeichne aktivierte und gebundene Daten als Dreiecke
        gl.glDrawElements(GL3.GL_TRIANGLES, 3, GL3.GL_UNSIGNED_INT, 0);

        // Setze Backbuffer nach vorn
        canvas.swapBuffers();
    }

    // JOGL Methode, wird bei Initialisierung aufgerufen
    public void init(GLAutoDrawable drawable) {
        drawable.addGLEventListener(this);
        gl = drawable.getGL().getGL3();

        // Erzeuge serverseitige Datenstrukturen für Per-Vertex Daten,
        // hier Position und Farbe. (siehe Abschnitt 21.6.1)
        IntBuffer bufferNames = BufferUtil.newIntBuffer(3);
        gl.glGenBuffers(3, bufferNames);
        FloatBuffer vertexCoords = BufferUtil.newFloatBuffer(
                            new float[]{0, 1, 0, -1, -1, 0, 1, -1, 0});
        coordBuffer = bufferNames.get(0);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
        gl.glBufferData(GL3.GL_ARRAY_BUFFER, 9*4, vertexCoords,
                        GL3.GL_STATIC_DRAW);
        FloatBuffer vertexColors = BufferUtil.newFloatBuffer(new float[]
                    {1, 0, 0, 0, 1, 0, 0, 0, 1});//rot, gruen, blau
        colorBuffer = bufferNames.get(1);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, colorBuffer);
        gl.glBufferData(GL3.GL_ARRAY_BUFFER, 9*4, vertexColors,
                        GL3.GL_STATIC_DRAW);
```

```
        // Erzeuge einen IndexBuffer.  (siehe Abschnitt 21.6.2)
        IntBuffer indices = BufferUtil.newIntBuffer(new int[] {0, 1, 2});
        int indexBuffer = bufferNames.get(2);
        gl.glBindBuffer(GL3.GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
        gl.glBufferData(GL3.GL_ELEMENT_ARRAY_BUFFER, 3*4, indices,
                        GL3.GL_STATIC_DRAW );

        // Vertex Shader zum Durchreichen von Position und Farbe
        String vsCode[] = {" #version 140                    \n"
                         + " in vec3 vPosition;              \n"
                         + " in vec3 vColor;                 \n"
                         + " out vec3 color;                 \n"
                         + " void main() {                   \n"
                         + "  color = vColor;                \n"
                         + "  gl_Position.xyz = vPosition;   \n"
                         + "  gl_Position.w = 1.0;           \n"
                         + " }                               \n"};

        // Fragment Shader zum Einfärben mit der interpolierten
        // Vertex Farbe
        String fsCode[] = {" #version 140                    \n"
                         + " in vec3 color;                  \n"
                         + " out vec4 fragColor;             \n"
                         + " void main(){                    \n"
                         + "  fragColor.rgb = color;         \n"
                         + "  fragColor.a = 1.0;             \n"
                         + " }                               \n"};

        // Erzeugen eines Program Objects bestehend aus obigen Shadern
        // Siehe Abschnitt 21.10
        int vertexShader = LoadShader(GL3.GL_VERTEX_SHADER, vsCode);
        int fragmentShader = LoadShader(GL3.GL_FRAGMENT_SHADER, fsCode);
        programObject = gl.glCreateProgram();
        gl.glAttachShader(programObject, vertexShader);
        gl.glAttachShader(programObject, fragmentShader);
        gl.glBindFragDataLocation(programObject, 0, "fragColor");
        gl.glLinkProgram(programObject);
        gl.glUseProgram(programObject);

        new FPSAnimator(canvas, 60).start();
    }

    // JOGL Methode, wird bei Veränderung des Fensters aufgerufen
    public void reshape(GLAutoDrawable drawable, int x, int y, int width,
            int height) {
        requestFocusInWindow();
        setSize(width, height);
    }

    public void dispose(GLAutoDrawable arg0) {}
}
```

Die Ausgabe des Dreiecks mit Farbinformationen für die Vertices ist in Abbildung 21.7 zu sehen. Einer der Gründe für die Kürze des Beispiels ist der Verzicht auf eine Projektion der Vertices. Dementsprechend müssen deren Koordinaten direkt im Bildraum angegeben werden.

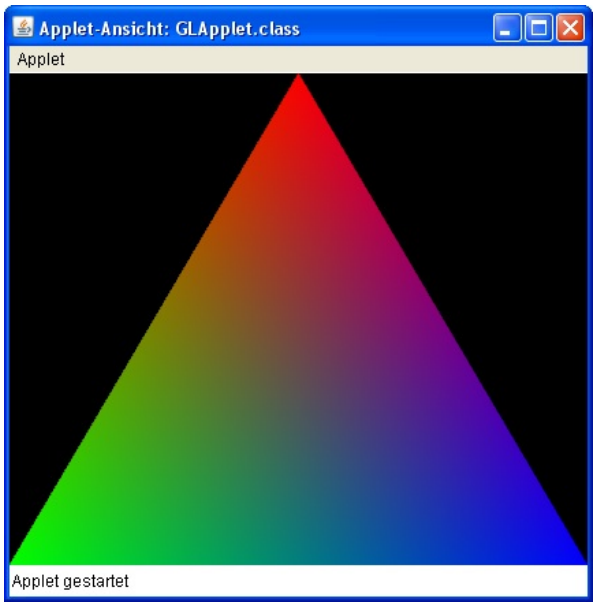


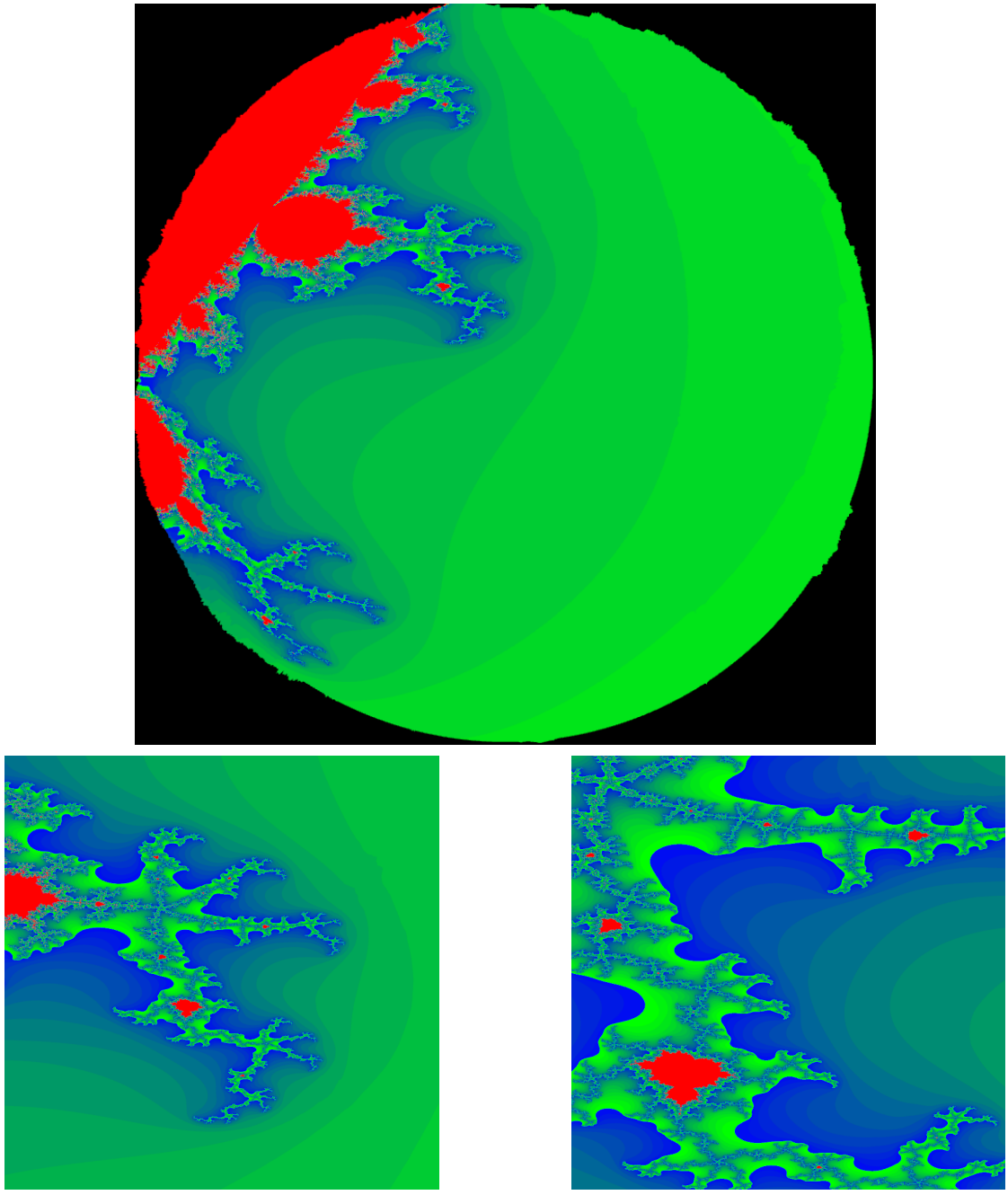

Abbildung 21.7: Ein Dreieck, eingefärbt mit den interpolierten Farben der Vertices

Als abschließendes Beispiel enthält das folgende Listing einen Fragment Shader zur Einfärbung einer Oberfläche mit einem Mandelbrot Fraktal:

```
#version 140
const float maxIterations = 100.0;
const vec3 innerColor  = vec3(1.0, 0.0, 0.0);
const vec3 outerColor1 = vec3(0.0, 1.0, 0.0);
const vec3 outerColor2 = vec3(0.0, 0.0, 1.0);

// Berechnet nur in Abhängigkeit von den Texturkoordinaten...
in vec2 texCoords;

// ...die Farbe des Fragments
out vec4 myFragColor;

void main() {
    float real  = texCoords.x; float imag = texCoords.y;
    float cReal = real; float cImag = imag;
    float r2 = 0.0;
    float iter;
    for(iter = 0.0; iter < maxIterations && r2 < 4.0; iter++) {
    float tempreal = real;
        real = (tempreal * tempreal) - ( imag * imag) + cReal;
        imag = 2.0 * tempreal * imag + cImag;
        r2   = (real * real) + (imag * imag);
    }
    vec3 color;
    if (r2 < 4.0)
        color = innerColor;
    else // mix ist eine GLSL-Funktion für lineare Interpolation
        color = mix(outerColor1, outerColor2, fract(iter * 0.05));
    myFragColor = vec4(color, 1.0);
}
```

Dieser Shader berechnet die Oberflächenfarbe, wie der Fragment Shader aus Abschnitt 21.9.2, in Abhängigkeit der Texturkoordinate und kann folglich an dessen Stelle im Beispielprogramm eingesetzt werden. Die Oberfläche der auch in Abb. 21.1 dargestellten Erdkugel sieht dann, in verschiedenen

Zoomstufen, wie folgt aus:



## 21.12 Literatur

Der begrenzte Rahmen in dieser Veranstaltung ermöglicht leider keine vollständige Einführung in OpenGL und die Shaderprogrammierung. So mussten nicht nur viele wichtige Themengebiete ausgeklammert werden, auch die Beschreibung der einzelnen OpenGL Funktionen hat größtenteils Bei-

spielcharakter und beschreibt diese nicht vollständig. Nachfolgend einige Vorschläge zur weiteren Vertiefung:

- **Shreiner: „OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1“, ISBN-13: 978-0321552624**
  Dieses Werk, auch unter dem Namen Redbook bekannt, stellte in früheren Versionen eine sehr lesbare Einführung in OpenGL dar. Es ist für die nicht abwärtskompatible GL Version 3.1 leider nicht komplett überarbeitet, sondern nur an einigen Stellen erweitert worden. Weiterhin sind die entfernten Befehle noch enthalten und werden mit Erläuterungen zu existierenden Funktionen vermischt. Trotzdem kann es mangels verständlich formulierter Alternativen unter Vorbehalt empfohlen werden. Hinweis: Dieses Buch ist in Safari verfügbar.

- **OpenGL 4.0 Core Profile Specification**
  Sicherlich nicht so verständlich geschrieben und mit Beispielen versehen wie das Redbook, enthält die Spezifikation konsequent überarbeitete und detaillierte Ausführungen zur aktuellen OpenGL Version.

- **Rost: „OpenGL Shading Language, Third Edition“, ISBN-13: 978-0321637635**
  Sehr lesenswerte Einführung in GLSL 1.40, die außerdem einige interessante weiterführende Themen anspricht und viele Beispiele enthält. Ein OpenGL (3.1) Überblick ist in diesem auch als Orange Book bezeichneten Werk ebenfalls vorhanden, dieser bleibt jedoch für den Einstieg ohne vorherige Erfahrung etwas knapp und ist auch nicht vollständig. Hinweis: Dieses Buch ist in Safari verfügbar.

- **OpenGL Shading Language 4.00 Specification**
  Die Spezifikation der aktuellen GLSL Version 4.0 kann ergänzend zum Orange Book gelesen werden, insbesondere wenn neuere Features wie der Geometry Shader oder Tessellation eingesetzt werden sollen.

- **Munshi: „OpenGL ES 2.0 Programming Guide“, ISBN-13: 978-0321502797**
  Gute Einführung in die aktuelle Version von OpenGL for Embedded Systems, welche vor allem im mobilen Bereich zum Einsatz kommt. OpenGL ES 2.0 ist OpenGL 3.1 hinsichtlich des Konzepts, des Funktionsumfangs und des Codes sehr ähnlich. Hinweis: Dieses Buch ist in Safari verfügbar.

Die in Safari verfügbaren Bücher sind von einem Rechner des Netzes der Uni Osnabrück aus erreichbar unter unter:

```
http://proquest.safaribooksonline.com/?uicode=osnabrueck
```

Die Spezifikationen zu OpenGL und GLSL sind verfügbar unter:

```
http://www.opengl.org/documentation/current_version/
```

Abschließend muss noch eine Warnung vor Büchern sowie zahlreichen im Netz verfügbaren Beispielen, Texten und Tutorials ausgesprochen werden, die nicht mindesten OpenGL 3.1 und GLSL 1.4 thematisieren. Der Code ist nicht kompatibel und die beschriebenen Konzepte haben unter Umständen mit den Aktuellen wenig gemein.